Table of Contents

Micronaut

Natively Cloud Native

Version: 3.1.3

1 Introduction

Micronaut is a modern, JVM-based, full stack Java framework designed for building modular, easily testable JVM applications with support for Java, Kotlin, and Groovy.

Micronaut is developed by the creators of the Grails framework and takes inspiration from lessons learnt over the years building real-world applications from monoliths to microservices using Spring, Spring Boot and Grails.

Micronaut aims to provide all the tools necessary to build JVM applications including:

- Dependency Injection and Inversion of Control (IoC)
- Aspect Oriented Programming (AOP)
- Sensible Defaults and Auto-Configuration

With Micronaut you can build Message-Driven Applications, Command Line Applications, HTTP Servers and more whilst for Microservices in particular Micronaut also provides:

- Distributed Configuration
- Service Discovery
- HTTP Routing
- Client-Side Load Balancing

At the same time Micronaut aims to avoid the downsides of frameworks like Spring, Spring Boot and Grails by providing:

- Fast startup time
- Reduced memory footprint
- Minimal use of reflection

- Minimal use of proxies
- No runtime bytecode generation
- Easy Unit Testing

Historically, frameworks such as Spring and Grails were not designed to run in scenarios such as serverless functions, Android apps, or low memory footprint microservices. In contrast, Micronaut is designed to be suitable for all of these scenarios.

This goal is achieved through the use of Java's **annotation processors** (https://docs.oracle.com/javase/8/docs/api/javax/annotation/processing/Processor.html), which are usable on any JVM language that supports them, as well as an HTTP Server and Client built on **Netty** (https://netty.io/). To provide a similar programming model to Spring and Grails, these annotation processors precompile the necessary metadata to perform DI, define AOP proxies and configure your application to run in a low-memory environment.

Many APIs in Micronaut are heavily inspired by Spring and Grails. This is by design, and helps bring developers up to speed quickly.

1.1 What's New?

3.1.0

Core Features

Primitive Beans

Factory Beans can now create beans that are primitive types or primitive array types.

See the section on **Primitive Beans and Arrays** in the documentation for more information.

Repeatable Qualifiers

Qualifiers can now be repeatable (an annotation annotated with `java.lang.annotation.Repeatable`) allowing narrowing bean resolution by a complete or partial match of the qualifiers declared on the injection point.

InjectScope

A new **@InjectScope** annotation has been added which destroys any beans with no defined scope and injected into a method or constructor annotated with `@Inject` after the method or constructor completes.

More Build Time Optimizations

Further build time metadata optimizations have been added included reducing the number and size of the classes generated to support **Bean Introspection** and including knowledge of repeatable annotations in generated metadata avoiding further reflective calls and optimizing Micronaut's memory usage, in particular with GraalVM.

Improvements to Context Propagation

Support for **Reactive context propagation** has been further improved by inclusion of request context information in the **Reactor context** (https://projectreactor.io/docs/core/release/reference/#context) and **documentation on how to effectively propagate the context across reactive flows** when using Kotlin coroutines.

Improvements to the Element API

The build-time **Element** API has been improved in a number of ways:

- New methods were added to the **MethodElement** API to resolve the retriever type and throws declaration
- A new experimental API has been added to the **ClassElement** API to resolve generic placeholders and resolve the generic bound to the element

HTTP Features

Filter By Regex

HTTP filters now support matching URLs by a regular expression. Set the `patternStyle` member of the annotation to `REGEX` and the value will be treated as a regular expression.

Random Port Binding

The way the server binds to random ports has improved and should result in fewer port binding exceptions in tests.

Client Data Formatting

The **@Format** annotation now supports several new values that can be used in conjunction with the declarative HTTP client to support formatting data in several new ways. See the **client parameters** documentation for more information.

StreamingFileUpload

The **StreamingFileUpload** API has been improved to support streaming directly to an output stream. As with the other `transferTo` methods, the write to the stream is offloaded to the IO pool automatically.

Server SSL Configuration

The SSL configuration for the Netty server now responds to refresh events. This allows for swapping out certificates without having to restart the server. See the **https documentation** for information on how to trigger the refresh.

New Netty Server API

If you wish to programmatically start additional Netty servers on different ports with potentially different configurations, new APIs have been added to do so including a new **NettyEmbeddedServerFactory** interface.

See the documentation on **Starting Secondary Servers** for more information.

Deprecations

The `netty.responses.file.*` configuration is deprecated in favor of `micronaut.server.netty.responses.file.*`. The old configuration key will be removed in the next major version of the framework.

Module Upgrades

Micronaut Data 3.1.0

- Kotlin's coroutines support. New repository interface `CoroutineCrudRepository`.
- Support for `AttributeConverter`
- R2DBC upgraded to `Arabba-SR11`
- JPA Criteria specifications

Micronaut JAX-RS 3.1

The **JAX-RS module** (https://micronaut-projects.github.io/micronaut-jaxrs/latest/guide/) now integrated with Micronaut Security allowing binding of the JAX-RS `SecurityContext`

Micronaut Kubernetes 3.1.0

Micronaut Kubernetes 3.1 introduces new annotation **@Informer** (https://micronaut-projects.github.io/micronaut-kubernetes/latest/api/io/micronaut/kubernetes/client/informer/Informer.html). By using the annotation on the **ResourceEventHandler** (https://javadoc.io/doc/io.kubernetes/client-java/latest/io/kubernetes/client/informer/ResourceEventHandler.html) the Micronaut will instantiate the **SharedInformer** (https://javadoc.io/doc/io.kubernetes/client-java/latest/io/kubernetes/client/informer/SharedIndexInformer.html) from the official **Kubernetes Java SDK** (https://github.com/kubernetes-client/java). Then you only need to take care of handling the changes of the watched Kubernetes resource. See more on **Kubernetes Informer** (https://micronaut-projects.github.io/micronaut-kubernetes/latest/guide/#kubernetes-informer).

Micronaut Oracle Coherence 3.0.0

The **Micronaut Oracle Coherence** (https://micronaut-projects.github.io/micronaut-coherence/latest/guide/) module is now out of preview status and includes broad integration with Oracle Coherence including support for caching, messaging and Micronaut Data.

3.0.0

Core Features

Optimized Build-Time Metadata

Micronaut 3.0 introduces a new build time metadata format that is more efficient in terms of startup and code size.

The result is significant improvements to startup and native image sizes when building native images with GraalVM Native Image.

It is recommended that users re-compile their applications and libraries with Micronaut 3.0 to benefit from these changes.

Support for GraalVM 21.2

Micronaut has been updated to support the latest GraalVM 21.2 release.

Jakarta Inject

The `jakarta.inject` annotations are now the default injection annotations for Micronaut 3

Support for JSR-330 Bean Import

Using the **@Import** annotation it is now possible to import bean definitions into your application where JSR-330 (either `javax.inject` or `jakarta.inject` annotations) are used in an external library.

See the documentation on **Bean Import** for more information.

Support for Controlling Annotation Inheritance

AnnotationMetadata inheritance can now be controlled via Java's `@Inherited` annotation. If an annotation is not explicitly annotated with `@Inherited` it will not be included in the metadata. See the **Annotation Inheritance** section of the documentation for more information.

This is an important behavioural change from Micronaut 2.x, see the **Breaking Changes** section for information on how to upgrade.

Support Narrowing Injection by Generic Type Arguments

Micronaut can now resolve the correct bean to inject based on the generic type arguments specified on the injection point:

Java	Groovy	Kotlin

```
    @Inject
    public Vehicle(Engine<V8> engine) {
        this.engine = engine;
    }
```

For more information see the section on **Qualifying by Generic Type Arguments**.

Support for using Annotation Members in Qualifiers

You can now use annotation members in qualifiers and specify which members should be excluded with the new **@NonBinding** annotation.

For more information see the section on **Qualifying By Annotation Members**.

Support for Limiting the Injectable Types

You can now limit the exposed types of a bean using the `typed` member of the **@Bean** annotation:

Java | Groovy | Kotlin

```
@Singleton
@Bean(typed = Engine.class)  1
public class V8Engine implements Engine {  2
    @Override
    public String start() {
        return "Starting V8";
    }

    @Override
    public int getCylinders() {
        return 8;
    }
}
```

For more information see the section on **Limiting Injectable Types**.

Factories can produce bean from fields

Beans defined with the **@Factory** annotation can now produce beans from public or package protected fields, for example:

Java | Groovy | Kotlin

```
@MicronautTest
public class VehicleMockSpec {
    @Requires(beans=VehicleMockSpec.class)
    @Bean @Replaces(Engine.class)
    Engine mockEngine = () -> "Mock Started";  1

    @Inject Vehicle vehicle;  2

    @Test
    void testStartEngine() {
        final String result = vehicle.start();
        assertEquals("Mock Started", result);  3
    }
}
```

For more information see the **Bean Factories** section of the documentation.

Enhanced `BeanProvider` Interface

The **BeanProvider** interface has been enhanced with new methods such as `iterator()` and `stream()` as well as methods to check for bean existence and uniqueness.

New `@Any` Qualifier for use in Bean Factories

A new **@Any** qualifier has been introduced to allow injecting any available instance into an injection point and can be used in combination with the new `BeanProvider` interface mentioned above to allow more dynamic behaviour.

Using BeanProvider with Any

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.BeanProvider;
import io.micronaut.context.annotation.Any;
import jakarta.inject.Singleton;

@Singleton
public class Vehicle {
    final BeanProvider<Engine> engineProvider;

    public Vehicle(@Any BeanProvider<Engine> engineProvider) {  1
        this.engineProvider = engineProvider;
    }
    void start() {
        engineProvider.ifPresent(Engine::start);  2
    }
}
```

Copy to Clipboard

The annotation can also be used on **@Factory** methods to allow customization of how objects are injected via the **InjectionPoint** API.

Support for Fields in Bean Introspections

Bean introspections on public or package protected fields are now supported:

Java Groovy

JAVA

```
import io.micronaut.core.annotation.Introspected;

@Introspected(accessKind = Introspected.AccessKind.FIELD)
public class User {
    public final String name;  1
    public int age = 18;  2

    public User(String name) {
        this.name = name;
    }
}
```

Copy to Clipboard

For more information see the "Bean Fields" section of the **Bean Introspections** documentation.

`ApplicationEventPublisher` has now a generic event type

For the performance reasons it's advised to inject an instance of `ApplicationEventPublisher` with a generic type parameter - `ApplicationEventPublisher<MyEvent>`.

AOP Features

Support for Constructor Interception

It is now possible to intercept bean construction invocations through the **ConstructorInterceptor** interface and **@AroundConstruct** annotation.

See the section on **Bean Life Cycle Advice** for more information.

Support for `@PostConstruct` & `@PreDestroy` Interception

It is now possible to intercept `@PostConstruct` and `@PreDestroy` method invocations through the **MethodInterceptor** interface and **@InterceptorBinding** annotation.

See the section on **Bean Life Cycle Advice** for more information.

Random Configuration Values

It is now possible to set a max and a range for random numbers in configuration. For example to set an integer between 0 and 9, `${random.int(10)}` can be used as the configuration value. See the **documentation** under "Using Random Properties" for more information.

Project Reactor used internally instead of RxJava2

Micronaut 3 uses internally **Project Reactor** (https://projectreactor.io) instead **RxJava 2** (https://github.com/ReactiveX/RxJava). Project Reactor allows Micronaut 3 to simplify instrumentation, thanks to **Reactor's Context** (https://projectreactor.io/docs/core/release/api/reactor/util/context/Context.html), simplifies conversion login and eases the integration with R2DBC drivers. We recommend users to migrate to Reactor. However, it is possible to continue to use RxJava. See **Reactive Programming section**.

Module Upgrades

Micronaut Data 3.1.0

- Kotlin's coroutines support. New repository interface `CoroutineCrudRepository`.
- Support for `AttributeConverter`
- R2DBC upgraded to `Arabba-SR11`
- JPA Criteria specifications

Micronaut Micrometer 4.0.0

The **Micrometer module** (https://micronaut-projects.github.io/micronaut-micrometer/latest/guide/) has been upgraded and now supports repeated definitions of the **@Timed** (https://micrometer.io/docs/concepts#_the_timed_annotation) annotation as well as also supporting the `@Counted` annotation for counters when you add the `micronaut-micrometer-annotation` dependency to your annotation processor classpath.

Micronaut Oracle Cloud 2.0.0

Micronaut's **Oracle Cloud Integration** (https://micronaut-projects.github.io/micronaut-oracle-cloud/latest/guide/) has been updated with support for Cloud Monitoring and Tracing.

Micronaut Cassandra 4.0.0

The **Micronaut Cassandra** (https://micronaut-projects.github.io/micronaut-cassandra/latest/guide/) integration now includes support for GraalVM out of the box.

Other Modules

- Micronaut Acme 3.0.0
- Micronaut Aws 3.0.0
- Micronaut Azure 3.0.0
- Micronaut Cache 3.0.0
- Micronaut Discovery Client 3.0.0
- Micronaut ElasticSearch 3.0.0
- Micronaut Flyway 4.1.0
- Micronaut GCP 4.0.0
- Micronaut GraphQL 3.0.0
- Micronaut Groovy 3.0.0
- Micronaut Grpc 3.0.0
- Micronaut Jackson XML 3.0.0
- Micronaut Jaxrs 3.0.0
- Micronaut JMX 3.0.0
- Micronaut Kafka 4.0.0
- Micronaut Kotlin 3.0.0
- Micronaut Kubernetes 3.0.0
- Micronaut Liquibase 4.0.2
- Micronaut Mongo 4.0.0
- Micronaut MQTT 2.0.0
- Micronaut Multitenancy 4.0.0
- Micronaut Nats Io 3.0.0
- Micronaut Neo4j 5.0.0
- Micronaut OpenApi 3.0.1
- Micronaut Picocli 4.0.0
- Micronaut Problem Json 2.0.0
- Micronaut R2DBC 2.0.0
- Micronaut RabbitMQ 3.0.0
- Micronaut Reactor 2.0.0
- Micronaut Redis 5.0.0
- Micronaut RSS 3.0.0
- Micronaut RxJava2 1.0.0 (new)
- Micronaut RxJava3 2.0.0
- Micronaut Security 3.0.0
- Micronaut Servlet 3.0.0
- Micronaut Spring 4.0.0
- Micronaut SQL 4.0.0
- Micronaut Test 3.0.0
- Micronaut Views 3.0.0

Dependency Upgrades

- Caffeine 2.9.1
- Cassandra 4.11.1

- Elasticsearch 7.12.0
- Flyway 7.12.1
- GraalVM 21.2.0
- H2 Database 1.4.200
- Hazelcast 4.2.1
- Hibernate 5.5.3.Final
- Hikari 4.0.3
- Infinispan 12.1.6.Final
- Jackson 2.12.4
- Jaeger 1.6.0
- Jakarta Annotation API 2.0.0
- JAsync 1.2.2
- JDBI 3.20.1
- JOOQ 3.14.12
- JUnit 5.7.2
- Kafka 2.8.0
- Kotlin 1.5.21
- Kotlin Coroutines 1.5.1
- Ktor 1.6.1
- Liquibase 4.4.3
- MariaDB Driver 2.7.3
- Micrometer 1.7.1
- MongoDB 4.3.0
- MS SQL Driver 9.2.1.jre8
- MySQL Driver 8.0.25
- Neo4j Driver 4.2.7
- Postgres Driver 42.2.23
- Reactor 3.4.8
- RxJava3 3.0.13
- SLF4J 1.7.29
- Snake YAML 1.29
- Spock 2.0-groovy-3.0
- Spring 5.3.9
- Spring Boot 2.5.3
- Testcontainers 1.15.3
- Tomcat JDBC 10.0.8
- Vertx SQL Drivers 4.1.1

1.2 Upgrading to Micronaut 3.x

This section covers the steps required to upgrade a Micronaut 2.x application to Micronaut 3.0.0.

The sections below go into more detail, but at a high level the process generally involves:

- updating versions
- updating annotations
- choosing a Reactive implementation
- adjusting code affected by breaking changes

Typically, upgrading should be straightforward, but it's possible to save yourself some work with **OpenRewrite** (https://docs.openrewrite.org/), an automated refactoring tool that you can use to make many of the required upgrade changes.

Automating Upgrades with OpenRewrite

OpenRewrite works with Micronaut applications written in Java, but OpenRewrite doesn't currently support Kotlin or Groovy. Like any automated tool it does much of the work for you, but be sure to review the resulting changes and manually make any changes that aren't supported by OpenRewrite, for example converting from RxJava2 to Reactor.

If you will be using OpenRewrite, don't make any upgrade changes yet that would cause your application not to compile, for example updating the Micronaut version to 3.x. This would cause application classes that use `javax.inject` annotations like `@Singleton` or RxJava2 classes like `io.reactivex.Flowable` to not compile since those dependencies are no longer included by default. Instead, use OpenRewrite to do the initial work and just do the steps yourself that aren't possible or practical to automate.

Adding OpenRewrite support to your build is easy, it just requires adding the Gradle or Maven plugin and configuring the plugin to use the Micronaut upgrade recipe.

See the **Gradle feature diff** (https://micronaut.io/launch?features=openrewrite&lang=JAVA&build=GRADLE&activity=diff) or the **Maven feature diff** (https://micronaut.io/launch?features=openrewrite&lang=JAVA&build=MAVEN&activity=diff) to see the required build script changes.

Once you've made the build script changes, you can "dry-run" the Micronaut upgrade recipe to see what changes would be made.

For Gradle, run

```
$ ./gradlew rewriteDryRun
```
BASH

and view the diff report generated in `build/reports/rewrite/rewrite.patch`

and for Maven, run

```
$ ./mvnw rewrite:dryRun
```
BASH

and view the diff report generated in `target/site/rewrite/rewrite.patch`.

Then you can run the recipe for real, letting OpenRewrite update your code.

For Gradle, run

```
$ ./gradlew rewriteRun
```
BASH

and for Maven, run

```
$ ./mvnw rewrite:run
```
BASH

Once the changes have been made, you could remove the plugin, but it's fine to leave it since OpenRewrite doesn't run automatically, only when you run one of its commands. And there are many more recipes available beyond the Micronaut upgrade recipe that you might want to include to automate other code changes.

The plugin includes another command to list all recipes currently in the classpath (in this case the core recipes plus those added by the `rewrite-micronaut` module).

For Gradle, run

```
$ ./gradlew rewriteDiscover
```
BASH

and for Maven, run

```
$ ./mvnw rewrite:discover
```
BASH

and the available recipes and styles will be output to the console. Check out the **OpenRewrite documentation** (https://docs.openrewrite.org/) for more information and to see the many other available recipes available.

Version Update

If you use Gradle, update the `micronautVersion` property in `gradle.properties`, e.g.

gradle.properties

```
micronautVersion=3.1.3
```
PROPERTIES

If you use Maven, update the parent POM version and `micronaut.version` property in `pom.xml`, e.g.

pom.xml

XML

```
<parent>
  <groupId>io.micronaut</groupId>
  <artifactId>micronaut-parent</artifactId>
  <version>3.1.3</version>
</parent>

<properties>
  ...
  <micronaut.version>3.1.3</micronaut.version>
  ...
</properties>
```

Build Plugin Update

If you use the Micronaut Gradle plugin, update the version to `2.0.3`

```
id("io.micronaut.application") version "2.0.3"
```

For Maven users the plugin version is updated automatically when you update the Micronaut version.

Inject Annotations

The `javax.inject` annotations are no longer a transitive dependency. Micronaut now ships with the Jakarta inject annotations. Either replace all `javax.inject` imports with `jakarta.inject`, or add a dependency on `javax-inject` to continue using the older annotations:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

Copy to Clipboard

Any code that relied on the `javax.inject` annotations being present in the annotation metadata will still work as expected, however any code that interacts with them must be changed to no longer reference the annotation classes themselves. Static variables in the **AnnotationUtil** class (e.g. `AnnotationUtil.INJECT`, `AnnotationUtil.SINGLETON`, etc.) should be used in place of the annotation classes when working with annotation metadata.

Nullability Annotations

Micronaut now only comes with its own set of annotations to declare nullability. The findbugs, javax, and jetbrains annotations are all still supported, however you must add a dependency to use them. Either switch to the Micronaut **@Nullable** / **@NonNull** annotations or add a dependency for the annotation library you wish to use.

RxJava2

Micronaut no longer ships any reactive implementation as a default in any of our modules or core libraries. Upgrading to Micronaut 3 requires choosing which reactive streams implementation to use, and then adding the relevant dependency.

For those already using RxJava3 or Project Reactor, there should be no changes required to upgrade to Micronaut 3. If you use RxJava2 and wish to continue using it, you must add a dependency:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.rxjava2</groupId>
    <artifactId>micronaut-rxjava2</artifactId>
</dependency>
```

Copy to Clipboard

In addition, if any of the `Rx` HTTP client interfaces were used, a dependency must be added and the imports must be updated.

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.rxjava2</groupId>
    <artifactId>micronaut-rxjava2-http-client</artifactId>
</dependency>
```

Copy to Clipboard

Table 1. RxJava2 HTTP Client Imports

Old	New
io.micronaut.http.client.RxHttpClient	io.micronaut.rxjava2.http.client.RxHttpClient
io.micronaut.http.client.RxProxyHttpClient	io.micronaut.rxjava2.http.client.proxy.RxProxyHttpClient

Old	New
io.micronaut.http.client.RxStreamingHttpClient	io.micronaut.rxjava2.http.client.RxStreamingHttpClient
io.micronaut.http.client.sse.RxSseClient	io.micronaut.rxjava2.http.client.sse.RxSseClient
io.micronaut.websocket.RxWebSocketClient	io.micronaut.rxjava2.http.client.websockets.RxWebSocketClient

If the Netty based server implementation is being used, an additional dependency must be added:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.rxjava2</groupId>
    <artifactId>micronaut-rxjava2-http-server-netty</artifactId>
</dependency>
```

Copy to Clipboard

We recommend switching to Project Reactor as that is the implementation used internally by Micronaut. Adding a dependency to RxJava2 will result in both implementations in the runtime classpath of your application.

Breaking Changes

Review the section on **Breaking Changes** and update your affected application code.

2 Quick Start

The following sections walk you through a Quick Start on how to use Micronaut to setup a basic "Hello World" application.

Before getting started ensure you have a Java 8 or higher JDK installed, and it is recommended that you use a suitable IDE such as IntelliJ IDEA.

To follow the Quick Start it is also recommended that you have the Micronaut CLI installed.

2.1 Build/Install the CLI

The best way to install Micronaut on Unix systems is with **SDKMAN** (http://sdkman.io/) which greatly simplifies installing and managing multiple Micronaut versions.

To see all available installation methods, check the **Micronaut Starter documentation** (https://micronaut-projects.github.io/micronaut-starter/latest/guide/#installation).

2.2 Creating a Server Application

Although not required to use Micronaut, the Micronaut CLI is the quickest way to create a new server application.

Using the CLI you can create a new Micronaut application in either Groovy, Java, or Kotlin (the default is Java).

The following command creates a new "Hello World" server application in Java with a Gradle build:

Applications generated via our CLI include Gradle or Maven wrappers, so it is not even necessary to have Gradle or Maven installed on your machine to begin running the applications. Simply use the `mvnw` or `gradlew` command, as explained further below.

BASH

```
$ mn create-app hello-world
```

Supply `--build maven` to create a Maven-based build instead

The previous command creates a new Java application in a directory called `hello-world` featuring a Gradle build. You can run the application with `./gradlew run`:

BASH

```
$ ./gradlew run
> Task :run
[main] INFO  io.micronaut.runtime.Micronaut - Startup completed in 972ms. Server Running: http://localhost:28933
```

If you have created a Maven-based project, use `./mvnw mn:run` instead.

For Windows the ./ before commands is not needed

By default the Micronaut HTTP server is configured to run on port 8080. See the section **Running Server on a Specific Port** for more options.

To create a service that responds to "Hello World" you first need a controller. The following is an example:

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.MediaType;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;

@Controller("/hello")  1
public class HelloController {

    @Get(produces = MediaType.TEXT_PLAIN)  2
    public String index() {
        return "Hello World";  3
    }
}
```

Copy to Clipboard

1 The **@Controller** annotation defines the class as a controller mapped to the path `/hello`

2 The **@Get** annotation maps the `index` method to all requests that use an HTTP `GET`

3 A String "Hello World" is returned as the response

If you use Java, place the previous file in `src/main/java/hello/world`.
If you use Groovy, place the previous file in `src/main/groovy/hello/world`.
If you use Kotlin, place the previous file in `src/main/kotlin/hello/world`.

If you start the application and send a `GET` request to the `/hello` URI, the text "Hello World" is returned:

BASH

```
$ curl http://localhost:8080/hello
Hello World
```

2.3 Setting up an IDE

The application created in the previous section contains a main class located in `src/main/java` that looks like the following:

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.runtime.Micronaut;

public class Application {

    public static void main(String[] args) {
        Micronaut.run(Application.class);
    }
}
```

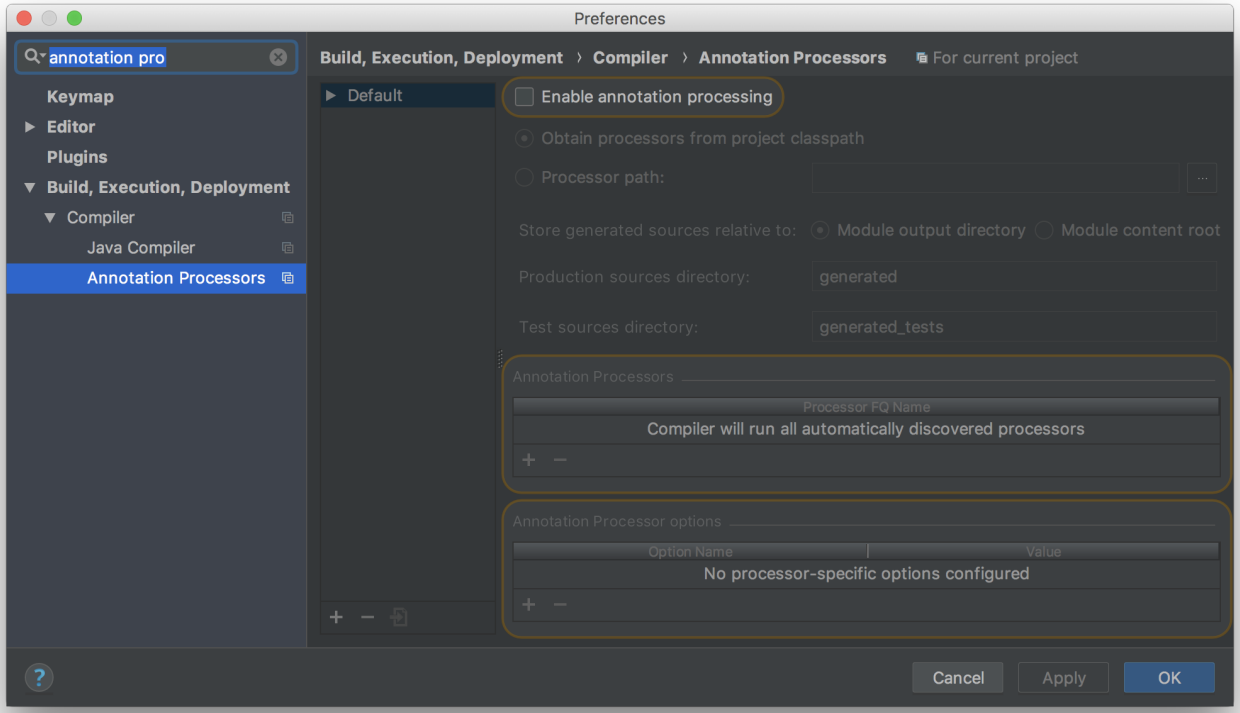
Copy to Clipboard

This is the class that is run when starting the application via Gradle/Maven or via deployment. You can also run the main class directly within your IDE.

2.3.1 IntelliJ IDEA

To import a Micronaut project into IntelliJ IDEA, open the `build.gradle` or `pom.xml` file and follow the instructions to import the project.

For IntelliJ IDEA, if you plan to use the IntelliJ compiler, enable annotation processing under "Build, Execution, Deployment → Compiler → Annotation Processors" by ticking the "Enable annotation processing" checkbox:



Once you have enabled annotation processing in IntelliJ you can run the application and tests directly within the IDE without the need of an external build tool such as Gradle or Maven.

2.3.2 Eclipse

To use Eclipse IDE, it is recommended you import your Micronaut project into Eclipse using either **Gradle BuildShip** (https://projects.eclipse.org/projects/tools.buildship) for Gradle or **M2Eclipse** (http://www.eclipse.org/m2e/) for Maven.

Micronaut requires Eclipse IDE 4.9 or higher

Eclipse and Gradle

Once you have set up Eclipse 4.9 or higher with **Gradle BuildShip** (https://projects.eclipse.org/projects/tools.buildship), first run the `gradle eclipse` task from the root of your project, then import the project by selecting `File → Import` and choosing `Gradle → Existing Gradle Project` and navigating to the root directory of your project (where the `build.gradle` file is located).

Eclipse and Maven

For Eclipse 4.9 and above with Maven you need the following Eclipse plugins:

- **M2Eclipse for Maven** (http://www.eclipse.org/m2e/)
- **Maven integration with Eclipse JDT Annotation Processor Toolkit** (https://github.com/jbosstools/m2e-apt)

Once these are installed, import the project by selecting `File → Import` and choosing `Maven → Existing Maven Project` and navigating to the root directory of your project (where the `pom.xml` file is located).

Then enable annotation processing by opening `Eclipse → Preferences` and navigating to `Maven → Annotation Processing` and selecting the option `Automatically configure JDT APT`.

2.3.3 Visual Studio Code

Micronaut can be set up within Visual Studio Code in one of two ways.

Option 1) GraalVM Extension Pack for Java

The prefered way is using the **GraalVM Extension Pack for Java** (https://marketplace.visualstudio.com/items?itemName=oracle-labs-graalvm.graalvm-pack) which ships with an **Apache NetBeans** (https://netbeans.apache.org/) Language server.

It is not possible to have both the official **Java Extension Pack** (https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-pack) and the GraalVM Extension Pack for Java installed at the same time so if you prefer the former, skip this section and go to Option 2)

The **GraalVM Tools for Java** (https://marketplace.visualstudio.com/items?itemName=oracle-labs-graalvm.graalvm) are preferred since they delegate to the build system for running applications and tests which means there is no additional setup or differences with regards to how javac is configured for annotation processing when compared to the **Java Extension Pack** (https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-pack) which is based on the Eclipse compiler.

The GraalVM Extension Pack also includes the **GraalVM Tools for Micronaut** (https://marketplace.visualstudio.com/items?itemName=oracle-labs-graalvm.micronaut) extension which features:

- An application creation wizard
- Code completion for YAML configuration
- Pallet commands to build, deploy, create Native Images etc.

Option 2) Red Hat/Microsoft Java Extension Pack

First install the **Java Extension Pack** (https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-pack).

You can also optionally install **STS** (https://marketplace.visualstudio.com/items?itemName=Pivotal.vscode-spring-boot) to enable code completion for `application.yml`.

If you use Gradle, prior to opening the project in VSC run the following command from a terminal window:

BASH

```
./gradlew eclipse
```

If you don't run the above command beforehand then annotation processing will not be configured correctly and the application will not work.

Once the extension pack is installed and if you have setup **terminal integration** (https://code.visualstudio.com/docs/setup/mac) just type `code .` in any project directory and the project will be automatically setup.

2.4 Creating a Client

As mentioned previously, Micronaut includes both an **HTTP server** and an **HTTP client**. A **low-level HTTP client** is provided which you can use to test the `HelloController` created in the previous section.

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.HttpRequest;
import io.micronaut.http.client.HttpClient;
import io.micronaut.http.client.annotation.Client;
import io.micronaut.runtime.server.EmbeddedServer;
import io.micronaut.test.extensions.junit5.annotation.MicronautTest;
import org.junit.jupiter.api.Test;

import jakarta.inject.Inject;

import static org.junit.jupiter.api.Assertions.assertEquals;

@MicronautTest
public class HelloControllerSpec {

    @Inject
    EmbeddedServer server; 1

    @Inject
    @Client("/")
    HttpClient client; 2

    @Test
    void testHelloWorldResponse() {
        String response = client.toBlocking() 3
                .retrieve(HttpRequest.GET("/hello"));
        assertEquals("Hello World", response); 4
    }
}
```

Copy to Clipboard

1 The **EmbeddedServer** is configured as a shared test field

2 A **HttpClient** instance shared field is also defined

3 The test uses the `toBlocking()` method to make a blocking call

4 The `retrieve` method returns the controller response as a `String`

In addition to a low-level client, Micronaut features a **declarative, compile-time HTTP client**, powered by the **Client** annotation.

To create a client, create an interface annotated with `@Client`, for example:

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.MediaType;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.client.annotation.Client;
import org.reactivestreams.Publisher;
import io.micronaut.core.async.annotation.SingleResult;

@Client("/hello")  1
public interface HelloClient {

    @Get(consumes = MediaType.TEXT_PLAIN)  2
    @SingleResult
    Publisher<String> hello();  3
}
```

Copy to Clipboard

1 The `@Client` annotation is used with a value that is a relative path to the current server

2 The same **@Get** annotation used on the server is used to define the client mapping

3 A `Publisher` annotated with `SingleResult` is returned with the value read from the server

To test the `HelloClient`, retrieve it from the **ApplicationContext** associated with the server:

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.test.extensions.junit5.annotation.MicronautTest;
import org.junit.jupiter.api.Test;
import io.micronaut.core.async.annotation.SingleResult;

import jakarta.inject.Inject;
import reactor.core.publisher.Mono;

import static org.junit.jupiter.api.Assertions.assertEquals;

@MicronautTest  1
public class HelloClientSpec  {

    @Inject
    HelloClient client;  2

    @Test
    public void testHelloWorldResponse(){
        assertEquals("Hello World", Mono.from(client.hello()).block());  3
    }
}
```

Copy to Clipboard

1 The `@MicronautTest` annotation defines the test

2 The `HelloClient` is injected from the **ApplicationContext**

3 The client is invoked using the **Project Reactor** **(https://projectreactor.io)** `Mono::block` method

The **Client** annotation produces an implementation automatically for you at compile time without the using proxies or runtime reflection.

The **Client** annotation is very flexible. See the section on the **Micronaut HTTP Client** for more information.

2.5 Deploying the Application

To deploy a Micronaut application you create an executable JAR file by running `./gradlew assemble` or `./mvnw package`.

The constructed JAR file can then be executed with `java -jar`. For example:

BASH

```
$ java -jar build/libs/hello-world-all.jar
```

if building with Gradle, or

```bash
$ java -jar target/hello-world.jar
```

if building with Maven.

The executable JAR can be run locally, or deployed to a virtual machine or managed Cloud service that supports executable JARs.

To publish a layered application to a Docker container registry, configure your Docker image name in `build.gradle` for Gradle:

```groovy
dockerBuild {
    images = ["[REPO_URL]/[NAMESPACE]/my-image:$project.version"]
}
```

Then use `dockerPush` to push a built image of the application:

```bash
$ ./gradlew dockerPush
```

For Maven, define the following plugin in your POM:

```xml
<plugin>
  <groupId>com.google.cloud.tools</groupId>
  <artifactId>jib-maven-plugin</artifactId>
  <configuration>
    <to>
      <image>docker.io/my-company/my-image:${project.version}</image>
    </to>
  </configuration>
</plugin>
```

Then invoke the `deploy` lifecycle phase specifying the packaging type as either `docker` or `docker-native`:

```bash
$ ./mvnw deploy -Dpackaging=docker
```

3 Inversion of Control

Unlike other frameworks which rely on runtime reflection and proxies, Micronaut uses compile time data to implement dependency injection.

This is a similar approach taken by tools such as Google **Dagger** (https://google.github.io/dagger/), which is designed primarily with Android in mind. Micronaut, on the other hand, is designed for building server-side microservices and provides many of the same tools and utilities as other frameworks but without using reflection or caching excessive amounts of reflection metadata.

The goals of the Micronaut IoC container are summarized as:

- Use reflection as a last resort
- Avoid proxies
- Optimize start-up time
- Reduce memory footprint
- Provide clear, understandable error handling

Note that the IoC part of Micronaut can be used completely independently of Micronaut for whatever application type you wish to build.

To do so, configure your build to include the `micronaut-inject-java` dependency as an annotation processor.

The easiest way to do this is with Micronaut's Gradle or Maven plugins. For example with Gradle:

Configuring Gradle

GROOVY

```
plugins {
    id 'io.micronaut.library' version '1.3.2'  1
}

version "0.1"
group "com.example"

repositories {
    mavenCentral()
}

micronaut {
    version = "3.1.3"  2
}
```

1 Define the **Micronaut Library plugin** (https://plugins.gradle.org/plugin/io.micronaut.library)

2 Specify the Micronaut version to use

The entry point for IoC is then the **ApplicationContext** interface, which includes a `run` method. The following example demonstrates using it:

Running the `ApplicationContext`

JAVA

```
try (ApplicationContext context = ApplicationContext.run()) {  1
    MyBean myBean = context.getBean(MyBean.class);  2
    // do something with your bean
}
```

1 Run the **ApplicationContext**

2 Retrieve a bean from the `ApplicationContext`

The example uses Java **try-with-resources** (https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html) syntax to ensure the **ApplicationContext** is cleanly shutdown when the application exits.

3.1 Defining Beans

A bean is an object whose lifecycle is managed by the Micronaut IoC container. That lifecycle may include creation, execution, and destruction. Micronaut implements the **JSR-330 (javax.inject) - Dependency Injection for Java** (http://javax-inject.github.io/javax-inject/) specification, hence to use Micronaut you simply use the **annotations provided by javax.inject** (https://docs.oracle.com/javaee/6/api/javax/inject/package-summary.html).

The following is a simple example:

Java	Groovy	Kotlin

JAVA

```
public interface Engine {  1
    int getCylinders();
    String start();
}

@Singleton 2
public class V8Engine implements Engine {
    private int cylinders = 8;

    @Override
    public String start() {
        return "Starting V8";
    }

    @Override
    public int getCylinders() {
        return cylinders;
    }

    public void setCylinders(int cylinders) {
        this.cylinders = cylinders;
    }
}

@Singleton
public class Vehicle {
    private final Engine engine;

    public Vehicle(Engine engine) { 3
        this.engine = engine;
    }

    public String start() {
        return engine.start();
    }
}
```

Copy to Clipboard

1 A common `Engine` interface is defined

2 A `V8Engine` implementation is defined and marked with `Singleton` scope

3 The `Engine` is injected via constructor injection

To perform dependency injection, run the **BeanContext** using the `run()` method and lookup a bean using `getBean(Class)`, as per the following example:

Java | Groovy | Kotlin

JAVA

```
final BeanContext context = BeanContext.run();
Vehicle vehicle = context.getBean(Vehicle.class);
System.out.println(vehicle.start());
```

Copy to Clipboard

Micronaut automatically discovers dependency injection metadata on the classpath and wires the beans together according to injection points you define.

Micronaut supports the following types of dependency injection:

- Constructor injection (must be one public constructor or a single constructor annotated with `@Inject`)
- Field injection
- JavaBean property injection
- Method parameter injection

3.2 How Does it Work?

At this point, you may be wondering how Micronaut performs the above dependency injection without requiring reflection.

The key is a set of AST transformations (for Groovy) and annotation processors (for Java) that generate classes that implement the **BeanDefinition** interface.

Micronaut uses the ASM bytecode library to generate classes, and because Micronaut knows ahead of time the injection points, there is no need to scan all of the methods, fields, constructors, etc. at runtime like other frameworks such as Spring do.

Also, since reflection is not used when constructing the bean, the JVM can inline and optimize the code far better, resulting in better runtime performance and reduced memory consumption. This is particularly important for non-singleton scopes where application performance depends on bean creation performance.

In addition, with Micronaut your application startup time and memory consumption are not affected by the size of your codebase in the same way as with a framework that uses reflection. Reflection-based IoC frameworks load and cache reflection data for every single field, method, and constructor in your code. Thus as your code grows in size so do your memory requirements, whilst with Micronaut this is not the case.

3.3 The BeanContext

The **BeanContext** is a container object for all your bean definitions (it also implements **BeanDefinitionRegistry**).

It is also the point of initialization for Micronaut. Generally speaking however, you don't interact directly with the `BeanContext` API and can simply use `jakarta.inject` annotations and the annotations in the **io.micronaut.context.annotation** package for your dependency injection needs.

3.4 Injectable Container Types

In addition to being able to inject beans, Micronaut natively supports injecting the following types:

Table 1. Injectable Container Types

Type	Description	Example
java.util.Optional (https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)	An `Optional` of a bean. `empty()` is injected if the bean doesn't exist	`Optional<Engine>`
java.lang.Collection (https://docs.oracle.com/javase/8/docs/api/java/lang/Collection.html)	An `Collection` or subtype of `Collection` (e.g. `List`, `Set`, etc.)	`Collection<Engine>`
java.util.stream.Stream (https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)	A lazy `Stream` of beans	`Stream<Engine>`
Array (https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Array.html)	A native array of beans of a given type	`Engine[]`
Provider (https://docs.oracle.com/javaee/6/api/javax/inject/Provider.html)	A `javax.inject.Provider` if a circular dependency requires it, or to instantiate a prototype for each `get` call.	`Provider<Engine>`
Provider (https://jakarta.ee/specifications/platform/9/apidocs/jakarta/inject/Provider.html)	A `jakarta.inject.Provider` if a circular dependency requires it or to instantiate a prototype for each `get` call.	`Provider<Engine>`
BeanProvider	A `io.micronaut.context.BeanProvider` if a circular dependency requires it or to instantiate a prototype for each `get` call.	`BeanProvider<Engine>`

There are 3 different provider types supported, however the `BeanProvider` is the one we suggest to use.

When injecting a `java.lang.Collection`, or `java.util.stream.Stream`, `Array` of beans into a bean matching the injection type, then the owning bean will not be be a member of the injected collection. A common pattern demonstrating this is aggregation. For example:

JAVA

```
@Singleton
class AggregateEngine implements Engine {
  @Inject
  List<Engine> engines;

  @Override
  public void start() {
    engines.forEach(Engine::start);
  }

  ...
}
```

In this example, the injected member variable `engines` will not contain an instance of `AggregateEngine`

A prototype bean will have one instance created per place the bean is injected. When a prototype bean is injected as a provider, each call to `get()` creates a new instance.

Collection Ordering

When injecting a collection of beans, they are not ordered by default. Implement the **Ordered** interface to inject an ordered collection. If the requested bean type does not implement **Ordered**, Micronaut searches for the **@Order** annotation on beans.

The **@Order** annotation is especially useful for ordering beans created by factories where the bean type is a class in a third-party library. In this example, both `LowRateLimit` and `HighRateLimit` implement the `RateLimit` interface.

Factory with @Order

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.Factory;
import io.micronaut.core.annotation.Order;

import jakarta.inject.Singleton;
import java.time.Duration;

@Factory
public class RateLimitsFactory {

    @Singleton
    @Order(20)
    LowRateLimit rateLimit2() {
        return new LowRateLimit(Duration.ofMinutes(50), 100);
    }

    @Singleton
    @Order(10)
    HighRateLimit rateLimit1() {
        return new HighRateLimit(Duration.ofMinutes(50), 1000);
    }
}
```

Copy to Clipboard

When a collection of `RateLimit` beans are requested from the context, they are returned in ascending order based on the value in the annotation.

Injecting a Bean by Order

When injecting a single instance of a bean the **@Order** annotation can also be used to define which bean has the highest precedence and hence should be injected.

The **Ordered** interface is not taken into account when selecting a single instance as this would require instantiating the bean to resolve the order.

3.5 Bean Qualifiers

If you have multiple possible implementations for a given interface to inject, you need to use a qualifier.

Once again Micronaut leverages JSR-330 and the **Qualifier** (https://docs.oracle.com/javaee/6/api/javax/inject/Qualifier.html) and **Named** (https://docs.oracle.com/javaee/6/api/javax/inject/Named.html) annotations to support this use case.

Qualifying By Name

To qualify by name, use the **Named** (https://docs.oracle.com/javaee/6/api/javax/inject/Named.html) annotation. For example, consider the following classes:

Java | Groovy | Kotlin

JAVA

```java
public interface Engine {  1
    int getCylinders();
    String start();
}

@Singleton
public class V6Engine implements Engine {   2
    @Override
    public String start() {
        return "Starting V6";
    }

    @Override
    public int getCylinders() {
        return 6;
    }
}

@Singleton
public class V8Engine implements Engine {   3
    @Override
    public String start() {
        return "Starting V8";
    }

    @Override
    public int getCylinders() {
        return 8;
    }

}

@Singleton
public class Vehicle {
    private final Engine engine;

    @Inject
    public Vehicle(@Named("v8") Engine engine) { 4
        this.engine = engine;
    }

    public String start() {
        return engine.start(); 5
    }
}
```

Copy to Clipboard

1. The `Engine` interface defines the common contract
2. The `V6Engine` class is the first implementation
3. The `V8Engine` class is the second implementation
4. The **javax.inject.Named** (https://docs.oracle.com/javaee/6/api/javax/inject/Named.html) annotation indicates that the `V8Engine` implementation is required
5. Calling the start method prints: `"Starting V8"`

Micronaut is capable of injecting `V8Engine` in the previous example, because:

`@Named` qualifier value (`v8`) + type being injected simple name (`Engine`) == (case-insensitive) == The simple name of a bean of type `Engine` (`V8Engine`)

You can also declare **@Named** (https://docs.oracle.com/javaee/6/api/javax/inject/Named.html) at the class level of a bean to explicitly define the name of the bean.

Qualifying By Annotation

In addition to being able to qualify by name, you can build your own qualifiers using the **Qualifier** (https://docs.oracle.com/javaee/6/api/javax/inject/Qualifier.html) annotation. For example, consider the following annotation:

Java	Groovy	Kotlin

JAVA

```
import jakarta.inject.Qualifier;
import java.lang.annotation.Retention;

import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Qualifier
@Retention(RUNTIME)
public @interface V8 {
}
```

Copy to Clipboard

The above annotation is itself annotated with the `@Qualifier` annotation to designate it as a qualifier. You can then use the annotation at any injection point in your code. For example:

Java | Groovy | Kotlin

JAVA

```
    @Inject Vehicle(@V8 Engine engine) {
        this.engine = engine;
    }
```

Copy to Clipboard

Qualifying By Annotation Members

Since Micronaut 3.0, annotation qualifiers can also use annotation members to resolve the correct bean to inject. For example, consider the following annotation:

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.NonBinding;
import jakarta.inject.Qualifier;
import java.lang.annotation.Retention;

import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Qualifier  1
@Retention(RUNTIME)
public @interface Cylinders {
    int value();

    @NonBinding  2
    String description() default "";
}
```

Copy to Clipboard

1 The `@Cylinders` annotation is meta-annotated with `@Qualifier`

2 The annotation has two members. The **@NonBinding** annotation is used to exclude the `description` member from being considered during dependency resolution.

You can then use the `@Cylinders` annotation on any bean and the members that are not annotated with **@NonBinding** are considered during dependency resolution:

Java | Groovy | Kotlin

JAVA

```
@Singleton
@Cylinders(value = 6, description = "6-cylinder V6 engine")   1
public class V6Engine implements Engine {  2

    @Override
    public int getCylinders() {
        return 6;
    }

    @Override
    public String start() {
        return "Starting V6";
    }
}
```

Copy to Clipboard

1 Here the `value` member is set to 6 for the `V6Engine` type

2 The class implements an `Engine` interface

Java | Groovy | Kotlin

JAVA

```
@Singleton
@Cylinders(value = 8, description = "8-cylinder V8 engine")  1
public class V8Engine implements Engine {  2
    @Override
    public int getCylinders() {
        return 8;
    }

    @Override
    public String start() {
        return "Starting V8";
    }
}
```

Copy to Clipboard

1 Here the `value` member is set to 8 for the `V8Engine` type

2 The class implements an `Engine` interface

You can then use the `@Cylinders` qualifier on any injection point to select the correct bean to inject. For example:

Java | Groovy | Kotlin

JAVA

```
@Inject Vehicle(@Cylinders(8) Engine engine) {
    this.engine = engine;
}
```

Copy to Clipboard

Qualifying by Generic Type Arguments

Since Micronaut 3.0, it is possible to select which bean to inject based on the generic type arguments of the class or interface. Consider the following example:

Java | Groovy | Kotlin

JAVA

```
public interface CylinderProvider {
    int getCylinders();
}
```

Copy to Clipboard

The `CylinderProvider` interface provides the number of cylinders.

Java | Groovy | Kotlin

JAVA

```
public interface Engine<T extends CylinderProvider> {  1
    default int getCylinders() {
        return getCylinderProvider().getCylinders();
    }

    default String start() {
        return "Starting " + getCylinderProvider().getClass().getSimpleName();
    }

    T getCylinderProvider();
}
```

Copy to Clipboard

1 The engine class defines a generic type argument `<T>` that must be an instance of `CylinderProvider`

You can define implementations of the `Engine` interface with different generic type arguments. For example for a V6 engine:

Java | Groovy | Kotlin

JAVA

```
public class V6 implements CylinderProvider {
    @Override
    public int getCylinders() {
        return 7;
    }
}
```

Copy to Clipboard

The above defines a `V6` class that implements the `CylinderProvider` interface.

Java | Groovy | Kotlin

JAVA

```
@Singleton
public class V6Engine implements Engine<V6> {   1
    @Override
    public V6 getCylinderProvider() {
        return new V6();
    }
}
```

Copy to Clipboard

1 The `V6Engine` implements `Engine` providing `V6` as a generic type parameter

And a V8 engine:

Java | Groovy | Kotlin

JAVA

```
public class V8 implements CylinderProvider {
    @Override
    public int getCylinders() {
        return 8;
    }
}
```

Copy to Clipboard

The above defines a `V8` class that implements the `CylinderProvider` interface.

Java | Groovy | Kotlin

JAVA

```
@Singleton
public class V8Engine implements Engine<V8> {   1
    @Override
    public V8 getCylinderProvider() {
        return new V8();
    }
}
```

Copy to Clipboard

1 The `V8Engine` implements `Engine` providing `V8` as a generic type parameter

You can then use the generic arguments when defining the injection point and Micronaut will pick the correct bean to inject based on the specific generic type arguments:

Java | Groovy | Kotlin

JAVA

```
    @Inject
    public Vehicle(Engine<V8> engine) {
        this.engine = engine;
    }
```

Copy to Clipboard

In the above example the `V8Engine` bean is injected.

Primary and Secondary Beans

Primary is a qualifier that indicates that a bean is the primary bean to be selected in the case of multiple interface implementations.

Consider the following example:

Java | Groovy | Kotlin

JAVA

```
public interface ColorPicker {
    String color();
}
```

Copy to Clipboard

`ColorPicker` is implemented by these classes:

The Primary Bean

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.Primary;
import jakarta.inject.Singleton;

@Primary
@Singleton
class Green implements ColorPicker {

    @Override
    public String color() {
        return "green";
    }
}
```

Copy to Clipboard

The `Green` bean class implements `ColorPicker` and is annotated with `@Primary`.

Another Bean of the Same Type

Java | Groovy | Kotlin

JAVA

```
import jakarta.inject.Singleton;

@Singleton
public class Blue implements ColorPicker {

    @Override
    public String color() {
        return "blue";
    }
}
```

Copy to Clipboard

The `Blue` bean class also implements `ColorPicker` and hence you have two possible candidates when injecting the `ColorPicker` interface. Since `Green` is the primary, it will always be favoured.

Java | Groovy | Kotlin

JAVA

```
@Controller("/testPrimary")
public class TestController {

    protected final ColorPicker colorPicker;

    public TestController(ColorPicker colorPicker) {  1
        this.colorPicker = colorPicker;
    }

    @Get
    public String index() {
        return colorPicker.color();
    }
}
```

Copy to Clipboard

1 Although there are two `ColorPicker` beans, `Green` gets injected due to the `@Primary` annotation.

If multiple possible candidates are present and no `@Primary` is defined a **NonUniqueBeanException** is thrown.

In addition to `@Primary`, there is also a **Secondary** annotation which causes the opposite effect and allows de-prioritizing a bean.

Injecting Any Bean

If you are not particular about which bean gets injected then you can use the **@Any** qualifier which will inject the first available bean, for example:

Injecting Any Instance

Java | Groovy | Kotlin

JAVA

```
    @Inject @Any
    Engine engine;
```

Copy to Clipboard

The **@Any** qualifier is typically used in conjunction with the **BeanProvider** interface to allow more dynamic use cases. For example the following `Vehicle` implementation will start the `Engine` if the bean is present:

Using BeanProvider with Any

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.BeanProvider;
import io.micronaut.context.annotation.Any;
import jakarta.inject.Singleton;

@Singleton
public class Vehicle {
    final BeanProvider<Engine> engineProvider;

    public Vehicle(@Any BeanProvider<Engine> engineProvider) {  1
        this.engineProvider = engineProvider;
    }
    void start() {
        engineProvider.ifPresent(Engine::start);  2
    }
}
```

Copy to Clipboard

1 Use `@Any` to inject the **BeanProvider**

2 Call the `start` method if the underlying bean is present using the `ifPresent` method

If there are multiple beans you can also adapt the behaviour. The following example starts all the engines installed in the `Vehicle` if any are present:

Using BeanProvider with Any

Java | Groovy | Kotlin

JAVA

```
    void startAll() {
        if (engineProvider.isPresent()) {  1
            engineProvider.stream().forEach(Engine::start);  2
        }
    }
```

Copy to Clipboard

1 Check if any beans present

2 If so iterate over each one via the `stream().forEach(..)` method, starting the engines

3.6 Limiting Injectable Types

By default when you annotate a bean with a scope such as `@Singleton` the bean class and all interfaces it implements and super classes it extends from become injectable via `@Inject`.

Consider the following example from the previous section on defining beans:

Java | Groovy | Kotlin

JAVA

```
@Singleton
public class V8Engine implements Engine {  3
    @Override
    public String start() {
        return "Starting V8";
    }

    @Override
    public int getCylinders() {
        return 8;
    }

}
```

Copy to Clipboard

In the above case other classes in your application can choose to either inject the interface `Engine` or the concrete implementation `V8Engine`.

If this is undesirable you can use the `typed` member of the **@Bean** annotation to limit the exposed types. For example:

Java | Groovy | Kotlin

JAVA

```
@Singleton
@Bean(typed = Engine.class)  1
public class V8Engine implements Engine {  2
    @Override
    public String start() {
        return "Starting V8";
    }

    @Override
    public int getCylinders() {
        return 8;
    }
}
```

Copy to Clipboard

1 `@Bean(typed=..)` is used to only allow injection the interface `Engine` and not the concrete type

2 The class must implement the class or interface defined by `typed` otherwise a compilation error will occur

The following test demonstrates the behaviour of `typed` using programmatic lookup and the **BeanContext** API:

Java | Groovy | Kotlin

JAVA

```
@MicronautTest
public class EngineSpec {
    @Inject
    BeanContext beanContext;

    @Test
    public void testEngine() {
        assertThrows(NoSuchBeanException.class, () ->
                beanContext.getBean(V8Engine.class)  1
        );
        final Engine engine = beanContext.getBean(Engine.class);  2
        assertTrue(engine instanceof V8Engine);
    }
}
```

Copy to Clipboard

1 Trying to lookup `V8Engine` throws a **NoSuchBeanException**

2 Whilst looking up the `Engine` interface succeeds

3.7 Scopes

Micronaut features an extensible bean scoping mechanism based on JSR-330. The following default scopes are supported:

3.7.1 Built-In Scopes

Table 1. Micronaut Built-in Scopes

Type	Description
@Singleton (https://docs.oracle.com/javaee/6/api/javax/inject/Singleton.html)	Singleton scope indicates only one instance of the bean will exist
@Context	Context scope indicates that the bean will be created at the same time as the `ApplicationContext` (eager initialization)
@Prototype	Prototype scope indicates that a new instance of the bean is created each time it is injected
@Infrastructure	Infrastructure scope represents a bean that cannot be overridden or replaced using **@Replaces** because it is critical to the functioning of the system.
@ThreadLocal	`@ThreadLocal` scope is a custom scope that associates a bean per thread via a ThreadLocal
@Refreshable	`@Refreshable` scope is a custom scope that allows a bean's state to be refreshed via the `/refresh` endpoint.
@RequestScope	`@RequestScope` scope is a custom scope that indicates a new instance of the bean is created and associated with each HTTP request

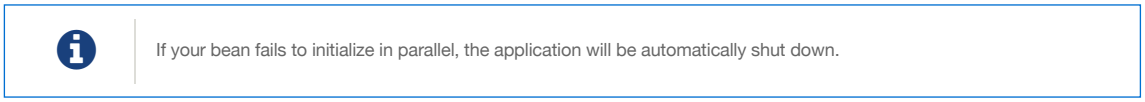
The **@Prototype** annotation is a synonym for **@Bean** because the default scope is prototype.

Additional scopes can be added by defining a `@Singleton` bean that implements the **CustomScope** interface.

Note that when starting an **ApplicationContext**, by default `@Singleton`-scoped beans are created lazily and on-demand. This is by design to optimize startup time.

If this presents a problem for your use case you have the option of using the **@Context** annotation which binds the lifecycle of your object to the lifecycle of the **ApplicationContext**. In other words when the **ApplicationContext** is started your bean will be created.

Alternatively, annotate any `@Singleton`-scoped bean with **@Parallel** which allows parallel initialization of your bean without impacting overall startup time.



3.7.1.1 Eager Initialization of Singletons

Eager initialization of `@Singleton` beans maybe desirable in certain scenarios, such as on AWS Lambda where more CPU resources are assigned to Lambda construction than execution.

You can specify whether to eagerly initialize `@Singleton`-scoped beans using the **ApplicationContextBuilder** interface:

Enabling Eager Initialization of Singletons

JAVA

```
public class Application {

    public static void main(String[] args) {
        Micronaut.build(args)
            .eagerInitSingletons(true)  1
            .mainClass(Application.class)
            .start();
    }
}
```

1 Setting eager init to `true` initializes all singletons

When you use Micronaut in environments such as **Serverless Functions**, you will not have an Application class and instead you extend a Micronaut-provided class. In those cases, Micronaut provides methods which you can override to enhance the **ApplicationContextBuilder**

Override of newApplicationContextBuilder()

JAVA

```
public class MyFunctionHandler extends MicronautRequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEver
...
    @Nonnull
    @Override
    protected ApplicationContextBuilder newApplicationContextBuilder() {
        ApplicationContextBuilder builder = super.newApplicationContextBuilder();
        builder.eagerInitSingletons(true);
        return builder;
    }
    ...
}
```

@ConfigurationReader beans such as **@EachProperty** or **@ConfigurationProperties** are singleton beans. To eagerly init configuration but keep other `@Singleton`-scoped bean creation lazy, use `eagerInitConfiguration`:

Enabling Eager Initialization of Configuration

JAVA

```
public class Application {

    public static void main(String[] args) {
        Micronaut.build(args)
            .eagerInitConfiguration(true)  1
            .mainClass(Application.class)
            .start();
    }
}
```

1 Setting eager init to true initializes all configuration reader beans.

3.7.2 Refreshable Scope

The **Refreshable** scope is a custom scope that allows a bean's state to be refreshed via:

- `/refresh` endpoint.

- Publication of a **RefreshEvent**.

The following example illustrates `@Refreshable` scope behavior.

Java | Groovy | Kotlin

JAVA

```
@Refreshable 1
public static class WeatherService {
    private String forecast;

    @PostConstruct
    public void init() {
        forecast = "Scattered Clouds " + new SimpleDateFormat("dd/MMM/yy HH:mm:ss.SSS").format(new Date()); 2
    }

    public String latestForecast() {
        return forecast;
    }
}
```

Copy to Clipboard

1 The `WeatherService` is annotated with `@Refreshable` scope which stores an instance until a refresh event is triggered

2 The value of the `forecast` property is set to a fixed value when the bean is created and won't change until the bean is refreshed

If you invoke `latestForecast()` twice, you will see identical responses such as `"Scattered Clouds 01/Feb/18 10:29.199"`.

When the `/refresh` endpoint is invoked or a **RefreshEvent** is published, the instance is invalidated and a new instance is created the next time the object is requested. For example:

Java | Groovy | Kotlin

JAVA

```
applicationContext.publishEvent(new RefreshEvent());
```

Copy to Clipboard

3.7.3 Scopes on Meta Annotations

Scopes can be defined on meta annotations that you can then apply to your classes. Consider the following example meta annotation:

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.Requires;

import jakarta.inject.Singleton;
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;

import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Requires(classes = Car.class) 1
@Singleton 2
@Documented
@Retention(RUNTIME)
public @interface Driver {
}
```

Copy to Clipboard

1 The scope declares a requirement on a `Car` class using **Requires**

2 The annotation is declared as `@Singleton`

In the example above the `@Singleton` annotation is applied to the `@Driver` annotation which results in every class that is annotated with `@Driver` being regarded as singleton.

Note that in this case it is not possible to alter the scope when the annotation is applied. For example, the following will not override the scope declared by `@Driver` and is invalid:

Declaring Another Scope

JAVA

```
@Driver
@Prototype
class Foo {}
```

For the scope to be overridable, instead use the **DefaultScope** annotation on `@Driver` which allows a default scope to be specified if none other is present:

Using @DefaultScope

Java | Groovy | Kotlin

JAVA

```
@Requires(classes = Car.class)
@DefaultScope(Singleton.class)  1
@Documented
@Retention(RUNTIME)
public @interface Driver {
}
```

Copy to Clipboard

1 **DefaultScope** declares the scope to use if none is specified

3.8 Bean Factories

In many cases, you may want to make available as a bean a class that is not part of your codebase such as those provided by third-party libraries. In this case, you cannot annotate the compiled class. Instead, implement a **@Factory**.

A factory is a class annotated with the **Factory** annotation that provides one or more methods annotated with a bean scope annotation. Which annotation you use depends on what scope you want the bean to be in. See the section on **bean scopes** for more information.

The return types of methods annotated with a bean scope annotation are the bean types. This is best illustrated by an example:

Java | Groovy | Kotlin

JAVA

```
@Singleton
class CrankShaft {
}

class V8Engine implements Engine {
    private final int cylinders = 8;
    private final CrankShaft crankShaft;

    public V8Engine(CrankShaft crankShaft) {
        this.crankShaft = crankShaft;
    }

    @Override
    public String start() {
        return "Starting V8";
    }
}

@Factory
class EngineFactory {

    @Singleton
    Engine v8Engine(CrankShaft crankShaft) {
        return new V8Engine(crankShaft);
    }
}
```

Copy to Clipboard

In this case, a `V8Engine` is created by the `EngineFactory` class' `v8Engine` method. Note that you can inject parameters into the method and they will be resolved as beans. The resulting `V8Engine` bean will be a singleton.

A factory can have multiple methods annotated with bean scope annotations, each one returning a distinct bean type.

If you take this approach you should not invoke other bean methods internally within the class. Instead, inject the types via parameters.

To allow the resulting bean to participate in the application context shutdown process, annotate the method with **@Bean** and set the `preDestroy` argument to the name of the method to be called to close the bean.

Programmatically Disabling Beans

Factory methods can throw **DisabledBeanException** to conditionally disable beans. Using **@Requires** should always be the preferred method to conditionally create beans; throwing an exception in a factory method should only be done if using **@Requires** is not possible.

For example:

Java | Groovy | Kotlin

JAVA

```
public interface Engine {
    Integer getCylinders();
}

@EachProperty("engines")
public class EngineConfiguration implements Toggleable {

    private boolean enabled = true;
    private Integer cylinders;

    @NotNull
    public Integer getCylinders() {
        return cylinders;
    }

    public void setCylinders(Integer cylinders) {
        this.cylinders = cylinders;
    }

    @Override
    public boolean isEnabled() {
        return enabled;
    }

    public void setEnabled(boolean enabled) {
        this.enabled = enabled;
    }

}

@Factory
public class EngineFactory {

    @EachBean(EngineConfiguration.class)
    public Engine buildEngine(EngineConfiguration engineConfiguration) {
        if (engineConfiguration.isEnabled()) {
            return engineConfiguration::getCylinders;
        } else {
            throw new DisabledBeanException("Engine configuration disabled");
        }
    }
}
```

Copy to Clipboard

Injection Point

A common use case with factories is to take advantage of annotation metadata from the point at which an object is injected such that behaviour can be modified based on said metadata.

Consider an annotation such as the following:

Java	Groovy	Kotlin

JAVA

```
@Documented
@Retention(RUNTIME)
@Target(ElementType.PARAMETER)
public @interface Cylinders {
    int value() default 8;
}
```

Copy to Clipboard

The above annotation could be used to customize the type of engine we want to inject into a vehicle at the point at which the injection point is defined:

Java	Groovy	Kotlin

JAVA

```
@Singleton
class Vehicle {

    private final Engine engine;

    Vehicle(@Cylinders(6) Engine engine) {
        this.engine = engine;
    }

    String start() {
        return engine.start();
    }
}
```

Copy to Clipboard

The above `Vehicle` class specifies an annotation value of `@Cylinders(6)` indicating an `Engine` of six cylinders is required.

To implement this use case, define a factory that accepts the **InjectionPoint** instance to analyze the defined annotation values:

Java | Groovy | Kotlin

JAVA

```
@Factory
class EngineFactory {

    @Prototype
    Engine v8Engine(InjectionPoint<?> injectionPoint, CrankShaft crankShaft) {  1
        final int cylinders = injectionPoint
                .getAnnotationMetadata()
                .intValue(Cylinders.class).orElse(8);  2
        switch (cylinders) {  3
            case 6:
                return new V6Engine(crankShaft);
            case 8:
                return new V8Engine(crankShaft);
            default:
                throw new IllegalArgumentException("Unsupported number of cylinders specified: " + cylinders);
        }
    }
}
```

Copy to Clipboard

1. The factory method defines a parameter of type **InjectionPoint**.
2. The annotation metadata is used to obtain the value of the `@Cylinder` annotation
3. The value is used to construct an engine, throwing an exception if an engine cannot be constructed.

It is important to note that the factory is declared as **@Prototype** scope so the method is invoked for each injection point. If the `V8Engine` and `V6Engine` types are required to be singletons, the factory should use a Map to ensure the objects are only constructed once.

Beans from Fields

With Micronaut 3.0 or above it is also possible to produce beans from fields by declaring the **@Bean** annotation on a field.

Whilst generally this approach should be discouraged in favour for factory methods, which provide more flexibility it does simplify testing code. For example with bean fields you can easily produce mocks in your test code:

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.*;
import io.micronaut.test.extensions.junit5.annotation.MicronautTest;
import org.junit.jupiter.api.Test;
import jakarta.inject.Inject;

import static org.junit.jupiter.api.Assertions.assertEquals;

@MicronautTest
public class VehicleMockSpec {
    @Requires(beans=VehicleMockSpec.class)
    @Bean @Replaces(Engine.class)
    Engine mockEngine = () -> "Mock Started";  1

    @Inject Vehicle vehicle;  2

    @Test
    void testStartEngine() {
        final String result = vehicle.start();
        assertEquals("Mock Started", result);  3
    }
}
```

Copy to Clipboard

1 A bean is defined from a field that replaces the existing `Engine`.

2 The `Vehicle` is injected.

3 The code asserts that the mock implementation is called.

Note that only public or package protected fields are supported on non-primitive types. If the field is `static`, `private`, or `protected` a compilation error will occur.

Primitive Beans and Arrays

Since Micronaut 3.1 it is possible to define and inject primitive types and array types from factories.

For example:

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.Bean;
import io.micronaut.context.annotation.Factory;
import jakarta.inject.Named;

@Factory
class CylinderFactory {
    @Bean
    @Named("V8")  1
    final int v8 = 8;

    @Bean
    @Named("V6")  1
    final int v6 = 6;
}
```

Copy to Clipboard

1 Two primitive integer beans are defined with different names

Primitive beans can be injected like any other bean:

Java | Groovy | Kotlin

JAVA

```
import jakarta.inject.Named;
import jakarta.inject.Singleton;

@Singleton
public class V8Engine {
    private final int cylinders;

    public V8Engine(@Named("V8") int cylinders) {  1
        this.cylinders = cylinders;
    }

    public int getCylinders() {
        return cylinders;
    }
}
```

Copy to Clipboard

Note that primitive beans and primitive array beans have the following limitations:

- **AOP advice** cannot be applied to primitives or wrapper types
- Due to the above custom scopes that proxy are not supported
- The `@Bean(preDestroy=..)` member is not supported

3.9 Conditional Beans

At times you may want a bean to load conditionally based on various potential factors including the classpath, the configuration, the presence of other beans, etc.

The **Requires** annotation provides the ability to define one or many conditions on a bean.

Consider the following example:

Using @Requires

Java | Groovy | Kotlin

JAVA

```
@Singleton
@Requires(beans = DataSource.class)
@Requires(property = "datasource.url")
public class JdbcBookService implements BookService {

    DataSource dataSource;

    public JdbcBookService(DataSource dataSource) {
        this.dataSource = dataSource;
    }
```

Copy to Clipboard

The above bean defines two requirements. The first indicates that a `DataSource` bean must be present for the bean to load. The second requirement ensures that the `datasource.url` property is set before loading the `JdbcBookService` bean.

Kotlin currently does not support repeatable annotations. Use the `@Requirements` annotation when multiple requires are needed. For example, `@Requirements(Requires(…), Requires(…))`. See **https://youtrack.jetbrains.com/issue/KT-12794** to track this feature.

If multiple beans require the same combination of requirements, you can define a meta-annotation with the requirements:

Using a @Requires meta-annotation

Java | Groovy | Kotlin

JAVA

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.PACKAGE, ElementType.TYPE})
@Requires(beans = DataSource.class)
@Requires(property = "datasource.url")
public @interface RequiresJdbc {
}
```

Copy to Clipboard

In the above example the `RequiresJdbc` annotation can be used on the `JdbcBookService` instead:

Using a meta-annotation

JAVA

```
@RequiresJdbc
public class JdbcBookService implements BookService {
    ...
}
```

If you have multiple beans that need to fulfill a given requirement before loading, you may want to consider a bean configuration group, as explained in the next section.

Configuration Requirements

The **@Requires** annotation is very flexible and can be used for a variety of use cases. The following table summarizes some possibilities:

Table 1. Using @Requires

Requirement	Example
Require the presence of one or more classes	`@Requires(classes=javax.servlet.Servlet)`
Require the absence of one or more classes	`@Requires(missing=javax.servlet.Servlet)`

Requirement	Example
Require the presence one or more beans	`@Requires(beans=javax.sql.DataSource)`
Require the absence of one or more beans	`@Requires(missingBeans=javax.sql.DataSource)`
Require the environment to be applied	`@Requires(env="test")`
Require the environment to not be applied	`@Requires(notEnv="test")`
Require the presence of another configuration package	`@Requires(configuration="foo.bar")`
Require the absence of another configuration package	`@Requires(missingConfigurations="foo.bar")`
Require particular SDK version	`@Requires(sdk=Sdk.JAVA, value="1.8")`
Requires classes annotated with the given annotations to be available to the application via package scanning	`@Requires(entities=javax.persistence.Entity)`
Require a property with an optional value	`@Requires(property="data-source.url")`
Require a property to not be part of the configuration	`@Requires(missingProperty="data-source.url")`
Require the presence of one or more files in the file system	`@Requires(resources="file:/path/to/file")`
Require the presence of one or more classpath resources	`@Requires(resources="classpath:myFile.properties")`
Require the current operating system to be in the list	`@Requires(os={Requires.Family.WINDOWS})`
Require the current operating system to **not** be in the list	`@Requires(notOs={Requires.Family.WINDOWS})`

Additional Notes on Property Requirements.

Adding a requirement on a property has some additional functionality. You can require the property to be a certain value, not be a certain value, and use a default in those checks if it is not set.

JAVA
```
@Requires(property="foo")  1
@Requires(property="foo", value="John")  2
@Requires(property="foo", value="John", defaultValue="John")  3
@Requires(property="foo", notEquals="Sally")  4
```

1. Requires the property to be set
2. Requires the property to be "John"
3. Requires the property to be "John" or not set
4. Requires the property to not be "Sally" or not set

Debugging Conditional Beans

If you have multiple conditions and complex requirements it may become difficult to understand why a particular bean has not been loaded.

To help resolve issues with conditional beans you can enable debug logging for the `io.micronaut.context.condition` package which will log the reasons why beans were not loaded.

logback.xml

XML
```
<logger name="io.micronaut.context.condition" level="DEBUG"/>
```

Consult the logging chapter for details **howto setup logging**.

3.10 Bean Replacement

One significant difference between Micronaut's Dependency Injection system and Spring's is the way beans are replaced.

In a Spring application, beans have names and are overridden by creating a bean with the same name, regardless of the type of the bean. Spring also has the notion of bean registration order, hence in Spring Boot you have `@AutoConfigureBefore` and `@AutoConfigureAfter` annotations that control how beans override each other.

This strategy leads to problems that are difficult to debug, for example:

- Bean loading order changes, leading to unexpected results
- A bean with the same name overrides another bean with a different type

To avoid these problems, Micronaut's DI has no concept of bean names or load order. Beans have a type and a **Qualifier**. You cannot override a bean of a completely different type with another.

A useful benefit of Spring's approach is that it allows overriding existing beans to customize behaviour. To support the same ability, Micronaut's DI provides an explicit **@Replaces** annotation, which integrates nicely with support for **Conditional Beans** and clearly documents and expresses the intention of the developer.

Any existing bean can be replaced by another bean that declares **@Replaces**. For example, consider the following class:

JdbcBookService

Java | Groovy | Kotlin

JAVA

```
@Singleton
@Requires(beans = DataSource.class)
@Requires(property = "datasource.url")
public class JdbcBookService implements BookService {

    DataSource dataSource;

    public JdbcBookService(DataSource dataSource) {
        this.dataSource = dataSource;
    }
```

Copy to Clipboard

You can define a class in `src/test/java` that replaces this class just for your tests:

Using @Replaces

Java | Groovy | Kotlin

JAVA

```
@Replaces(JdbcBookService.class) 1
@Singleton
public class MockBookService implements BookService {

    Map<String, Book> bookMap = new LinkedHashMap<>();

    @Override
    public Book findBook(String title) {
        return bookMap.get(title);
    }
}
```

Copy to Clipboard

1 The `MockBookService` declares that it replaces `JdbcBookService`

Factory Replacement

The `@Replaces` annotation also supports a `factory` argument. That argument allows the replacement of factory beans in their entirety or specific types created by the factory.

For example, it may be desired to replace all or part of the given factory class:

BookFactory

Java | Groovy | Kotlin

JAVA

```
@Factory
public class BookFactory {

    @Singleton
    Book novel() {
        return new Book("A Great Novel");
    }

    @Singleton
    TextBook textBook() {
        return new TextBook("Learning 101");
    }
}
```

Copy to Clipboard

To replace a factory entirely, your factory methods must match the return types of all methods in the replaced factory.

In this example, `BookFactory#textBook()` is **not** replaced because this factory does not have a factory method that returns a `TextBook`.

CustomBookFactory

Java | Groovy | Kotlin

JAVA

```
@Factory
@Replaces(factory = BookFactory.class)
public class CustomBookFactory {

    @Singleton
    Book otherNovel() {
        return new Book("An OK Novel");
    }
}
```

Copy to Clipboard

To replace one or more factory methods but retain the rest, apply the `@Replaces` annotation on the method(s) and denote the factory to apply to.

TextBookFactory

Java | Groovy | Kotlin

JAVA

```
@Factory
public class TextBookFactory {

    @Singleton
    @Replaces(value = TextBook.class, factory = BookFactory.class)
    TextBook textBook() {
        return new TextBook("Learning 305");
    }
}
```

Copy to Clipboard

The `BookFactory#novel()` method will not be replaced because the TextBook class is defined in the annotation.

Default Implementation

When exposing an API, it may be desirable to not expose the default implementation of an interface as part of the public API. Doing so prevents users from being able to replace the implementation because they will not be able to reference the class. The solution is to annotate the interface with **DefaultImplementation** to indicate which implementation to replace if a user creates a bean that `@Replaces(YourInterface.class)`.

For example consider:

A public API contract

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.DefaultImplementation;

@DefaultImplementation(DefaultResponseStrategy.class)
public interface ResponseStrategy {
}
```

Copy to Clipboard

The default implementation

Java | Groovy | Kotlin

JAVA

```
import jakarta.inject.Singleton;

@Singleton
class DefaultResponseStrategy implements ResponseStrategy {

}
```

Copy to Clipboard

The custom implementation

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.Replaces;
import jakarta.inject.Singleton;

@Singleton
@Replaces(ResponseStrategy.class)
public class CustomResponseStrategy implements ResponseStrategy {

}
```

Copy to Clipboard

In the above example, the `CustomResponseStrategy` replaces the `DefaultResponseStrategy` because the **DefaultImplementation** annotation points to the `DefaultResponseStrategy`.

3.11 Bean Configurations

A bean **@Configuration** is a grouping of multiple bean definitions within a package.

The `@Configuration` annotation is applied at the package level and informs Micronaut that the beans defined with the package form a logical grouping.

The `@Configuration` annotation is typically applied to `package-info` classes. For example:

package-info.groovy

GROOVY

```
@Configuration
package my.package

import io.micronaut.context.annotation.Configuration
```

Where this grouping becomes useful is when the bean configuration is made conditional via the `@Requires` annotation. For example:

package-info.groovy

GROOVY

```
@Configuration
@Requires(beans = javax.sql.DataSource)
package my.package
```

In the above example, all bean definitions within the annotated package are only loaded and made available if a `javax.sql.DataSource` bean is present. This lets you implement conditional auto-configuration of bean definitions.

Java and Kotlin also support this functionality via `package-info.java`. Kotlin does not support a `package-info.kt` as of version 1.3.

3.12 Life-Cycle Methods

When The Context Starts

To invoke a method when a bean is constructed, use the `jakarta.annotation.PostConstruct` annotation:

Java	Groovy	Kotlin

JAVA

```java
import jakarta.annotation.PostConstruct;  1
import jakarta.inject.Singleton;

@Singleton
public class V8Engine implements Engine {

    private int cylinders = 8;
    private boolean initialized = false;  2

    @Override
    public String start() {
        if (!initialized) {
            throw new IllegalStateException("Engine not initialized!");
        }

        return "Starting V8";
    }

    @Override
    public int getCylinders() {
        return cylinders;
    }

    public boolean isInitialized() {
        return initialized;
    }

    @PostConstruct  3
    public void initialize() {
        initialized = true;
    }
}
```

Copy to Clipboard

1 The `PostConstruct` annotation is imported

2 A field is defined that requires initialization

3 A method is annotated with `@PostConstruct` and will be invoked once the object is constructed and fully injected.

To manage when a bean is constructed, see the section on **bean scopes**.

When The Context Closes

To invoke a method when the context is closed, use the `javax.annotation.PreDestroy` annotation:

Java | Groovy | Kotlin

JAVA

```java
import jakarta.annotation.PreDestroy;  1
import jakarta.inject.Singleton;
import java.util.concurrent.atomic.AtomicBoolean;

@Singleton
public class PreDestroyBean implements AutoCloseable {

    AtomicBoolean stopped = new AtomicBoolean(false);

    @PreDestroy  2
    @Override
    public void close() throws Exception {
        stopped.compareAndSet(false, true);
    }
}
```

Copy to Clipboard

1 The `PreDestroy` annotation is imported

2 A method is annotated with `@PreDestroy` and will be invoked when the context is closed.

For factory beans, the `preDestroy` value in the **Bean** annotation tells Micronaut which method to invoke.

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.Bean;
import io.micronaut.context.annotation.Factory;

import jakarta.inject.Singleton;

@Factory
public class ConnectionFactory {

    @Bean(preDestroy = "stop")  1
    @Singleton
    public Connection connection() {
        return new Connection();
    }
}
```

Copy to Clipboard

Java | Groovy | Kotlin

JAVA

```
import java.util.concurrent.atomic.AtomicBoolean;

public class Connection {

    AtomicBoolean stopped = new AtomicBoolean(false);

    public void stop() {  2
        stopped.compareAndSet(false, true);
    }

}
```

Copy to Clipboard

1 The `preDestroy` value is set on the annotation

2 The annotation value matches the method name

Simply implementing the `Closeable` or `AutoCloseable` interface is not enough for a bean to be closed with the context. One of the above methods must be used.

3.13 Context Events

Micronaut supports a general event system through the context. The **ApplicationEventPublisher** API publishes events and the **ApplicationEventListener** API is used to listen to events. The event system is not limited to events that Micronaut publishes and supports custom events created by users.

Publishing Events

The **ApplicationEventPublisher** API supports events of any type, although all events that Micronaut publishes extend **ApplicationEvent**.

To publish an event, use dependency injection to obtain an instance of **ApplicationEventPublisher** where the generic type is the type of event and invoke the `publishEvent` method with your event object.

Publishing an Event

Java | Groovy | Kotlin

JAVA

```java
public class SampleEvent {
    private String message = "Something happened";

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}

import io.micronaut.context.event.ApplicationEventPublisher;
import jakarta.inject.Inject;
import jakarta.inject.Singleton;

@Singleton
public class SampleEventEmitterBean {

    @Inject
    ApplicationEventPublisher<SampleEvent> eventPublisher;

    public void publishSampleEvent() {
        eventPublisher.publishEvent(new SampleEvent());
    }

}
```

Copy to Clipboard

Publishing an event is **synchronous** by default! The `publishEvent` method will not return until all listeners have been executed. Move this work off to a thread pool if it is time-intensive.

Listening for Events

To listen to an event, register a bean that implements **ApplicationEventListener** where the generic type is the type of event.

Listening for Events with `ApplicationEventListener`

Java	Groovy	Kotlin

JAVA

```java
import io.micronaut.context.event.ApplicationEventListener;
import io.micronaut.docs.context.events.SampleEvent;
import jakarta.inject.Singleton;

@Singleton
public class SampleEventListener implements ApplicationEventListener<SampleEvent> {
    private int invocationCounter = 0;

    @Override
    public void onApplicationEvent(SampleEvent event) {
        invocationCounter++;
    }

    public int getInvocationCounter() {
        return invocationCounter;
    }
}

import io.micronaut.context.ApplicationContext;
import io.micronaut.docs.context.events.SampleEventEmitterBean;
import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class SampleEventListenerSpec {

    @Test
    public void testEventListenerIsNotified() {
        try (ApplicationContext context = ApplicationContext.run()) {
            SampleEventEmitterBean emitter = context.getBean(SampleEventEmitterBean.class);
            SampleEventListener listener = context.getBean(SampleEventListener.class);
            assertEquals(0, listener.getInvocationCounter());
            emitter.publishSampleEvent();
            assertEquals(1, listener.getInvocationCounter());
        }
    }
}
```

Copy to Clipboard

The **supports** method can be overridden to further clarify events to be processed.

Alternatively, use the **@EventListener** annotation if you do not wish to implement an interface or utilize one of the built-in events like **StartupEvent** and **ShutdownEvent**:

Listening for Events with `@EventListener`

Java	Groovy	Kotlin

JAVA

```
import io.micronaut.docs.context.events.SampleEvent;
import io.micronaut.context.event.StartupEvent;
import io.micronaut.context.event.ShutdownEvent;
import io.micronaut.runtime.event.annotation.EventListener;

@Singleton
public class SampleEventListener {
    private int invocationCounter = 0;

    @EventListener
    public void onSampleEvent(SampleEvent event) {
        invocationCounter++;
    }

    @EventListener
    public void onStartupEvent(StartupEvent event) {
        // startup logic here
    }

    @EventListener
    public void onShutdownEvent(ShutdownEvent event) {
        // shutdown logic here
    }

    public int getInvocationCounter() {
        return invocationCounter;
    }
}
```

Copy to Clipboard

If your listener performs work that might take a while, use the **@Async** annotation to run the operation on a separate thread:

Asynchronously listening for Events with `@EventListener`

Java	Groovy	Kotlin

JAVA

```java
import io.micronaut.docs.context.events.SampleEvent;
import io.micronaut.runtime.event.annotation.EventListener;
import io.micronaut.scheduling.annotation.Async;

@Singleton
public class SampleEventListener {
    private AtomicInteger invocationCounter = new AtomicInteger(0);

    @EventListener
    @Async
    public void onSampleEvent(SampleEvent event) {
        invocationCounter.getAndIncrement();
    }

    public int getInvocationCounter() {
        return invocationCounter.get();
    }
}
```

```java
import io.micronaut.context.ApplicationContext;
import io.micronaut.docs.context.events.SampleEventEmitterBean;
import org.junit.Test;

import static java.util.concurrent.TimeUnit.SECONDS;
import static org.awaitility.Awaitility.await;
import static org.hamcrest.Matchers.equalTo;
import static org.junit.Assert.assertEquals;

public class SampleEventListenerSpec {

    @Test
    public void testEventListenerIsNotified() {
        try (ApplicationContext context = ApplicationContext.run()) {
            SampleEventEmitterBean emitter = context.getBean(SampleEventEmitterBean.class);
            SampleEventListener listener = context.getBean(SampleEventListener.class);
            assertEquals(0, listener.getInvocationCounter());
            emitter.publishSampleEvent();
            await().atMost(5, SECONDS).until(listener::getInvocationCounter, equalTo(1));
        }
    }
}
```

Copy to Clipboard

The event listener by default runs on the `scheduled` executor. You can configure this thread pool as required in `application.yml`:

Configuring Scheduled Task Thread Pool

YAML

```yaml
micronaut:
  executors:
    scheduled:
      type: scheduled
      core-pool-size: 30
```

3.14 Bean Events

You can hook into the creation of beans using one of the following interfaces:

- **BeanInitializedEventListener** - allows modifying or replacing a bean after properties have been set but prior to `@PostConstruct` event hooks.
- **BeanCreatedEventListener** - allows modifying or replacing a bean after the bean is fully initialized and all `@PostConstruct` hooks called.

The `BeanInitializedEventListener` interface is commonly used in combination with **Factory** beans. Consider the following example:

Java	Groovy	Kotlin

JAVA

```
public class V8Engine implements Engine {
    private final int cylinders = 8;
    private double rodLength;  1

    public V8Engine(double rodLength) {
        this.rodLength = rodLength;
    }

    @Override
    public String start() {
        return "Starting V" + getCylinders() + " [rodLength=" + getRodLength() + ']';
    }

    @Override
    public final int getCylinders() {
        return cylinders;
    }

    public double getRodLength() {
        return rodLength;
    }

    public void setRodLength(double rodLength) {
        this.rodLength = rodLength;
    }
}

@Factory
public class EngineFactory {

    private V8Engine engine;
    private double rodLength = 5.7;

    @PostConstruct
    public void initialize() {
        engine = new V8Engine(rodLength);  2
    }

    @Singleton
    public Engine v8Engine() {
        return engine; 3
    }

    public void setRodLength(double rodLength) {
        this.rodLength = rodLength;
    }
}

@Singleton
public class EngineInitializer implements BeanInitializedEventListener<EngineFactory> {  4
    @Override
    public EngineFactory onInitialized(BeanInitializingEvent<EngineFactory> event) {
        EngineFactory engineFactory = event.getBean();
        engineFactory.setRodLength(6.6); 5
        return engineFactory;
    }
}
```

Copy to Clipboard

1 The `V8Engine` class defines a `rodLength` property

2 The `EngineFactory` initializes the value of `rodLength` and creates the instance

3 The created instance is returned as a **Bean**

4 The `BeanInitializedEventListener` interface is implemented to listen for the initialization of the factory

5 Within the `onInitialized` method the `rodLength` is overridden prior to the engine being created by the factory bean.

The **BeanCreatedEventListener** interface is more typically used to decorate or enhance a fully initialized bean, for example by creating a proxy.

Bean event listeners are initialized **before** type converters. If your event listener relies on type conversion either by relying on a configuration properties bean or by any other mechanism, you may see errors related to type conversion.

3.15 Bean Introspection

Since Micronaut 1.1, a compile-time replacement for the JDK's **Introspector** (https://docs.oracle.com/javase/8/docs/api/java/beans/Introspector.html) class has been included.

The **BeanIntrospector** and **BeanIntrospection** interfaces allow looking up bean introspections to instantiate and read/write bean properties without using reflection or caching reflective metadata, which consume excessive memory for large beans.

Making a Bean Available for Introspection

Unlike the JDK's **Introspector** (https://docs.oracle.com/javase/8/docs/api/java/beans/Introspector.html), every class is not automatically available for introspection. To make a class available for introspection you must at a minimum enable Micronaut's annotation processor (`micronaut-inject-java` for Java and Kotlin and `micronaut-inject-groovy` for Groovy) in your build and ensure you have a runtime time dependency on `micronaut-core`.

Gradle **Maven**

MAVEN

```
<annotationProcessorPaths>
    <path>
        <groupId>io.micronaut</groupId>
        <artifactId>micronaut-inject-java</artifactId>
        <version>3.1.3</version>
    </path>
</annotationProcessorPaths>
```

Copy to Clipboard

For Kotlin, add the `micronaut-inject-java` dependency in `kapt` scope, and for Groovy add `micronaut-inject-groovy` in `compileOnly` scope.

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut</groupId>
    <artifactId>micronaut-core</artifactId>
    <version>3.1.3</version>
    <scope>runtime</scope>
</dependency>
```

Copy to Clipboard

Once your build is configured you have a few ways to generate introspection data.

Use the `@Introspected` Annotation

The **@Introspected** annotation can be used on any class to make it available for introspection. Simply annotate the class with **@Introspected**:

Java Groovy Kotlin

JAVA

```
import io.micronaut.core.annotation.Introspected;

@Introspected
public class Person {

    private String name;
    private int age = 18;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

Copy to Clipboard

Once introspection data has been produced at compile time, retrieve it via the **BeanIntrospection** API:

Java Groovy Kotlin

JAVA

```
final BeanIntrospection<Person> introspection = BeanIntrospection.getIntrospection(Person.class);  1
Person person = introspection.instantiate("John");  2
System.out.println("Hello " + person.getName());

final BeanProperty<Person, String> property = introspection.getRequiredProperty("name", String.class);  3
property.set(person, "Fred");  4
String name = property.get(person);  5
System.out.println("Hello " + person.getName());
```

Copy to Clipboard

1. You can retrieve a **BeanIntrospection** with the static `getIntrospection` method
2. Once you have a **BeanIntrospection** you can instantiate a bean with the `instantiate` method.
3. A **BeanProperty** can be retrieved from the introspection
4. Use the `set` method to set the property value
5. Use the `get` method to retrieve the property value

Bean Fields

By default Java introspections treat only JavaBean getters/setters or Java 16 record components as bean properties. You can however define classes with public or package protected fields in Java using the `accessKind` member of the **@Introspected** annotation:

Java | Groovy

JAVA

```
import io.micronaut.core.annotation.Introspected;

@Introspected(accessKind = Introspected.AccessKind.FIELD)
public class User {
    public final String name;  1
    public int age = 18;  2

    public User(String name) {
        this.name = name;
    }
}
```

Copy to Clipboard

1. Final fields are treated like read-only properties
2. Mutable fields are treated like read-write properties

The `accessKind` accepts an array so it is possible to allow for both types of accessors but prefer one or the other depending on the order they appear in the annotation. The first one in the list has priority.

Introspections on fields are not possible in Kotlin because it is not possible to declare fields directly.

Constructor Methods

For classes with multiple constructors, apply the **@Creator** annotation to the constructor to use.

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.core.annotation.Creator;
import io.micronaut.core.annotation.Introspected;

import javax.annotation.concurrent.Immutable;

@Introspected
@Immutable
public class Vehicle {

    private final String make;
    private final String model;
    private final int axles;

    public Vehicle(String make, String model) {
        this(make, model, 2);
    }

    @Creator 1
    public Vehicle(String make, String model, int axles) {
        this.make = make;
        this.model = model;
        this.axles = axles;
    }

    public String getMake() {
        return make;
    }

    public String getModel() {
        return model;
    }

    public int getAxles() {
        return axles;
    }
}
```

Copy to Clipboard

1 The **@Creator** annotation denotes which constructor to use

This class has no default constructor, so calls to instantiate without arguments throw an **InstantiationException**.

Static Creator Methods

The **@Creator** annotation can be applied to static methods that create class instances.

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.core.annotation.Creator;
import io.micronaut.core.annotation.Introspected;

import javax.annotation.concurrent.Immutable;

@Introspected
@Immutable
public class Business {

    private final String name;

    private Business(String name) {
        this.name = name;
    }

    @Creator 1
    public static Business forName(String name) {
        return new Business(name);
    }

    public String getName() {
        return name;
    }
}
```

Copy to Clipboard

1 The **@Creator** annotation is applied to the static method which instantiates the class

There can be multiple "creator" methods annotated. If there is one without arguments, it will be the default construction method. The first method with arguments will be used as the primary construction method.

Enums

It is possible to introspect enums as well. Add the annotation to the enum and it can be constructed through the standard `valueOf` method.

Use the `@Introspected` Annotation on a Configuration Class

If the class to introspect is already compiled and not under your control, an alternative option is to define a configuration class with the `classes` member of the **@Introspected** annotation set.

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.core.annotation.Introspected;

@Introspected(classes = Person.class)
public class PersonConfiguration {
}
```

Copy to Clipboard

In the above example the `PersonConfiguration` class generates introspections for the `Person` class.

You can also use the `packages` member of the **@Introspected** which package scans at compile time and generates introspections for all classes within a package. Note however this feature is currently regarded as experimental.

Write an `AnnotationMapper` to Introspect Existing Annotations

If there is an existing annotation that you wish to introspect by default you can write an **AnnotationMapper**.

An example of this is **EntityIntrospectedAnnotationMapper** (https://github.com/micronaut-projects/micronaut-core/blob/master/inject/src/main/java/io/micronaut/inject/beans/visitor/EntityIntrospectedAnnotationMapper.java) which ensures all beans annotated with `javax.persistence.Entity` are introspectable by default.

The `AnnotationMapper` must be on the annotation processor classpath.

The BeanWrapper API

A **BeanProperty** provides raw access to read and write a property value for a given class and does not provide any automatic type conversion.

It is expected that the values you pass to the `set` and `get` methods match the underlying property type, otherwise an exception will occur.

To provide additional type conversion smarts the **BeanWrapper** interface allows wrapping an existing bean instance and setting and getting properties from the bean, plus performing type conversion as necessary.

Java | Groovy | Kotlin

JAVA

```
final BeanWrapper<Person> wrapper = BeanWrapper.getWrapper(new Person("Fred"));  1

wrapper.setProperty("age", "20");  2
int newAge = wrapper.getRequiredProperty("age", int.class);  3

System.out.println("Person's age now " + newAge);
```

Copy to Clipboard

1 Use the static `getWrapper` method to obtain a **BeanWrapper** for a bean instance.

2 You can set properties, and the **BeanWrapper** will perform type conversion, or throw **ConversionErrorException** if conversion is not possible.

3 You can retrieve a property using `getRequiredProperty` and request the appropriate type. If the property doesn't exist a **IntrospectionException** is thrown, and if it cannot be converted a **ConversionErrorException** is thrown.

Jackson and Bean Introspection

Jackson is configured to use the **BeanIntrospection** API to read and write property values and construct objects, resulting in reflection-free serialization/deserialization. This is beneficial from a performance perspective and requires less configuration to operate correctly with runtimes such as GraalVM native.

This feature is enabled by default; disable it by setting the `jackson.bean-introspection-module` configuration to `false`.

Currently only bean properties (private field with public getter/setter) are supported and usage of public fields is not supported.

This feature is currently experimental and may be subject to change in the future.

3.16 Bean Validation

Since Micronaut 1.2, Micronaut has built-in support for validating beans annotated with `javax.validation` annotations. At a minimum include the `micronaut-validation` module as a compile dependency:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut</groupId>
    <artifactId>micronaut-validation</artifactId>
</dependency>
```

Copy to Clipboard

Note that Micronaut's implementation is not currently fully compliant with the **Bean Validator specification** (https://beanvalidation.org/2.0/spec/) as the specification heavily relies on reflection-based APIs.

The following features are unsupported at this time:

- Annotations on generic argument types, since only the Java language supports this feature.
- Any interaction with the **constraint metadata API** (https://beanvalidation.org/2.0/spec/#constraintmetadata), since Micronaut uses compile-time generated metadata.
- XML-based configuration
- Instead of using `javax.validation.ConstraintValidator`, use **ConstraintValidator** (io.micronaut.validation.validator.constraints.ConstraintValidator) to define custom constraints, which supports validating annotations at compile time.

Micronaut's implementation includes the following benefits:

- Reflection and Runtime Proxy free validation, resulting in reduced memory consumption
- Smaller JAR size since Hibernate Validator adds another 1.4MB
- Faster startup since Hibernate Validator adds 200ms+ startup overhead
- Configurability via Annotation Metadata
- Support for Reactive Bean Validation
- Support for validating the source AST at compile time
- Automatic compatibility with GraalVM native without additional configuration

If you require full Bean Validator 2.0 compliance, add the `micronaut-hibernate-validator` module to your build, which replaces Micronaut's implementation.

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.beanvalidation</groupId>
    <artifactId>micronaut-hibernate-validator</artifactId>
</dependency>
```

Copy to Clipboard

Validating Bean Methods

You can validate methods of any class declared as a Micronaut bean by applying `javax.validation` annotations to arguments:

Java Groovy Kotlin

JAVA

```
import jakarta.inject.Singleton;
import javax.validation.constraints.NotBlank;

@Singleton
public class PersonService {
    public void sayHello(@NotBlank String name) {
        System.out.println("Hello " + name);
    }
}
```

Copy to Clipboard

The above example declares that the `@NotBlank` annotation will be validated when invoking the `sayHello` method.

If you use Kotlin, the class and method must be declared `open` so Micronaut can create a compile-time subclass. Alternatively you can annotate the class with **@Validated** and configure the Kotlin `all-open` plugin to open classes annotated with this type. See the **Compiler plugins** (https://kotlinlang.org/docs/reference/compiler-plugins.html) section.

A `javax.validation.ConstraintViolationException` is thrown if a validation error occurs. For example:

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.test.extensions.junit5.annotation.MicronautTest;
import org.junit.jupiter.api.Test;

import jakarta.inject.Inject;
import javax.validation.ConstraintViolationException;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

@MicronautTest
class PersonServiceSpec {

    @Inject PersonService personService;

    @Test
    void testThatNameIsValidated() {
        final ConstraintViolationException exception =
                assertThrows(ConstraintViolationException.class, () ->
                personService.sayHello("")  1
        );

        assertEquals("sayHello.name: must not be blank", exception.getMessage());  2
    }
}
```

Copy to Clipboard

1 The method is called with a blank string

2 An exception occurs

Validating Data Classes

To validate data classes, e.g. POJOs (typically used in JSON interchange), the class must be annotated with **@Introspected** (see the previous section on **Bean Introspection**) or, if the class is external, be imported by the `@Introspected` annotation.

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.core.annotation.Introspected;

import javax.validation.constraints.Min;
import javax.validation.constraints.NotBlank;

@Introspected
public class Person {

    private String name;

    @Min(18)
    private int age;

    @NotBlank
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

Copy to Clipboard

The **@Introspected** annotation can be used as a meta-annotation; common annotations like `@javax.persistence.Entity` are treated as `@Introspected`

The above example defines a `Person` class that has two properties (`name` and `age`) that have constraints applied. Note that in Java the annotations can be on the field or the getter, and with Kotlin data classes, the annotation should target the field.

To validate the class manually, inject an instance of **Validator**:

Java | Groovy | Kotlin

JAVA

```
@Inject
Validator validator;

@Test
void testThatPersonIsValidWithValidator() {
    Person person = new Person();
    person.setName("");
    person.setAge(10);

    final Set<ConstraintViolation<Person>> constraintViolations = validator.validate(person);  1

    assertEquals(2, constraintViolations.size());  2
}
```

Copy to Clipboard

1 The validator validates the person

2 The constraint violations are verified

Alternatively on Bean methods you can use `javax.validation.Valid` to trigger cascading validation:

Java | Groovy | Kotlin

JAVA

```
@Singleton
public class PersonService {
    public void sayHello(@Valid Person person) {
        System.out.println("Hello " + person.getName());
    }
}
```

Copy to Clipboard

The `PersonService` now validates the `Person` class when invoked:

Java | Groovy | Kotlin

JAVA

```
@Inject
PersonService personService;

@Test
void testThatPersonIsValid() {
    Person person = new Person();
    person.setName("");
    person.setAge(10);

    final ConstraintViolationException exception =
        assertThrows(ConstraintViolationException.class, () ->
            personService.sayHello(person)  1
        );

    assertEquals(2, exception.getConstraintViolations().size());  2
}
```

Copy to Clipboard

1 A validated method is invoked

2 The constraint violations are verified

Validating Configuration Properties

You can also validate the properties of classes that are annotated with **@ConfigurationProperties** to ensure configuration is correct.

It is recommended that you annotate **@ConfigurationProperties** that features validation with **@Context** to ensure that the validation occurs at startup.

Defining Additional Constraints

To define additional constraints, create a new annotation, for example:

Java | Groovy | Kotlin

JAVA

```
import javax.validation.Constraint;
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.CONSTRUCTOR;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE_USE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Retention(RUNTIME)
@Constraint(validatedBy = { }) 1
public @interface DurationPattern {

    String message() default "invalid duration ({validatedValue})"; 2

    /**
     * Defines several constraints on the same element.
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        DurationPattern[] value(); 3
    }
}
```

Copy to Clipboard

1 The annotation should be annotated with `javax.validation.Constraint`

2 A `message` template can be provided in a hard-coded manner as above. If none is specified, Micronaut tries to find a message using `ClassName.message` using the **MessageSource** interface (optional)

3 To support repeated annotations you can define an inner annotation (optional)

You can add messages and message bundles using the **MessageSource** and **ResourceBundleMessageSource** classes.

Once you have defined the annotation, implement a **ConstraintValidator** that validates the annotation. You can either create a bean class that implements the interface directly or define a factory that returns one or more validators.

The latter approach is recommended if you plan to define multiple validators:

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.Factory;
import io.micronaut.validation.validator.constraints.ConstraintValidator;

import jakarta.inject.Singleton;

@Factory
public class MyValidatorFactory {

    @Singleton
    ConstraintValidator<DurationPattern, CharSequence> durationPatternValidator() {
        return (value, annotationMetadata, context) -> {
            context.messageTemplate("invalid duration ({validatedValue}), additional custom message"); 1
            return value == null || value.toString().matches("^PT?[\\d]+[SMHD]{1}$");
        };
    }
}
```

Copy to Clipboard

1 Override the default message template with an inline call for more control over the validation error message. (Since `2.5.0`)

The above example implements a validator that validates any field, parameter etc. that is annotated with `DurationPattern`, ensuring that the string can be parsed with `java.time.Duration.parse`.

Generally `null` is regarded as valid and `@NotNull` is used to constrain a value as not being `null`. The example above regards `null` as a valid value.

For example:

Java | Groovy | Kotlin

JAVA

```
@Singleton
public class HolidayService {

    public String startHoliday(@NotBlank String person,
                               @DurationPattern String duration) {
        final Duration d = Duration.parse(duration);
        return "Person " + person + " is off on holiday for " + d.toMinutes() + " minutes";
    }
}
```

Copy to Clipboard

To verify the above examples validates the `duration` parameter, define a test:

Java | Groovy | Kotlin

JAVA

```
@Inject HolidayService holidayService;

@Test
void testCustomValidator() {
    final ConstraintViolationException exception =
        assertThrows(ConstraintViolationException.class, () ->
            holidayService.startHoliday("Fred", "junk")  1
        );

    assertEquals("startHoliday.duration: invalid duration (junk), additional custom message", exception.getMessage())
}
```

Copy to Clipboard

1 A validated method is invoked

2 THe constraint violations are verified

Validating Annotations at Compile Time

You can use Micronaut's validator to validate annotation elements at compile time by including `micronaut-validation` in the annotation processor classpath:

Gradle | **Maven**

MAVEN

```
<annotationProcessorPaths>
    <path>
        <groupId>io.micronaut</groupId>
        <artifactId>micronaut-validation</artifactId>
    </path>
</annotationProcessorPaths>
```

Copy to Clipboard

Then Micronaut will at compile time validate annotation values that are themselves annotated with `javax.validation`. For example consider the following annotation:

Java | Groovy | Kotlin

JAVA

```
import java.lang.annotation.Retention;

import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Retention(RUNTIME)
public @interface TimeOff {
    @DurationPattern
    String duration();
}
```

Copy to Clipboard

If you attempt to use `@TimeOff(duration="junk")` in your source, Micronaut will fail compilation due to the `duration` value violating the `DurationPattern` constraint.

If `duration` is a property placeholder such as `@TimeOff(duration="${my.value}")`, validation is deferred until runtime.

Note that to use a custom `ConstraintValidator` at compile time you must instead define the validator as a class:

Java | Groovy | Kotlin

```java
import io.micronaut.core.annotation.NonNull;
import io.micronaut.core.annotation.Nullable;
import io.micronaut.core.annotation.AnnotationValue;
import io.micronaut.validation.validator.constraints.ConstraintValidator;
import io.micronaut.validation.validator.constraints.ConstraintValidatorContext;

public class DurationPatternValidator implements ConstraintValidator<DurationPattern, CharSequence> {
    @Override
    public boolean isValid(
            @Nullable CharSequence value,
            @NonNull AnnotationValue<DurationPattern> annotationMetadata,
            @NonNull ConstraintValidatorContext context) {
        return value == null || value.toString().matches("^PT?[\\d]+[SMHD]{1}$");
    }
}
```

Additionally:

- Define a `META-INF/services/io.micronaut.validation.validator.constraints.ConstraintValidator` file that references the class.
- The class must be public and have a public no-argument constructor
- The class must be on the annotation processor classpath of the project to be validated.

3.17 Bean Annotation Metadata

The methods provided by Java's **AnnotatedElement** (https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/AnnotatedElement.html) API in general don't provide the ability to introspect annotations without loading the annotations themselves. Nor do they provide any ability to introspect annotation stereotypes (often called meta-annotations; an annotation stereotype is where an annotation is annotated with another annotation, essentially inheriting its behaviour).

To solve this problem many frameworks produce runtime metadata or perform expensive reflection to analyze the annotations of a class.

Micronaut instead produces this annotation metadata at compile time, avoiding expensive reflection and saving memory.

The **BeanContext** API can be used to obtain a reference to a **BeanDefinition** which implements the **AnnotationMetadata** interface.

For example the following code obtains all bean definitions annotated with a particular stereotype:

Lookup Bean Definitions by Stereotype

```java
BeanContext beanContext = ... // obtain the bean context
Collection<BeanDefinition> definitions =
    beanContext.getBeanDefinitions(Qualifiers.byStereotype(Controller.class))

for (BeanDefinition definition : definitions) {
    AnnotationValue<Controller> controllerAnn = definition.getAnnotation(Controller.class);
    // do something with the annotation
}
```

The above example finds all **BeanDefinition** instances annotated with `@Controller` whether `@Controller` is used directly or inherited via an annotation stereotype.

Note that the `getAnnotation` method and the variations of the method return an **AnnotationValue** type and not a Java annotation. This is by design, and you should generally try to work with this API when reading annotation values, since synthesizing a proxy implementation is worse from a performance and memory consumption perspective.

If you require a reference to an annotation instance you can use the `synthesize` method, which creates a runtime proxy that implements the annotation interface:

Synthesizing Annotation Instances

```java
Controller controllerAnn = definition.synthesize(Controller.class);
```

This approach is not recommended however, as it requires reflection and increases memory consumption due to the use of runtime generated proxies, and should be used as a last resort, for example if you need an instance of the annotation to integrate with a third-party library.

Annotation Inheritance

Micronaut will respect the rules defined in Java's **AnnotatedElement** (https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/AnnotatedElement.html) API with regards to annotation inheritance:

- Annotations meta-annotated with **Inherited** (https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Inherited.html) will be available via the `getAnnotation*` methods of the **AnnotationMetadata** API whilst those directly declared are available via the `getDeclaredAnnotation*` methods.
- Annotations not meta-annotated with **Inherited** (https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Inherited.html) will not be included in the metadata

Micronaut differs from the **AnnotatedElement** (https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/AnnotatedElement.html) API in that it extends these rules to methods and method parameters such that:

- Any annotations annotated with **Inherited** (https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Inherited.html) and present on a method of interface or super class `A` that is overridden by child interface or class `B` will be inherited into the **AnnotationMetadata** retrievable via the **ExecutableMethod** API from a **BeanDefinition** or an **AOP interceptor**.
- Any annotations annotated with **Inherited** (https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Inherited.html) and present on a method parameter of interface or super class `A` that is overridden by child interface or class `B` will be inherited into the **AnnotationMetadata** retrievable via the **Argument** interface from the `getArguments` method of the **ExecutableMethod** API.

In general behaviour which you may wish to override is not inherited by default including **Bean Scopes**, **Bean Qualifiers**, **Bean Conditions**, **Validation Rules** and so on.

If you wish a particular scope, qualifier, or set of requirements to be inherited when subclassing then you can define a meta-annotation that is annotated with `@Inherited`. For example:

Defining Inherited Meta Annotations

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.AliasFor;
import io.micronaut.context.annotation.Requires;
import io.micronaut.core.annotation.AnnotationMetadata;
import jakarta.inject.Named;
import jakarta.inject.Singleton;

import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Inherited 1
@Retention(RetentionPolicy.RUNTIME)
@Requires(property = "datasource.url") 2
@Named 3
@Singleton 4
public @interface SqlRepository {
    @AliasFor(annotation = Named.class, member = AnnotationMetadata.VALUE_MEMBER) 5
    String value() default "";
}
```

Copy to Clipboard

1. The annotation is declared as `@Inherited`
2. **Bean Conditions** will be inherited by child classes
3. **Bean Qualifiers** will be inherited by child classes
4. **Bean Scopes** will be inherited by child classes
5. You can also alias annotations and they will be inherited

With this meta-annotation in place you can add the annotation to a super class:

Using Inherited Meta Annotations on a Super Class

Java | Groovy | Kotlin

JAVA

```
@SqlRepository
public abstract class BaseSqlRepository {
}
```

Copy to Clipboard

And then a subclass will inherit all the annotations:

Inherting Annotations in a Child Class

Java | Groovy | Kotlin

JAVA

```
import jakarta.inject.Named;
import javax.sql.DataSource;

@Named("bookRepository")
public class BookRepository extends BaseSqlRepository {
    private final DataSource dataSource;

    public BookRepository(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

Copy to Clipboard

A child class must at least have one bean definition annotation such as a scope or qualifier.

Aliasing / Mapping Annotations

There are times when you may want to alias the value of an annotation member to the value of another annotation member. To do this, use the **@AliasFor** annotation.

A common use case is for example when an annotation defines the `value()` member, but also supports other members. for example the **@Client** annotation:

The @Client Annotation

JAVA

```
public @interface Client {

    /**
     * @return The URL or service ID of the remote service
     */
    @AliasFor(member = "id")  1
    String value() default "";

    /**
     * @return The ID of the client
     */
    @AliasFor(member = "value")  2
    String id() default "";
}
```

1 The `value` member also sets the `id` member

2 The `id` member also sets the `value` member

With these aliases in place, whether you define `@Client("foo")` or `@Client(id="foo")`, both the `value` and `id` members will be set, making it easier to parse and work with the annotation.

If you do not have control over the annotation, another approach is to use an **AnnotationMapper**. To create an `AnnotationMapper`, do the following:

- Implement the **AnnotationMapper** interface
- Define a `META-INF/services/io.micronaut.inject.annotation.AnnotationMapper` file referencing the implementation class
- Add the JAR file containing the implementation to the `annotationProcessor` classpath (`kapt` for Kotlin)

Because `AnnotationMapper` implementations must be on the annotation processor classpath, they should generally be in a project that includes few external dependencies to avoid polluting the annotation processor classpath.

The following is an example `AnnotationMapper` that improves the introspection capabilities of JPA entities.

EntityIntrospectedAnnotationMapper Mapper Example

JAVA

```
public class EntityIntrospectedAnnotationMapper implements NamedAnnotationMapper {
    @NonNull
    @Override
    public String getName() {
        return "javax.persistence.Entity";
    }

    @Override
    public List<AnnotationValue<?>> map(AnnotationValue<Annotation> annotation, VisitorContext visitorContext) {  1
        final AnnotationValueBuilder<Introspected> builder = AnnotationValue.builder(Introspected.class)
                // don't bother with transients properties
                .member("excludedAnnotations", "javax.persistence.Transient");  2
        return Arrays.asList(
                builder.build(),
                AnnotationValue.builder(ReflectiveAccess.class).build()
        );
    }
}
```

1 The `map` method receives a **AnnotationValue** with the values for the annotation.

2 One or more annotations can be returned, in this case `@Transient`.

The example above implements the **NamedAnnotationMapper** interface which allows for annotations to be mixed with runtime code. To operate against a concrete annotation type, use **TypedAnnotationMapper** instead, although note it requires the annotation class itself to be on the annotation processor classpath.

3.18 Importing Beans from Libraries

You can use the **@Import** annotation to import beans from external, already compiled libraries that use JSR-330 annotations.

Bean import is currently only supported in the Java language as other languages have limitations on classpath scanning during source code processing.

For example, to import the JSR-330 TCK into an application, add a dependency on the TCK:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut</groupId>
    <artifactId>jakarta.inject</artifactId>
</dependency>
```

Copy to Clipboard

Then define the `@Import` annotation on your `Application` class:

JAVA

```
package example;

import io.micronaut.context.annotation.Import;

@Import(  1
        packages = {  2
                "org.atinject.tck.auto",
                "org.atinject.tck.auto.accessories"},
        annotated = "*")  3
public class Application {
}
```

1 The **@Import** is defined

2 The `packages` to import are defined. Note that Micronaut will not recurse through sub-packages so sub-packages need to be listed explicitly

3 By default Micronaut will only import classes that feature a scope or a qualifier. By using `*` you can make every type a bean.

In general `@Import` should be used in applications rather than libraries since if two libraries import the same beans the result will likely be a **NonUniqueBeanException**

3.19 Micronaut Beans And Spring

Micronaut has integrations with Spring in several forms. See the **Micronaut Spring Documentation** (https://micronaut-projects.github.io/micronaut-spring/latest/guide) for more information.

3.20 Android Support

Since Micronaut dependency injection is based on annotation processors and doesn't rely on reflection, it can be used on Android when using the Android plugin 3.0.0 or higher.

This lets you use the same application framework for both your Android client and server implementation.

Configuring Your Android Build

To get started, add the Micronaut annotation processors to the processor classpath using the `annotationProcessor` dependency configuration.

Include the Micronaut `micronaut-inject-java` dependency in both the `annotationProcessor` and `compileOnly` scopes of your Android build configuration:

Example Android build.gradle

GROOVY

```
dependencies {
    ...
    annotationProcessor "io.micronaut:micronaut-inject-java:3.1.3"
    compileOnly "io.micronaut:micronaut-inject-java:3.1.3"
    ...
}
```

If you use `lint` as part of your build you may also need to disable the invalid packages check since Android includes a hard-coded check that regards the `javax.inject` package as invalid unless you use Dagger:

Configure lint within build.gradle

GROOVY

```
android {
    ...
    lintOptions {
        lintOptions { warning 'InvalidPackage' }
    }
}
```

You can find more information on **configuring annotations processors** (https://developer.android.com/studio/build/gradle-plugin-3-0-0-migration.html#annotationProcessor_config) in the Android documentation.

Micronaut `inject-java` dependency uses **Android Java 8 support** (https://developer.android.com/studio/write/java8-support.html) features.

Enabling Dependency Injection

Once you have configured the classpath correctly, the next step is start the **ApplicationContext**.

The following example demonstrates creating a subclass of **android.app.Application** (https://developer.android.com/reference/android/app/Application.html) for that purpose:

Example Android Application Class

JAVA

```
import android.app.Activity;
import android.app.Application;
import android.os.Bundle;

import io.micronaut.context.ApplicationContext;
import io.micronaut.context.env.Environment;

public class BaseApplication extends Application {  1

    private ApplicationContext ctx;

    @Override
    public void onCreate() {
        super.onCreate();
        ctx = ApplicationContext.run(MainActivity.class, Environment.ANDROID);  2
        registerActivityLifecycleCallbacks(new ActivityLifecycleCallbacks() {  3
            @Override
            public void onActivityCreated(Activity activity, Bundle bundle) {
                ctx.inject(activity);
            }
            ... // shortened for brevity; it is not necessary to implement other methods
        });
    }
}
```

1. Extend the `android.app.Application` class
2. Run the `ApplicationContext` with the `ANDROID` environment
3. Register an `ActivityLifecycleCallbacks` instance to allow dependency injection of Android `Activity` instances

4 Application Configuration

Configuration in Micronaut takes inspiration from both Spring Boot and Grails, integrating configuration properties from multiple sources directly into the core IoC container.

Configuration can by default be provided in Java properties, YAML, JSON, or Groovy files. The convention is to search for a file named `application.yml`, `application.properties`, `application.json` or `application.groovy`.

In addition, like Spring and Grails, Micronaut allows overriding any property via system properties or environment variables.

Each source of configuration is modeled with the **PropertySource** interface and the mechanism is extensible, allowing the implementation of additional **PropertySourceLoader** implementations.

4.1 The Environment

The application environment is modelled by the **Environment** interface, which allows specifying one or many unique environment names when creating an **ApplicationContext**.

Initializing the Environment

Java | Groovy | Kotlin

JAVA

```
ApplicationContext applicationContext = ApplicationContext.run("test", "android");
Environment environment = applicationContext.getEnvironment();

assertTrue(environment.getActiveNames().contains("test"));
assertTrue(environment.getActiveNames().contains("android"));
```

Copy to Clipboard

The active environment names allow loading different configuration files depending on the environment, and also using the **@Requires** annotation to conditionally load beans or bean **@Configuration** packages.

In addition, Micronaut attempts to detect the current environments. For example within a Spock or JUnit test the **TEST** environment is automatically active.

Additional active environments can be specified using the `micronaut.environments` system property or the `MICRONAUT_ENVIRONMENTS` environment variable. These are specified as a comma-separated list. For example:

Specifying environments

BASH

```
$ java -Dmicronaut.environments=foo,bar -jar myapp.jar
```

The above activates environments called `foo` and `bar`.

Finally, Cloud environment names are also detected. See the section on **Cloud Configuration** for more information.

Environment Priority

Micronaut loads property sources based on the environments specified, and if the same property key exists in multiple property sources specific to an environment, the environment order determines which value to use.

Micronaut uses the following hierarchy for environment processing (lowest to highest priority):

- Deduced environments
- Environments from the `micronaut.environments` system property
- Environments from the `MICRONAUT_ENVIRONMENTS` environment variable
- Environments specified explicitly through the application context builder

This also applies to `@MicronautTest(environments = …)`

Disabling Environment Detection

Automatic detection of environments can be disabled by setting the `micronaut.env.deduction` system property or the `MICRONAUT_ENV_DEDUCTION` environment variable to `false`. This prevents Micronaut from detecting current environments, while still using any environments that are specifically provided as shown above.

Disabling environment detection via system property

BASH

```
$  java -Dmicronaut.env.deduction=false -jar myapp.jar
```

Alternatively, you can disable environment deduction using the **ApplicationContextBuilder** `deduceEnvironment` method when setting up your application.

Using ApplicationContextBuilder to disable environment deduction

Java | Groovy | Kotlin

JAVA

```
@Test
public void testDisableEnvironmentDeductionViaBuilder() {
    ApplicationContext ctx = ApplicationContext.builder()
            .deduceEnvironment(false)
            .properties(Collections.singletonMap("micronaut.server.port", -1))
            .start();
    assertFalse(ctx.getEnvironment().getActiveNames().contains(Environment.TEST));
    ctx.close();
}
```

Copy to Clipboard

Default Environment

Micronaut supports the concept of one or many default environments. A default environment is one that is only applied if no other environments are explicitly specified or deduced. Environments can be explicitly specified either through the application context builder `Micronaut.build().environments(…)`, through the `micronaut.environments` system property, or the `MICRONAUT_ENVIRONMENTS` environment variable. Environments can be deduced to automatically apply the environment appropriate for cloud deployments. If an environment is found through any of the above means, the default environment will **not** be applied.

To set the default environments, modify your application main method:

JAVA

```
public static void main(String[] args) {
    Micronaut.build(args)
             .mainClass(Application.class)
             .defaultEnvironments("dev")
             .start();
}
```

Micronaut Banner

Since Micronaut 2.3 a banner is shown when the application starts. It is enabled by default and it also shows the Micronaut version.

SHELL

```
$ ./gradlew run
  __  __ _                            _
 |  \/  (_) ___ _ __ ___  _ __   __ _ _   _| |_
 | |\/| | |/ __| '__/ _ \| '_ \ / _` | | | | __|
 | |  | | | (__| | | (_) | | | | (_| | |_| | |_
 |_|  |_|_|\___|_|  \___/|_| |_|\__,_|\__,_|\__|
  Micronaut (3.1.3)

17:07:22.997 [main] INFO  io.micronaut.runtime.Micronaut - Startup completed in 611ms. Server Running: http://localhost
```

To customize the banner with your own ASCII Art (just plain ASCII at this moment), create the file `src/main/resources/micronaut-banner.txt` and it will be used instead.

To disable it, modify your `Application` class:

JAVA

```
public class Application {

    public static void main(String[] args) {
        Micronaut.build(args)
                 .banner(false)  1
                 .start();
    }
}
```

1 Disable the banner

4.2 Externalized Configuration with PropertySources

Additional **PropertySource** instances can be added to the environment prior to initializing the **ApplicationContext**.

Initializing the Environment

Java | Groovy | Kotlin

JAVA

```
ApplicationContext applicationContext = ApplicationContext.run(
        PropertySource.of(
                "test",
                CollectionUtils.mapOf(
                    "micronaut.server.host", "foo",
                    "micronaut.server.port", 8080
                )
        ),
        "test", "android");
Environment environment = applicationContext.getEnvironment();

assertEquals(
        "foo",
        environment.getProperty("micronaut.server.host", String.class).orElse("localhost")
);
```

Copy to Clipboard

The **PropertySource.of** method can be used to create a `PropertySource` from a map of values.

Alternatively one can register a **PropertySourceLoader** by creating a `META-INF/services/io.micronaut.context.env.PropertySourceLoader` file containing a reference to the class name of the `PropertySourceLoader`.

Included PropertySource Loaders

Micronaut by default contains `PropertySourceLoader` implementations that load properties from the given locations and priority:

1. Command line arguments
2. Properties from `SPRING_APPLICATION_JSON` (for Spring compatibility)
3. Properties from `MICRONAUT_APPLICATION_JSON`
4. Java System Properties
5. OS environment variables
6. Configuration files loaded in order from the system property 'micronaut.config.files' or the environment variable `MICRONAUT_CONFIG_FILES`. The value can be a comma-separated list of paths with the last file having precedence. The files can be referenced from the file system as a path, or the classpath with a `classpath:` prefix.
7. Environment-specific properties from `application-{environment}.{extension}`
8. Application-specific properties from `application.{extension}`

`.properties`, `.json`, `.yml` are supported out of the box. For Groovy users `.groovy` is supported as well.

Supplying Configuration via Command Line

Configuration can be supplied at the command line using Gradle or our Maven plugin. For example:

Gradle

```bash
$ ./gradlew run --args="-endpoints.health.enabled=true -config.property=test"
```

Maven

```bash
$ ./mvnw mn:run -Dmn.appArgs="-endpoints.health.enabled=true -config.property=test"
```

For the configuration to be a part of the context, the args from the main method must be passed to the context builder. For example:

```java
import io.micronaut.runtime.Micronaut;

public class Application {

    public static void main(String[] args) {
        Micronaut.run(Application.class, args); // passing args
    }
}
```

Property Value Placeholders

Micronaut includes a property placeholder syntax to reference configuration properties both within configuration values and with any Micronaut annotation. See **@Value** and the section on **Configuration Injection**.

Programmatic usage is also possible via the **PropertyPlaceholderResolver** interface.

The basic syntax is to wrap a reference to a property in `${…}`. For example in `application.yml`:

Defining Property Placeholders

```yaml
myapp:
  endpoint: http://${micronaut.server.host}:${micronaut.server.port}/foo
```

The above example embeds references to the `micronaut.server.host` and `micronaut.server.port` properties.

You can specify default values by defining a value after the `:` character. For example:

Using Default Values

```yaml
myapp:
  endpoint: http://${micronaut.server.host:localhost}:${micronaut.server.port:8080}/foo
```

The above example defaults to `localhost` and port `8080` if no value is found (rather than throwing an exception). Note that if the default value contains a `:` character, you must escape it using backticks:

Using Backticks

YAML

```
myapp:
  endpoint: ${server.address:`http://localhost:8080`}/foo
```

The above example looks for a `server.address` property and defaults to `http://localhost:8080`. This default value is escaped with backticks since it has a `:` character.

Property Value Binding

Note that these property references should be in kebab case (lowercase and hyphen-separated) when placing references in code or in placeholder values. For example, use `micronaut.server.default-charset` and not `micronaut.server.defaultCharset`.

Micronaut still allows specifying the latter in configuration, but normalizes the properties into kebab case form to optimize memory consumption and reduce complexity when resolving properties. The following table summarizes how properties are normalized from different sources:

Table 1. Property Value Normalization

Configuration Value	Resulting Properties	Property Source
`myApp.myStuff`	`my-app.my-stuff`	Properties, YAML etc.
`my-app.myStuff`	`my-app.my-stuff`	Properties, YAML etc.
`myApp.my-stuff`	`my-app.my-stuff`	Properties, YAML etc.
`MYAPP_MYSTUFF`	`myapp.mystuff`, `myapp-mystuff`	Environment Variable
`MY_APP_MY_STUFF`	`my.app.my.stuff`, `my.app.my-stuff`, `my.app-my.stuff`, `my.app-my-stuff`, `my-app.my.stuff`, `my-app.my-stuff`, `my-app-my.stuff`, `my-app-my-stuff`	Environment Variable

Environment variables are treated specially to allow more flexibility. Note that there is no way to reference an environment variable with camel-case.

Because the number of properties generated is exponential based on the number of `_` characters in an environment variable, it is recommended to refine which, if any, environment variables are included in configuration if the number of environment variables with multiple underscores is high.

To control how environment properties participate in configuration, call the respective methods on the `Micronaut` builder.

Application class

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.runtime.Micronaut;

public class Application {

    public static void main(String[] args) {
        Micronaut.build(args)
                .mainClass(Application.class)
                .environmentPropertySource(false)
                //or
                .environmentVariableIncludes("THIS_ENV_ONLY")
                //or
                .environmentVariableExcludes("EXCLUDED_ENV")
                .start();
    }
}
```

Copy to Clipboard

The configuration above does not have any impact on property placeholders. It is still possible to reference an environment variable in a placeholder regardless of whether environment configuration is disabled, or even if the specific property is explicitly excluded.

Using Random Properties

You can use `random` values by using the following properties. These can be used in configuration files as variables like the following.

YAML

```
micronaut:
  application:
    name: myapplication
    instance:
      id: ${random.shortuuid}
```

Table 2. Random Values

Property	Value
random.port	An available random port number
random.int	Random int
random.integer	Random int
random.long	Random long
random.float	Random float
random.shortuuid	Random UUID of only 10 chars in length (Note: As this isn't full UUID, collision COULD occur)
random.uuid	Random UUID with dashes
random.uuid2	Random UUID without dashes

The `random.int`, `random.integer`, `random.long` and `random.float` properties supports a range suffix whose syntax is one of as follows:

- `(max)` where max is an exclusive value
- `[min,max]` where min being inclusive and max being exclusive values.

YAML

```
instance:
  id: ${random.int[5,10]}
  count: ${random.int(5)}
```

The range could vary from negative to positive as well.

Fail Fast Property Injection

For beans that inject required properties, the injection and potential failure will not occur until the bean is requested. To verify at startup that the properties exist and can be injected, the bean can be annotated with **@Context**. Context-scoped beans are injected at startup, and startup fails if any required properties are missing or cannot be converted to the required type.

It is recommended to use this feature sparingly to ensure fast startup.

4.3 Configuration Injection

You can inject configuration values into beans using the **@Value** annotation.

Using the `@Value` Annotation

Consider the following example:

@Value Example

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.Value;

import jakarta.inject.Singleton;

@Singleton
public class EngineImpl implements Engine {

    @Value("${my.engine.cylinders:6}") 1
    protected int cylinders;

    @Override
    public int getCylinders() {
        return cylinders;
    }

    @Override
    public String start() { 2
        return "Starting V" + getCylinders() + " Engine";
    }

}
```

Copy to Clipboard

1 The `@Value` annotation accepts a string that can have embedded placeholder values (the default value can be provided by specifying a value after the colon `:` character).

2 The injected value can then be used within code.

Note that `@Value` can also be used to inject a static value. For example the following injects the number 10:

Static @Value Example

GROOVY

```
@Value("10")
int number;
```

This is even more useful when used to compose injected values combining static content and placeholders. For example to set up a URL:

Placeholders with @Value

GROOVY

```
@Value("http://${my.host}:${my.port}")
URL url;
```

In the above example the URL is constructed from two placeholder properties that must be present in configuration: `my.host` and `my.port`.

Remember that to specify a default value in a placeholder expression, you use the colon `:` character. However, if the default you specify includes a colon, you must escape the value with backticks. For example:

Placeholders with @Value

GROOVY

```
@Value("${my.url:`http://foo.com`}")
URL url;
```

Note that there is nothing special about `@Value` itself regarding the resolution of property value placeholders.

Due to Micronaut's extensive support for annotation metadata you can use property placeholder expressions on any annotation. For example, to make the path of a `@Controller` configurable you can do:

JAVA

```
@Controller("${hello.controller.path:/hello}")
class HelloController {
    ...
}
```

In the above case, if `hello.controller.path` is specified in configuration the controller will be mapped to the specified path, otherwise it will be mapped to `/hello`.

You can also make the target server for **@Client** configurable (although service discovery approaches are often better), for example:

JAVA

```
@Client("${my.server.url:`http://localhost:8080`}")
interface HelloClient {
    ...
}
```

In the above example the property `my.server.url` can be used to configure the client, otherwise the client falls back to a localhost address.

Using the `@Property` Annotation

Recall that the **@Value** annotation receives a String value which can be a mix of static content and placeholder expressions. This can lead to confusion if you attempt to do the following:

Incorrect usage of `@Value`

GROOVY

```
@Value("my.url")
String url;
```

In the above case the literal string value `my.url` is injected and set to the `url` field and **not** the value of the `my.url` property from your application configuration. This is because `@Value` only resolves placeholders within the value specified to it.

To inject a specific property name, you may be better off using **@Property**:

Using @Property

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.Property;

import jakarta.inject.Inject;
import jakarta.inject.Singleton;

@Singleton
public class Engine {

    @Property(name = "my.engine.cylinders")  1
    protected int cylinders;  2

    private String manufacturer;

    public int getCylinders() {
        return cylinders;
    }

    public String getManufacturer() {
        return manufacturer;
    }

    @Inject
    public void setManufacturer(@Property(name = "my.engine.manufacturer") String manufacturer) {  3
        this.manufacturer = manufacturer;
    }

}
```

Copy to Clipboard

1 The `my.engine.cylinders` property is resolved from configuration and injected into the field.

2 Fields subject to injection should not be private because expensive reflection must be used

3 The `@Property` annotation is used to inject through the setter

Because it is not possible to define a default value with `@Property`, if the value doesn't exist or cannot be converted to the required type, bean instantiation will fail.

The above instead injects the value of the `my.url` property resolved from application configuration. If the property cannot be found in configuration, an exception is thrown. As with other types of injection, the injection point can also be annotated with `@Nullable` to make the injection optional.

You can also use this feature to resolve sub maps. For example, consider the following configuration:

Example `application.yml` *configuration*

YAML

```
datasources:
  default:
    name: 'mydb'
jpa:
  default:
    properties:
      hibernate:
        hbm2ddl:
          auto: update
        show_sql: true
```

To resolve a flattened map containing only the properties starting with `hibernate`, use `@Property`, for example:

Using `@Property`

JAVA

```
@Property(name = "jpa.default.properties")
Map<String, String> jpaProperties;
```

The injected map will contain the keys `hibernate.hbm2ddl.auto` and `hibernate.show_sql` and their values.

The **@MapFormat** annotation can be used to customize the injected map depending on whether you want nested keys or flat keys, and it allows customization of the key style via the **StringConvention** enum.

4.4 Configuration Properties

You can create type-safe configuration by creating classes that are annotated with **@ConfigurationProperties**.

Micronaut will produce a reflection-free `@ConfigurationProperties` bean and will also at compile time calculate the property paths to evaluate, greatly improving the speed and efficiency of loading `@ConfigurationProperties`.

For example:

@ConfigurationProperties Example

Java	Groovy	Kotlin

JAVA

```java
import io.micronaut.context.annotation.ConfigurationProperties;

import javax.validation.constraints.Min;
import javax.validation.constraints.NotBlank;
import java.util.Optional;

@ConfigurationProperties("my.engine") 1
public class EngineConfig {

    public String getManufacturer() {
        return manufacturer;
    }

    public void setManufacturer(String manufacturer) {
        this.manufacturer = manufacturer;
    }

    public int getCylinders() {
        return cylinders;
    }

    public void setCylinders(int cylinders) {
        this.cylinders = cylinders;
    }

    public CrankShaft getCrankShaft() {
        return crankShaft;
    }

    public void setCrankShaft(CrankShaft crankShaft) {
        this.crankShaft = crankShaft;
    }

    @NotBlank 2
    private String manufacturer = "Ford"; 3

    @Min(1L)
    private int cylinders;

    private CrankShaft crankShaft = new CrankShaft();

    @ConfigurationProperties("crank-shaft")
    public static class CrankShaft { 4

        private Optional<Double> rodLength = Optional.empty(); 5

        public Optional<Double> getRodLength() {
            return rodLength;
        }

        public void setRodLength(Optional<Double> rodLength) {
            this.rodLength = rodLength;
        }
    }
}
```

Copy to Clipboard

1 The `@ConfigurationProperties` annotation takes the configuration prefix

2 You can use `javax.validation` annotations to validate the configuration

3 Default values can be assigned to the property

4 Static inner classes can provide nested configuration

5 Optional configuration values can be wrapped in `java.util.Optional`

Once you have prepared a type-safe configuration it can be injected into your beans like any other bean:

@ConfigurationProperties Dependency Injection

Java	Groovy	Kotlin

JAVA

```
@Singleton
public class EngineImpl implements Engine {
    private final EngineConfig config;

    public EngineImpl(EngineConfig config) {  1
        this.config = config;
    }

    @Override
    public int getCylinders() {
        return config.getCylinders();
    }

    @Override
    public String start() {  2
        return getConfig().getManufacturer() + " Engine Starting V" + getConfig().getCylinders() +
                " [rodLength=" + getConfig().getCrankShaft().getRodLength().orElse(6d) + "]";
    }

    public final EngineConfig getConfig() {
        return config;
    }
}
```

Copy to Clipboard

1 Inject the `EngineConfig` bean

2 Use the configuration properties

Configuration values can then be supplied from one of the **PropertySource** instances. For example:

Supply Configuration

Java | Groovy | Kotlin

JAVA

```
Map<String, Object> map = new LinkedHashMap<>(1);
map.put("my.engine.cylinders", "8");
ApplicationContext applicationContext = ApplicationContext.run(map, "test");

Vehicle vehicle = applicationContext.getBean(Vehicle.class);
System.out.println(vehicle.start());
```

Copy to Clipboard

The above example prints: `"Ford Engine Starting V8 [rodLength=6.0]"`

Note for more complex configurations you can structure **@ConfigurationProperties** beans through inheritance.

For example creating a subclass of `EngineConfig` with `@ConfigurationProperties('bar')` will resolve all properties under the path `my.engine.bar`.

Includes / Excludes

For the cases where the configuration properties class inherits properties from a parent class, it may be desirable to exclude properties from the parent class. The `includes` and `excludes` members of the **@ConfigurationProperties** annotation allow for that functionality. The list applies to both local properties and inherited properties.

The names supplied to the includes/excludes list must be the "property" name. For example if a setter method is injected, the property name is the de-capitalized setter name (`setConnectionTimeout` → `connectionTimeout`).

Property Type Conversion

Micronaut uses the **ConversionService** bean to convert values when resolving properties. You can register additional converters for types not supported by Micronaut by defining beans that implement the **TypeConverter** interface.

Micronaut features some built-in conversions that are useful, which are detailed below.

Duration Conversion

Durations can be specified by appending the unit with a number. Supported units are `s`, `ms`, `m` etc. The following table summarizes examples:

Table 1. Duration Conversion

Configuration Value	Resulting Value
`10ms`	`Duration` of 10 milliseconds
`10m`	`Duration` of 10 minutes
`10s`	`Duration` of 10 seconds

Configuration Value	Resulting Value
`10d`	`Duration` of 10 days
`10h`	`Duration` of 10 hours
`10ns`	`Duration` of 10 nanoseconds
`PT15M`	`Duration` of 15 minutes using ISO-8601 format

For example to configure the default HTTP client read timeout:

Using Duration Values

YAML

```yaml
micronaut:
  http:
    client:
      read-timeout: 15s
```

List / Array Conversion

Lists and arrays can be specified in Java properties files as comma-separated values, or in YAML using native YAML lists. The generic types are used to convert the values. For example in YAML:

Specifying lists or arrays in YAML

YAML

```yaml
my:
  app:
    integers:
      - 1
      - 2
    urls:
      - http://foo.com
      - http://bar.com
```

Or in Java properties file format:

Specifying lists or arrays in Java properties comma-separated

PROPERTIES

```
my.app.integers=1,2
my.app.urls=http://foo.com,http://bar.com
```

Alternatively you can use an index:

Specifying lists or arrays in Java properties using index

PROPERTIES

```
my.app.integers[0]=1
my.app.integers[1]=2
```

For the above example configurations you can define properties to bind to with the target type supplied via generics:

JAVA

```
List<Integer> integers;
List<URL> urls;
```

Readable Bytes

You can annotate any setter parameter with **@ReadableBytes** to allow the value to be set using a shorthand syntax for specifying bytes, kilobytes etc. For example the following is taken from **HttpClientConfiguration**:

Using `@ReadableBytes`

JAVA

```
public void setMaxContentLength(@ReadableBytes int maxContentLength) {
    this.maxContentLength = maxContentLength;
}
```

With the above in place you can set `micronaut.http.client.max-content-length` using the following values:

Table 2. @ReadableBytes Conversion

Configuration Value	Resulting Value
`10mb`	10 megabytes

Configuration Value	Resulting Value
`10kb`	10 kilobytes
`10gb`	10 gigabytes
`1024`	A raw byte length

Formatting Dates

The **@Format** annotation can be used on setters to specify the date format to use when binding `java.time` date objects.

Using `@Format` *for Dates*

JAVA

```
public void setMyDate(@Format("yyyy-MM-dd") LocalDate date) {
    this.myDate = date;
}
```

Configuration Builder

Many frameworks and tools already use builder-style classes to construct configuration.

You can use the **@ConfigurationBuilder** annotation to populate a builder-style class with configuration values. **ConfigurationBuilder** can be applied to fields or methods in a class annotated with **@ConfigurationProperties**.

Since there is no consistent way to define builders in the Java world, one or more method prefixes can be specified in the annotation to support builder methods like `withXxx` or `setXxx`. If the builder methods have no prefix, assign an empty string to the parameter.

A configuration prefix can also be specified to tell Micronaut where to look for configuration values. By default, builder methods use the configuration prefix specified in a class-level **@ConfigurationProperties** annotation.

For example:

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.ConfigurationBuilder;
import io.micronaut.context.annotation.ConfigurationProperties;

@ConfigurationProperties("my.engine") 1
class EngineConfig {

    @ConfigurationBuilder(prefixes = "with") 2
    EngineImpl.Builder builder = EngineImpl.builder();

    @ConfigurationBuilder(prefixes = "with", configurationPrefix = "crank-shaft") 3
    CrankShaft.Builder crankShaft = CrankShaft.builder();

    private SparkPlug.Builder sparkPlug = SparkPlug.builder();

    SparkPlug.Builder getSparkPlug() {
        return sparkPlug;
    }

    @ConfigurationBuilder(prefixes = "with", configurationPrefix = "spark-plug") 4
    void setSparkPlug(SparkPlug.Builder sparkPlug) {
        this.sparkPlug = sparkPlug;
    }
}
```

Copy to Clipboard

1. The `@ConfigurationProperties` annotation takes the configuration prefix
2. The first builder can be configured without the class configuration prefix; it inherits from the above.
3. The second builder can be configured with the class configuration prefix + the `configurationPrefix` value.
4. The third builder demonstrates that the annotation can be applied to a method as well as a property.

By default, only single-argument builder methods are supported. For methods with no arguments, set the `allowZeroArgs` parameter of the annotation to `true`.

Like in the previous example, we can construct an `EngineImpl`. Since we are using a builder, we can use a factory class to build the engine from the builder.

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.Factory;

import jakarta.inject.Singleton;

@Factory
class EngineFactory {

    @Singleton
    EngineImpl buildEngine(EngineConfig engineConfig) {
        return engineConfig.builder.build(engineConfig.crankShaft, engineConfig.getSparkPlug());
    }
}
```

Copy to Clipboard

The engine that was returned can then be injected anywhere an engine is required.

Configuration values can be supplied from one of the **PropertySource** instances. For example:

Java | Groovy | Kotlin

JAVA

```
Map<String, Object> properties = new HashMap<>();
properties.put("my.engine.cylinders"             ,"4");
properties.put("my.engine.manufacturer"          , "Subaru");
properties.put("my.engine.crank-shaft.rod-length", 4);
properties.put("my.engine.spark-plug.name"       , "6619 LFR6AIX");
properties.put("my.engine.spark-plug.type"       , "Iridium");
properties.put("my.engine.spark-plug.companyName", "NGK");
ApplicationContext applicationContext = ApplicationContext.run(properties, "test");

Vehicle vehicle = applicationContext.getBean(Vehicle.class);
System.out.println(vehicle.start());
```

Copy to Clipboard

The above example prints: `"Subaru Engine Starting V4 [rodLength=4.0, sparkPlug=Iridium(NGK 6619 LFR6AIX)]"`

MapFormat

For some use cases it may be desirable to accept a map of arbitrary configuration properties that can be supplied to a bean, especially if the bean represents a third-party API where not all the possible configuration properties are known. For example, a datasource may accept a map of configuration properties specific to a particular database driver, allowing the user to specify any desired options in the map without coding each property explicitly.

For this purpose, the **MapFormat** annotation lets you bind a map to a single configuration property, and specify whether to accept a flat map of keys to values, or a nested map (where the values may be additional maps).

@MapFormat Example

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.ConfigurationProperties;
import io.micronaut.core.convert.format.MapFormat;

import javax.validation.constraints.Min;
import java.util.Map;

@ConfigurationProperties("my.engine")
public class EngineConfig {

    @Min(1L)
    private int cylinders;

    @MapFormat(transformation = MapFormat.MapTransformation.FLAT)  1
    private Map<Integer, String> sensors;

    public int getCylinders() {
        return cylinders;
    }

    public void setCylinders(int cylinders) {
        this.cylinders = cylinders;
    }

    public Map<Integer, String> getSensors() {
        return sensors;
    }

    public void setSensors(Map<Integer, String> sensors) {
        this.sensors = sensors;
    }
}
```

Copy to Clipboard

1 Note the `transformation` argument to the annotation; possible values are `MapTransformation.FLAT` (for flat maps) and `MapTransformation.NESTED` (for nested maps)

EngineImpl

Java | Groovy | Kotlin

JAVA

```
@Singleton
public class EngineImpl implements Engine {

    @Inject
    EngineConfig config;

    @Override
    public Map getSensors() {
        return config.getSensors();
    }

    @Override
    public String start() {
        return "Engine Starting V" + getConfig().getCylinders() +
               " [sensors=" + getSensors().size() + "]";
    }

    public EngineConfig getConfig() {
        return config;
    }

    public void setConfig(EngineConfig config) {
        this.config = config;
    }
}
```

Copy to Clipboard

Now a map of properties can be supplied to the `my.engine.sensors` configuration property.

Use Map Configuration

Java | Groovy | Kotlin

JAVA

```
Map<String, Object> map = new LinkedHashMap<>(2);
map.put("my.engine.cylinders", "8");

Map<Integer, String> map1 = new LinkedHashMap<>(2);
map1.put(0, "thermostat");
map1.put(1, "fuel pressure");

map.put("my.engine.sensors", map1);

ApplicationContext applicationContext = ApplicationContext.run(map, "test");

Vehicle vehicle = applicationContext.getBean(Vehicle.class);
System.out.println(vehicle.start());
```

Copy to Clipboard

The above example prints: `"Engine Starting V8 [sensors=2]"`

4.5 Custom Type Converters

Micronaut includes an extensible type conversion mechanism. To add additional type converters you register beans of type [TypeConverter](#).

The following example shows how to use one of the built-in converters (Map to an Object) or create your own.

Consider the following [ConfigurationProperties](#):

Java | Groovy | Kotlin

JAVA

```
@ConfigurationProperties(MyConfigurationProperties.PREFIX)
public class MyConfigurationProperties {

    public static final String PREFIX = "myapp";

    protected LocalDate updatedAt;

    public LocalDate getUpdatedAt() {
        return updatedAt;
    }
}
```

Copy to Clipboard

The type `MyConfigurationProperties` has a property named `updatedAt` of type [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html).

To bind this property from a map via configuration:

Java | Groovy | Kotlin

JAVA

```
private static ApplicationContext ctx;

@BeforeClass
public static void setupCtx() {
    ctx = ApplicationContext.run(
            new LinkedHashMap<String, Object>() {{
                put("myapp.updatedAt",  1
                        new LinkedHashMap<String, Integer>() {{
                            put("day", 28);
                            put("month", 10);
                            put("year", 1982);
                        }}
                );
            }}
    );
}

@AfterClass
public static void teardownCtx() {
    if(ctx != null) {
        ctx.stop();
    }
}
```

Copy to Clipboard

1 Note how we match the `myapp` prefix and `updatedAt` property name in our `MyConfigurationProperties` class above

This won't work by default, since there is no built-in conversion from `Map` to `LocalDate`. To resolve this, define a custom [TypeConverter](#):

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.core.convert.ConversionContext;
import io.micronaut.core.convert.ConversionService;
import io.micronaut.core.convert.TypeConverter;

import jakarta.inject.Singleton;
import java.time.DateTimeException;
import java.time.LocalDate;
import java.util.Map;
import java.util.Optional;

@Singleton
public class MapToLocalDateConverter implements TypeConverter<Map, LocalDate> {  1
    @Override
    public Optional<LocalDate> convert(Map propertyMap, Class<LocalDate> targetType, ConversionContext context) {
        Optional<Integer> day = ConversionService.SHARED.convert(propertyMap.get("day"), Integer.class);
        Optional<Integer> month = ConversionService.SHARED.convert(propertyMap.get("month"), Integer.class);
        Optional<Integer> year = ConversionService.SHARED.convert(propertyMap.get("year"), Integer.class);
        if (day.isPresent() && month.isPresent() && year.isPresent()) {
            try {
                return Optional.of(LocalDate.of(year.get(), month.get(), day.get()));  2
            } catch (DateTimeException e) {
                context.reject(propertyMap, e);  3
                return Optional.empty();
            }
        }

        return Optional.empty();
    }
}
```

Copy to Clipboard

1 The class implements **TypeConverter** which has two generic arguments, the type you are converting from, and the type you are converting to

2 The implementation delegates to the default shared conversion service to convert the values from the Map used to create a `LocalDate`

3 If an exception occurs during binding, call `reject(..)` which propagates additional information to the container

4.6 Using @EachProperty to Drive Configuration

The **@ConfigurationProperties** annotation is great for a single configuration class, but sometimes you want multiple instances, each with its own distinct configuration. That is where **EachProperty** comes in.

The **@EachProperty** annotation creates a `ConfigurationProperties` bean for each sub-property within the given property. As an example consider the following class:

Using @EachProperty

Java | Groovy | Kotlin

JAVA

```
import java.net.URI;
import java.net.URISyntaxException;

import io.micronaut.context.annotation.Parameter;
import io.micronaut.context.annotation.EachProperty;

@EachProperty("test.datasource")   1
public class DataSourceConfiguration {

    private final String name;
    private URI url = new URI("localhost");

    public DataSourceConfiguration(@Parameter String name)  2
            throws URISyntaxException {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public URI getUrl() {  3
        return url;
    }

    public void setUrl(URI url) {
        this.url = url;
    }
}
```

Copy to Clipboard

1 The `@EachProperty` annotation defines the property name to be handled.

2 The `@Parameter` annotation can be used to inject the name of the sub-property that defines the name of the bean (which is also the bean qualifier)

3 Each property of the bean is bound to configuration.

The above `DataSourceConfiguration` defines a `url` property to configure one or more data sources. The URLs themselves can be configured using any of the **PropertySource** instances evaluated to Micronaut:

Providing Configuration to @EachProperty

Java | Groovy | Kotlin

JAVA

```
        ApplicationContext applicationContext = ApplicationContext.run(PropertySource.of(
                "test",
                CollectionUtils.mapOf(
                        "test.datasource.one.url", "jdbc:mysql://localhost/one",
                        "test.datasource.two.url", "jdbc:mysql://localhost/two")
        ));
```

Copy to Clipboard

In the above example two data sources (called `one` and `two`) are defined under the `test.datasource` prefix defined earlier in the `@EachProperty` annotation. Each of these configuration entries triggers the creation of a new `DataSourceConfiguration` bean such that the following test succeeds:

Evaluating Beans Built by @EachProperty

Java | Groovy | Kotlin

JAVA

```
        Collection<DataSourceConfiguration> beansOfType = applicationContext.getBeansOfType(DataSourceConfiguration.c
        assertEquals(2, beansOfType.size());  1

        DataSourceConfiguration firstConfig = applicationContext.getBean(
                DataSourceConfiguration.class,
                Qualifiers.byName("one")  2
        );

        assertEquals(
                new URI("jdbc:mysql://localhost/one"),
                firstConfig.getUrl()
        );
```

Copy to Clipboard

1 All beans of type `DataSourceConfiguration` can be retrieved using `getBeansOfType`

2 Individual beans can be retrieved by using the `byName` qualifier.

List-Based Binding

The default behavior of **@EachProperty** is to bind from a map style of configuration, where the key is the named qualifier of the bean and the value is the data to bind from. For cases where map style configuration doesn't make sense, it is possible to inform Micronaut that the class is bound from a list. Simply set the `list` member on the annotation to true.

@EachProperty List Example

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.EachProperty;
import io.micronaut.context.annotation.Parameter;
import io.micronaut.core.order.Ordered;

import java.time.Duration;

@EachProperty(value = "ratelimits", list = true)  1
public class RateLimitsConfiguration implements Ordered {  2

    private final Integer index;
    private Duration period;
    private Integer limit;

    RateLimitsConfiguration(@Parameter Integer index) {  3
        this.index = index;
    }

    @Override
    public int getOrder() {
        return index;
    }

    public Duration getPeriod() {
        return period;
    }

    public void setPeriod(Duration period) {
        this.period = period;
    }

    public Integer getLimit() {
        return limit;
    }

    public void setLimit(Integer limit) {
        this.limit = limit;
    }
}
```

Copy to Clipboard

1. The `list` member of the annotation is set to `true`
2. Implement `Ordered` if order matters when retrieving the beans
3. The index is injected into the constructor

4.7 Using @EachBean to Drive Configuration

The **@EachProperty** annotation is a great way to drive dynamic configuration, but typically you want to inject that configuration into another bean that depends on it. Injecting a single instance with a hard-coded qualifier is not a great solution, hence `@EachProperty` is typically used in combination with **@EachBean**:

Using @EachBean

Java | Groovy | Kotlin

JAVA

```
@Factory  1
public class DataSourceFactory {

    @EachBean(DataSourceConfiguration.class)  2
    DataSource dataSource(DataSourceConfiguration configuration) {  3
        URI url = configuration.getUrl();
        return new DataSource(url);
    }
```

Copy to Clipboard

1. The above example defines a bean **Factory** that creates instances of `javax.sql.DataSource`.
2. The **@EachBean** annotation indicates that a new `DataSource` bean will be created for each `DataSourceConfiguration` defined in the previous section.

3 The `DataSourceConfiguration` instance is injected as a method argument and used to drive the configuration of each `javax.sql.DataSource`

Note that `@EachBean` requires that the parent bean has a `@Named` qualifier, since the qualifier is inherited by each bean created by `@EachBean`.

In other words, to retrieve the `DataSource` created by `test.datasource.one` you can do:

Using a Qualifier

Java | Groovy | Kotlin

JAVA

```
Collection<DataSource> beansOfType = applicationContext.getBeansOfType(DataSource.class);
assertEquals(2, beansOfType.size());  1

DataSource firstConfig = applicationContext.getBean(
        DataSource.class,
        Qualifiers.byName("one")  2
);
```

Copy to Clipboard

1 We demonstrate here that there are indeed two data sources. How can we get one in particular?

2 By using `Qualifiers.byName("one")`, we can select which of the two beans we'd like to reference.

4.8 Immutable Configuration

Since 1.3, Micronaut supports the definition of immutable configuration.

There are two ways to define immutable configuration. The preferred way is to define an interface annotated with **@ConfigurationProperties**. For example:

@ConfigurationProperties Example

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.ConfigurationProperties;
import io.micronaut.core.bind.annotation.Bindable;

import javax.validation.constraints.Min;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.util.Optional;

@ConfigurationProperties("my.engine")  1
public interface EngineConfig {

    @Bindable(defaultValue = "Ford")  2
    @NotBlank  3
    String getManufacturer();

    @Min(1L)
    int getCylinders();

    @NotNull
    CrankShaft getCrankShaft();  4

    @ConfigurationProperties("crank-shaft")
    interface CrankShaft {  5
        Optional<Double> getRodLength();  6
    }
}
```

Copy to Clipboard

1 The **@ConfigurationProperties** annotation takes the configuration prefix and is declared on an interface

2 You can use **@Bindable** to set a default value

3 Validation annotations can also be used

4 You can also specify references to other **@ConfigurationProperties** beans.

5 You can nest immutable configuration

6 Optional configuration can be indicated by returning an `Optional` or specifying `@Nullable`

In this case Micronaut provides a compile-time implementation that delegates all getters to call the `getProperty(..)` method of the **Environment** interface.

This has the advantage that if the application configuration is **refreshed** (for example by invoking the `/refresh` endpoint), the injected interface automatically sees the new values.

If you try to specify any other abstract method other than a getter, a compilation error occurs (default methods are supported).

Another way to implement immutable configuration is to define a class and use the **@ConfigurationInject** annotation on a constructor of a **@ConfigurationProperties** or **@EachProperty** bean.

For example:

@ConfigurationProperties Example

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.core.bind.annotation.Bindable;
import io.micronaut.core.annotation.Nullable;
import io.micronaut.context.annotation.ConfigurationInject;
import io.micronaut.context.annotation.ConfigurationProperties;

import javax.validation.constraints.Min;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.util.Optional;

@ConfigurationProperties("my.engine")  1
public class EngineConfig {

    private final String manufacturer;
    private final int cylinders;
    private final CrankShaft crankShaft;

    @ConfigurationInject  2
    public EngineConfig(
            @Bindable(defaultValue = "Ford") @NotBlank String manufacturer,  3
            @Min(1L) int cylinders,  4
            @NotNull CrankShaft crankShaft) {
        this.manufacturer = manufacturer;
        this.cylinders = cylinders;
        this.crankShaft = crankShaft;
    }

    public String getManufacturer() {
        return manufacturer;
    }

    public int getCylinders() {
        return cylinders;
    }

    public CrankShaft getCrankShaft() {
        return crankShaft;
    }

    @ConfigurationProperties("crank-shaft")
    public static class CrankShaft {  5
        private final Double rodLength;  6

        @ConfigurationInject
        public CrankShaft(@Nullable Double rodLength) {
            this.rodLength = rodLength;
        }

        public Optional<Double> getRodLength() {
            return Optional.ofNullable(rodLength);
        }
    }
}
```

Copy to Clipboard

1. The **@ConfigurationProperties** annotation takes the configuration prefix
2. The **@ConfigurationInject** annotation is defined on the constructor
3. You can use **@Bindable** to set a default value
4. Validation annotations can be used too
5. You can nest immutable configuration
6. Optional configuration can be indicated with `@Nullable`

The **@ConfigurationInject** annotation provides a hint to Micronaut to prioritize binding values from configuration instead of injecting beans.

Using this approach, to make the configuration refreshable, add the **@Refreshable** annotation to the class as well. This allows the bean to be re-created in the case of a **runtime configuration refresh event**.

There are a few exceptions to this rule. Micronaut will not perform configuration binding for a parameter if any of these conditions is met:

- The parameter is annotated with `@Value` (explicit binding)
- The parameter is annotated with `@Property` (explicit binding)
- The parameter is annotated with `@Parameter` (parameterized bean handling)
- The parameter is annotated with `@Inject` (generic bean injection)
- The type of the parameter is annotated with a bean scope (such as `@Singleton`)

Once you have prepared a type-safe configuration it can be injected into your beans like any other bean:

1. Inject the `EngineConfig` bean
2. Use the configuration properties

Configuration values can then be supplied when running the application. For example:

Supply Configuration

Java | Groovy | Kotlin

JAVA

```
ApplicationContext applicationContext = ApplicationContext.run(CollectionUtils.mapOf(
        "my.engine.cylinders", "8",
        "my.engine.crank-shaft.rod-length", "7.0"
));

Vehicle vehicle = applicationContext.getBean(Vehicle.class);
System.out.println(vehicle.start());
```

Copy to Clipboard

The above example prints: `"Ford Engine Starting V8 [rodLength=7B.0]"`

4.9 Bootstrap Configuration

Most application configuration is stored in `application.yml`, environment-specific files like `application-{environment}.{extension}`, environment and system properties, etc. These configure the application context. But during application startup, before the application context is created, a "bootstrap" context can be created to store configuration necessary to retrieve additional configuration for the main context. Typically that additional configuration is in some remote source.

The bootstrap context is enabled depending on the following conditions. The conditions are checked in the following order:

- If The **micronaut.bootstrap.context^** system property is set, that value determines if the bootstrap context is enabled.
- If The application context builder option **bootstrapEnvironment** is set, that value determines if the bootstrap context is enabled.
- If a **BootstrapPropertySourceLocator** bean is present the bootstrap context is enabled. Normally this comes from the `micronaut-discovery-client` dependency.

Configuration properties that must be present before application context configuration properties are resolved, for example when using distributed configuration, are stored in a bootstrap configuration file. Once it is determined the bootstrap context is enabled (as described above), the bootstrap configuration files are read using the same rules as regular application configuration. See the **property source** documentation for the details. The only difference is the prefix (`bootstrap` instead of `application`).

The file name prefix `bootstrap` is configurable with a system property **micronaut.bootstrap.name**.

Any configuration in the bootstrap context is not necessary to be duplicated in the main context. The bootstrap context configuration is carried over to the main context automatically. That means if a configuration property is needed in both places, it should go into the bootstrap context configuration.

See the **distributed configuration** section of the documentation for the list of integrations with common distributed configuration solutions.

Bootstrap Context Beans

In order for a bean to be resolvable in the bootstrap context it must be annotated with **@BootstrapContextCompatible**. If any given bean is not annotated then it will not be able to be resolved in the bootstrap context. Typically any bean that is participating in the process of retrieving distributed configuration needs to be annotated.

4.10 JMX Support

Micronaut provides basic support for JMX.

For more information, see the **documentation** (https://micronaut-projects.github.io/micronaut-jmx/latest/guide/) for the micronaut-jmx project.

5 Aspect Oriented Programming

Aspect-Oriented Programming (AOP) has historically had many incarnations and some very complicated implementations. Generally AOP can be thought of as a way to define cross-cutting concerns (logging, transactions, tracing, etc.) separate from application code in the form of aspects that define advice.

There are typically two forms of advice:

- Around Advice - decorates a method or class
- Introduction Advice - introduces new behaviour to a class.

In modern Java applications, declaring advice typically takes the form of an annotation. The most well-known annotation advice in the Java world is probably `@Transactional`, which demarcates transaction boundaries in Spring and Grails applications.

The disadvantage of traditional approaches to AOP is the heavy reliance on runtime proxy creation and reflection, which slows application performance, makes debugging harder and increases memory consumption.

Micronaut tries to address these concerns by providing a simple compile-time AOP API that does not use reflection.

5.1 Around Advice

The most common type of advice you may want to apply is "Around" advice, which lets you decorate a method's behaviour.

Writing Around Advice

The first step is to define an annotation that will trigger a **MethodInterceptor**:

Around Advice Annotation Example

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.aop.Around;
import java.lang.annotation.*;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Documented
@Retention(RUNTIME) 1
@Target({TYPE, METHOD}) 2
@Around 3
public @interface NotNull {
}
```

Copy to Clipboard

1. The retention policy of the annotation should be `RUNTIME`
2. Generally you want to be able to apply advice at the class or method level so the target types are `TYPE` and `METHOD`
3. The **@Around** annotation is added to tell Micronaut that the annotation is Around advice

The next step to defining Around advice is to implement a **MethodInterceptor**. For example the following interceptor disallows parameters with `null` values:

MethodInterceptor Example

Java | Groovy | Kotlin

JAVA

```java
import io.micronaut.aop.InterceptorBean;
import io.micronaut.aop.MethodInterceptor;
import io.micronaut.aop.MethodInvocationContext;
import io.micronaut.core.annotation.Nullable;
import io.micronaut.core.type.MutableArgumentValue;

import jakarta.inject.Singleton;
import java.util.Map;
import java.util.Objects;
import java.util.Optional;

@Singleton
@InterceptorBean(NotNull.class)  1
public class NotNullInterceptor implements MethodInterceptor<Object, Object> {  2
    @Nullable
    @Override
    public Object intercept(MethodInvocationContext<Object, Object> context) {
        Optional<Map.Entry<String, MutableArgumentValue<?>>> nullParam = context.getParameters()
            .entrySet()
            .stream()
            .filter(entry -> {
                MutableArgumentValue<?> argumentValue = entry.getValue();
                return Objects.isNull(argumentValue.getValue());
            })
            .findFirst();  3
        if (nullParam.isPresent()) {
            throw new IllegalArgumentException("Null parameter [" + nullParam.get().getKey() + "] not allowed");  4
        }
        return context.proceed();  5
    }
}
```

Copy to Clipboard

1 The **@InterceptorBean** annotation is used to indicate what annotation the interceptor is associated with. Note that **@InterceptorBean** is meta-annotated with a default scope of `@Singleton` therefore if you want a new interceptor created and associated with each intercepted bean you should annotate the interceptor with **@Prototype**.

2 An interceptor implements the **MethodInterceptor** interface.

3 The passed **MethodInvocationContext** is used to find the first parameter that is `null`

4 If a `null` parameter is found an exception is thrown

5 Otherwise **proceed()** is called to proceed with the method invocation.

Micronaut AOP interceptors use no reflection which improves performance and reducing stack trace sizes, thus improving debugging.

Apply the annotation to target classes to put the new `MethodInterceptor` to work:

Around Advice Usage Example

Java | Groovy | Kotlin

JAVA

```java
import jakarta.inject.Singleton;

@Singleton
public class NotNullExample {

    @NotNull
    void doWork(String taskName) {
        System.out.println("Doing job: " + taskName);
    }
}
```

Copy to Clipboard

Whenever the type `NotNullExample` is injected into a class, a compile-time-generated proxy is injected that decorates method calls with the `@NotNull` advice defined earlier. You can verify that the advice works by writing a test. The following test verifies that the expected exception is thrown when the argument is `null`:

Around Advice Test

Java | Groovy | Kotlin

JAVA

```
@Rule
public ExpectedException thrown = ExpectedException.none();

@Test
public void testNotNull() {
    try (ApplicationContext applicationContext = ApplicationContext.run()) {
        NotNullExample exampleBean = applicationContext.getBean(NotNullExample.class);

        thrown.expect(IllegalArgumentException.class);
        thrown.expectMessage("Null parameter [taskName] not allowed");

        exampleBean.doWork(null);
    }
}
```

Copy to Clipboard

Since Micronaut injection happens at compile time, generally the advice should be packaged in a dependent JAR file that is on the classpath when the above test is compiled. It should not be in the same codebase since you don't want the test to be compiled before the advice itself is compiled.

Customizing Proxy Generation

The default behaviour of the **Around** annotation is to generate a proxy at compile time that is a subclass of the proxied class. In other words, in the previous example a compile-time subclass of the `NotNullExample` class will be produced where proxied methods are decorated with interceptor handling, and the original behaviour is invoked via a call to `super`.

This behaviour is more efficient as only one instance of the bean is required, however depending on the use case you may wish to alter this behaviour. The `@Around` annotation supports various attributes that allow you to alter this behaviour, including:

- `proxyTarget` (defaults to `false`) - If set to `true`, instead of a subclass that calls `super`, the proxy delegates to the original bean instance
- `hotswap` (defaults to `false`) - Same as `proxyTarget=true`, but in addition the proxy implements **HotSwappableInterceptedProxy** which wraps each method call in a `ReentrantReadWriteLock` and allows swapping the target instance at runtime.
- `lazy` (defaults to `false`) - By default Micronaut eagerly initializes the proxy target when the proxy is created. If set to `true` the proxy target is instead resolved lazily for each method call.

AOP Advice on @Factory Beans

The semantics of AOP advice when applied to **Bean Factories** differs from regular beans, with the following rules applying:

1. AOP advice applied at the class level of a **@Factory** bean applies the advice to the factory itself and not to any beans defined with the **@Bean** annotation.
2. AOP advice applied on a method annotated with a bean scope applies the AOP advice to the bean that the factory produces.

Consider the following two examples:

AOP Advice at the type level of a `@Factory`

Java | Groovy | Kotlin

JAVA

```
@Timed
@Factory
public class MyFactory {

    @Prototype
    public MyBean myBean() {
        return new MyBean();
    }
}
```

Copy to Clipboard

The above example logs the time it takes to create the `MyBean` bean.

Now consider this example:

AOP Advice at the method level of a `@Factory`

Java | Groovy | Kotlin

JAVA

```
@Factory
public class MyFactory {

    @Prototype
    @Timed
    public MyBean myBean() {
        return new MyBean();
    }
}
```

Copy to Clipboard

The above example logs the time it takes to execute the public methods of the `MyBean` bean, but not the bean creation.

The rationale for this behaviour is that you may at times wish to apply advice to a factory and at other times apply advice to the bean produced by the factory.

Note that there is currently no way to apply advice at the method level to a **@Factory** bean, and all advice for factories must be applied at the type level. You can control which methods have advice applied by defining methods as non-public which do not have advice applied.

5.2 Introduction Advice

Introduction advice is distinct from Around advice in that it involves providing an implementation instead of decorating.

Examples of introduction advice include **GORM** (http://gorm.grails.org) and **Spring Data** (http://projects.spring.io/spring-data) which implement persistence logic for you.

Micronaut's **Client** annotation is another example of introduction advice where Micronaut implements HTTP client interfaces for you at compile time.

The way you implement Introduction advice is very similar to how you implement Around advice.

You start by defining an annotation that powers the introduction advice. As an example, say you want to implement advice to return a stubbed value for every method in an interface (a common requirement in testing frameworks). Consider the following `@Stub` annotation:

Introduction Advice Annotation Example

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.aop.Introduction;
import io.micronaut.context.annotation.Bean;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Introduction  1
@Bean  2
@Documented
@Retention(RUNTIME)
@Target({TYPE, ANNOTATION_TYPE, METHOD})
public @interface Stub {
    String value() default "";
}
```

Copy to Clipboard

1 The introduction advice is annotated with **Introduction**

2 The **Bean** annotation is added so that all types annotated with `@Stub` become beans

The `StubIntroduction` class referred to in the previous example must then implement the **MethodInterceptor** interface, just like around advice.

The following is an example implementation:

StubIntroduction

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.aop.*;
import io.micronaut.core.annotation.Nullable;
import jakarta.inject.Singleton;

@Singleton
@InterceptorBean(Stub.class) 1
public class StubIntroduction implements MethodInterceptor<Object,Object> { 2

    @Nullable
    @Override
    public Object intercept(MethodInvocationContext<Object, Object> context) {
        return context.getValue( 3
                Stub.class,
                context.getReturnType().getType()
        ).orElse(null); 4
    }
}
```

Copy to Clipboard

1 The **InterceptorBean** annotation is used to associate the interceptor with the `@Stub` annotation

2 The class is annotated with `@Singleton` and implements the **MethodInterceptor** interface

3 The value of the `@Stub` annotation is read from the context and an attempt made to convert the value to the return type

4 Otherwise `null` is returned

To now use this introduction advice in an application, annotate your abstract classes or interfaces with `@Stub`:

StubExample

Java | Groovy | Kotlin

JAVA

```
@Stub
public interface StubExample {

    @Stub("10")
    int getNumber();

    LocalDateTime getDate();
}
```

Copy to Clipboard

All abstract methods delegate to the `StubIntroduction` class to be implemented.

The following test demonstrates the behaviour or `StubIntroduction`:

Testing Introduction Advice

Java | Groovy | Kotlin

JAVA

```
        StubExample stubExample = applicationContext.getBean(StubExample.class);

        assertEquals(10, stubExample.getNumber());
        assertNull(stubExample.getDate());
```

Copy to Clipboard

Note that if the introduction advice cannot implement the method, call the `proceed` method of the **MethodInvocationContext**. This lets other introduction advice interceptors implement the method, and an `UnsupportedOperationException` will be thrown if no advice can implement the method.

In addition, if multiple introduction advice are present you may wish to override the `getOrder()` method of **MethodInterceptor** to control the priority of advice.

The following sections cover core advice types provided by Micronaut.

5.3 Method Adapter Advice

There are cases where you want to introduce a new bean based on the presence of an annotation on a method. An example of this is the **@EventListener** annotation which produces an implementation of **ApplicationEventListener** for each annotated method that invokes the annotated method.

For example the following snippet runs the logic contained within the method when the **ApplicationContext** starts up:

JAVA

```
import io.micronaut.context.event.StartupEvent;
import io.micronaut.runtime.event.annotation.EventListener;
...

@EventListener
void onStartup(StartupEvent event) {
    // startup logic here
}
```

The presence of the **@EventListener** annotation causes Micronaut to create a new class that implements **ApplicationEventListener** and invokes the `onStartup` method defined in the bean above.

The actual implementation of the **@EventListener** is trivial; it simply uses the **@Adapter** annotation to specify which SAM (single abstract method) type it adapts:

JAVA

```
import io.micronaut.aop.Adapter;
import io.micronaut.context.event.ApplicationEventListener;
import io.micronaut.core.annotation.Indexed;

import java.lang.annotation.*;

import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Documented
@Retention(RUNTIME)
@Target({ElementType.ANNOTATION_TYPE, ElementType.METHOD})
@Adapter(ApplicationEventListener.class)  1
@Indexed(ApplicationEventListener.class)
@Inherited
public @interface EventListener {
}
```

1 The **@Adapter** annotation indicates which SAM type to adapt, in this case **ApplicationEventListener**.

Micronaut also automatically aligns the generic types for the SAM interface if they are specified.

Using this mechanism you can define custom annotations that use the **@Adapter** annotation and a SAM interface to automatically implement beans for you at compile time.

5.4 Bean Life Cycle Advice

Sometimes you may need to apply advice to a bean's lifecycle. There are 3 types of advice that are applicable in this case:

- Interception of the construction of the bean
- Interception of the bean's `@PostConstruct` invocation
- Interception of a bean's `@PreDestroy` invocation

Micronaut supports these 3 use cases by allowing the definition of additional **@InterceptorBinding** meta-annotations.

Consider the following annotation definition:

AroundConstruct example

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.aop.*;
import io.micronaut.context.annotation.Prototype;
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@AroundConstruct  1
@InterceptorBinding(kind = InterceptorKind.POST_CONSTRUCT)  2
@InterceptorBinding(kind = InterceptorKind.PRE_DESTROY)  3
@Prototype  4
public @interface ProductBean {
}
```

Copy to Clipboard

1 The **@AroundConstruct** annotation is added to indicate that interception of the constructor should occur

2 An **@InterceptorBinding** definition is used to indicate that `@PostConstruct` interception should occur

3 An **@InterceptorBinding** definition is used to indicate that `@PreDestroy` interception should occur

4 The bean is defined as **@Prototype** so a new instance is required for each injection point

Note that if you do not need `@PostConstruct` and `@PreDestroy` interception you can simply remove those bindings.

The `@ProductBean` annotation can then be used on the target class:

Using an AroundConstruct meta-annotation

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.Parameter;

import jakarta.annotation.PreDestroy;

@ProductBean  1
public class Product {
    private final String productName;
    private boolean active = false;

    public Product(@Parameter String productName) {  2
        this.productName = productName;
    }

    public String getProductName() {
        return productName;
    }

    public boolean isActive() {
        return active;
    }

    public void setActive(boolean active) {
        this.active = active;
    }

    @PreDestroy  3
    void disable() {
        active = false;
    }
}
```

Copy to Clipboard

1 The `@ProductBean` annotation is defined on a class of type `Product`

2 The **@Parameter** annotation indicates that this bean requires an argument to complete constructions

3 Any `@PreDestroy` or `@PostConstruct` methods are executed last in the interceptor chain

Now you can define **ConstructorInterceptor** beans for constructor interception and **MethodInterceptor** beans for `@PostConstruct` or `@PreDestroy` interception.

The following factory defines a **ConstructorInterceptor** that intercepts construction of `Product` instances and registers them with a hypothetical `ProductService` validating the product name first:

Defining a constructor interceptor

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.aop.*;
import io.micronaut.context.annotation.Factory;

@Factory
public class ProductInterceptors {
    private final ProductService productService;

    public ProductInterceptors(ProductService productService) {
        this.productService = productService;
    }

    @InterceptorBean(ProductBean.class)
    ConstructorInterceptor<Product> aroundConstruct() {  1
        return context -> {
            final Object[] parameterValues = context.getParameterValues();  2
            final Object parameterValue = parameterValues[0];
            if (parameterValue == null || parameterValues[0].toString().isEmpty()) {
                throw new IllegalArgumentException("Invalid product name");
            }
            String productName = parameterValues[0].toString().toUpperCase();
            parameterValues[0] = productName;
            final Product product = context.proceed();  3
            productService.addProduct(product);
            return product;
        };
    }
}
```

Copy to Clipboard

1 A new **@InterceptorBean** is defined that is a **ConstructorInterceptor**

2 The constructor parameter values can be retrieved and modified as needed

3 The constructor can be invoked with the `proceed()` method

Defining **MethodInterceptor** instances that interceptor the `@PostConstruct` and `@PreDestroy` methods is no different from defining interceptors for regular methods. Note however that you can use the passed **MethodInvocationContext** to identify what kind of interception is occurring and adapt the code accordingly like in the following example:

Defining a constructor interceptor

Java | Groovy | Kotlin

JAVA

```
@InterceptorBean(ProductBean.class)  1
MethodInterceptor<Product, Object> aroundInvoke() {
    return context -> {
        final Product product = context.getTarget();
        switch (context.getKind()) {
            case POST_CONSTRUCT:  2
                product.setActive(true);
                return context.proceed();
            case PRE_DESTROY:  3
                productService.removeProduct(product);
                return context.proceed();
            default:
                return context.proceed();
        }
    };
}
```

Copy to Clipboard

1 A new **@InterceptorBean** is defined that is a **MethodInterceptor**

2 `@PostConstruct` interception is handled

3 `@PreDestroy` interception is handled

5.5 Validation Advice

Validation advice is one of the most common advice types you are likely to want to use in your application.

Validation advice is built on **Bean Validation JSR 380** (https://beanvalidation.org/2.0/spec/), a specification of the Java API for bean validation which ensures that the properties of a bean meet specific criteria, using `javax.validation` annotations such as `@NotNull`, `@Min`, and `@Max`.

Micronaut provides native support for the `javax.validation` annotations with the `micronaut-validation` dependency:

Gradle | **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut</groupId>
    <artifactId>micronaut-validation</artifactId>
</dependency>
```

Copy to Clipboard

Or full JSR 380 compliance with the `micronaut-hibernate-validator` dependency:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut</groupId>
    <artifactId>micronaut-hibernate-validator</artifactId>
</dependency>
```

Copy to Clipboard

See the section on **Bean Validation** for more information on how to apply validation rules to your bean classes.

5.6 Cache Advice

Like Spring and Grails, Micronaut provides caching annotations in the **io.micronaut.cache** (https://micronaut-projects.github.io/micronaut-cache/latest/api/io/micronaut/cache/package-summary.html) package.

The **CacheManager** (https://micronaut-projects.github.io/micronaut-cache/latest/api/io/micronaut/cache/CacheManager.html) interface allows different cache implementations to be plugged in as necessary.

The **SyncCache** (https://micronaut-projects.github.io/micronaut-cache/latest/api/io/micronaut/cache/SyncCache.html) interface provides a synchronous API for caching, whilst the **AsyncCache** (https://micronaut-projects.github.io/micronaut-cache/latest/api/io/micronaut/cache/AsyncCache.html) API allows non-blocking operation.

Cache Annotations

The following cache annotations are supported:

- **@Cacheable** (https://micronaut-projects.github.io/micronaut-cache/latest/api/io/micronaut/cache/annotation/Cacheable.html) - Indicates a method is cacheable in the specified cache
- **@CachePut** (https://micronaut-projects.github.io/micronaut-cache/latest/api/io/micronaut/cache/annotation/CachePut.html) - Indicates that the return value of a method invocation should be cached. Unlike `@Cacheable` the original operation is never skipped.
- **@CacheInvalidate** (https://micronaut-projects.github.io/micronaut-cache/latest/api/io/micronaut/cache/annotation/CacheInvalidate.html) - Indicates the invocation of a method should cause the invalidation of one or more caches.

Using one of these annotations activates the **CacheInterceptor** (https://micronaut-projects.github.io/micronaut-cache/latest/api/io/micronaut/cache/interceptor/CacheInterceptor), which in the case of `@Cacheable` caches the return value of the method.

The emitted result is cached if the method return type is a non-blocking type (either **CompletableFuture** (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html) or an instance of **Publisher** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html)) .

In addition, if the underlying Cache implementation supports non-blocking cache operations, cache values are read without blocking, resulting in non-blocking cache operations.

Configuring Caches

By default, **Caffeine** (https://github.com/ben-manes/caffeine) is used to create caches from application configuration. For example with `application.yml`:

Cache Configuration Example

YAML

```
micronaut:
  caches:
    my-cache:
      maximum-size: 20
```

The above example configures a cache called "my-cache" with a maximum size of 20.

Naming Caches

Define names of caches under `micronaut.caches` in kebab case (lowercase and hyphen separated); if you use camel case, the names are normalized to kebab case. For example `myCache` becomes `my-cache`. The kebab-case form must be used when referencing caches in the **@Cacheable** (https://micronaut-projects.github.io/micronaut-cache/latest/api/io/micronaut/cache/annotation/Cacheable.html) annotation.

To configure a weigher to be used with the `maximumWeight` configuration, create a bean that implements `io.micronaut.caffeine.cache.Weigher`. To associate a given weigher with only a specific cache, annotate the bean with `@Named(<cache name>)`. Weighers without a named qualifier apply to all caches that don't have a named weigher. If no beans are found, a default implementation is used.

See the configuration reference (https://micronaut-projects.github.io/micronaut-cache/latest/guide/configurationreference.html#io.micronaut.cache.caffeine.DefaultCacheConfiguration) for all available configuration options.

Dynamic Cache Creation

A DynamicCacheManager (https://micronaut-projects.github.io/micronaut-cache/latest/api/io/micronaut/cache/DynamicCacheManager.html) bean can be registered for use cases where caches cannot be configured ahead of time. When a cache is attempted to be retrieved that was not predefined, the dynamic cache manager is invoked to create and return a cache.

By default, if there is no other dynamic cache manager defined in the application, Micronaut registers an instance of DefaultDynamicCacheManager (https://micronaut-projects.github.io/micronaut-cache/latest/api/io/micronaut/cache/caffeine/DefaultDynamicCacheManager.html) that creates Caffeine caches with default values.

Other Cache Implementations

Check the Micronaut Cache (https://micronaut-projects.github.io/micronaut-cache/latest/guide/index.html) project for more information.

5.7 Retry Advice

In distributed systems and microservice environments, failure is something you have to plan for, and it is common to want to attempt to retry an operation if it fails. If first you don't succeed try again!

With this in mind, Micronaut includes a Retryable annotation.

Simple Retry

The simplest form of retry is just to add the `@Retryable` annotation to a type or method. The default behaviour of `@Retryable` is to retry three times with an exponential delay of one second between each retry. (first attempt with 1s delay, second attempt with 2s delay, third attempt with 3s delay).

For example:

Simple Retry Example

Java | Groovy | Kotlin

JAVA

```
@Retryable
public List<Book> listBooks() {
    // ...
```

Copy to Clipboard

With the above example if the `listBooks()` method throws a RuntimeException, it is retried until the maximum number of attempts is reached.

The `multiplier` value of the `@Retryable` annotation can be used to configure a multiplier used to calculate the delay between retries, allowing exponential retry support.

Note also that the `@Retryable` annotation can be applied on interfaces, and the behaviour is inherited through annotation metadata. The implication of this is that `@Retryable` can be used in combination with **Introduction Advice** such as the HTTP Client annotation.

To customize retry behaviour, set the `attempts` and `delay` members, For example to configure five attempts with a two second delay:

Setting Retry Attempts

Java | Groovy | Kotlin

JAVA

```
@Retryable(attempts = "5",
           delay = "2s")
public Book findBook(String title) {
    // ...
```

Copy to Clipboard

Notice how both `attempts` and `delay` are defined as strings. This is to support configurability through annotation metadata. For example, you can allow the retry policy to be configured using property placeholder resolution:

Setting Retry via Configuration

Java | Groovy | Kotlin

JAVA

```
@Retryable(attempts = "${book.retry.attempts:3}",
           delay = "${book.retry.delay:1s}")
public Book getBook(String title) {
    // ...
```

Copy to Clipboard

With the above in place, if `book.retry.attempts` is specified in configuration it is bound to the value of the `attempts` member of the `@Retryable` annotation via annotation metadata.

Reactive Retry

`@Retryable` advice can also be applied to methods that return reactive types, such as `Publisher` (**Project Reactor** (https://projectreactor.io)'s `Flux` or **RxJava** (https://github.com/ReactiveX/RxJava)'s `Flowable`). For example:

Applying Retry Policy to Reactive Types

Java | Groovy | Kotlin

JAVA

```
@Retryable
public Publisher<Book> streamBooks() {
    // ...
```

Copy to Clipboard

In this case `@Retryable` advice applies the retry policy to the reactive type.

Circuit Breaker

Retry is useful in a microservice environment, but in some cases excessive retries can overwhelm the system as clients repeatedly re-attempt failing operations.

The **Circuit Breaker** (https://en.wikipedia.org/wiki/Circuit_breaker_design_pattern) pattern is designed to resolve this issue by allowing a certain number of failing requests and then opening a circuit that remains open for a period before allowing additional retry attempts.

The **CircuitBreaker** annotation is a variation of the `@Retryable` annotation that supports a `reset` member which indicates how long the circuit should remain open before it is reset (the default is 20 seconds).

Applying CircuitBreaker Advice

Java | Groovy | Kotlin

JAVA

```
@CircuitBreaker(reset = "30s")
public List<Book> findBooks() {
    // ...
```

Copy to Clipboard

The above example retries the `findBooks` method three times and then opens the circuit for 30 seconds, rethrowing the original exception and preventing potential downstream traffic such as HTTP requests and I/O operations flooding the system.

Factory Bean Retry

When **@Retryable** is applied to bean factory methods, it behaves as if the annotation was placed on the type being returned. The retry behavior applies when the methods on the returned object are invoked. Note that the bean factory method itself is **not** retried. If you want the functionality of creating the bean to be retried, it should be delegated to another singleton that has the **@Retryable** annotation applied.

For example:

JAVA

```
@Factory  1
public class Neo4jDriverFactory {
    ...
    @Retryable(ServiceUnavailableException.class)  2
    @Bean(preDestroy = "close")
    public Driver buildDriver() {
        ...
    }
}
```

1 A factory bean is created that defines methods that create beans

2 The **@Retryable** annotation is used to catch exceptions thrown from methods executed on the `Driver`.

Retry Events

You can register **RetryEventListener** instances as beans to listen for **RetryEvent** events that are published every time an operation is retried.

In addition, you can register event listeners for **CircuitOpenEvent** to be notified when a circuit breaker circuit is opened, or **CircuitClosedEvent** for when a circuit is closed.

5.8 Scheduled Tasks

Like Spring and Grails, Micronaut features a **Scheduled** annotation for scheduling background tasks.

Using the @Scheduled Annotation

The **Scheduled** annotation can be added to any method of a bean, and you should set one of the `fixedRate`, `fixedDelay`, or `cron` members.

Remember that the scope of a bean impacts behaviour. A `@Singleton` bean shares state (the fields of the instance) each time the scheduled method is executed, while for a `@Prototype` bean a new instance is created for each execution.

Scheduling at a Fixed Rate

To schedule a task at a fixed rate, use the `fixedRate` member. For example:

Fixed Rate Example

Java | Groovy | Kotlin

JAVA

```
@Scheduled(fixedRate = "5m")
void everyFiveMinutes() {
    System.out.println("Executing everyFiveMinutes()");
}
```

Copy to Clipboard

The task above executes every five minutes.

Scheduling with a Fixed Delay

To schedule a task so it runs five minutes after the termination of the previous task use the `fixedDelay` member. For example:

Fixed Delay Example

Java | Groovy | Kotlin

JAVA

```
@Scheduled(fixedDelay = "5m")
void fiveMinutesAfterLastExecution() {
    System.out.println("Executing fiveMinutesAfterLastExecution()");
}
```

Copy to Clipboard

Scheduling a Cron Task

To schedule a **Cron** (https://en.wikipedia.org/wiki/Cron) task use the `cron` member:

Cron Example

Java | Groovy | Kotlin

JAVA

```
@Scheduled(cron = "0 15 10 ? * MON")
void everyMondayAtTenFifteenAm() {
    System.out.println("Executing everyMondayAtTenFifteenAm()");
}
```

Copy to Clipboard

The above example runs the task every Monday morning at 10:15AM in the time zone of the server.

Scheduling with only an Initial Delay

To schedule a task so it runs once after the server starts, use the `initialDelay` member:

Initial Delay Example

Java | Groovy | Kotlin

JAVA

```
@Scheduled(initialDelay = "1m")
void onceOneMinuteAfterStartup() {
    System.out.println("Executing onceOneMinuteAfterStartup()");
}
```

Copy to Clipboard

The above example only runs once, one minute after the server starts.

Programmatically Scheduling Tasks

To programmatically schedule tasks, use the **TaskScheduler** bean which can be injected as follows:

Java | Groovy | Kotlin

JAVA

```
@Inject
@Named(TaskExecutors.SCHEDULED)
TaskScheduler taskScheduler;
```

Copy to Clipboard

Configuring Scheduled Tasks with Annotation Metadata

To make your application's tasks configurable, you can use annotation metadata and property placeholder configuration. For example:

Allow tasks to be configured

Java | Groovy | Kotlin

JAVA

```
@Scheduled(fixedRate = "${my.task.rate:5m}",
        initialDelay = "${my.task.delay:1m}")
void configuredTask() {
    System.out.println("Executing configuredTask()");
}
```

Copy to Clipboard

The above example allows the task execution frequency to be configured with the property `my.task.rate`, and the initial delay to be configured with the property `my.task.delay`.

Configuring the Scheduled Task Thread Pool

Tasks executed by `@Scheduled` are run by default on a **ScheduledExecutorService** (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledExecutorService.html) configured to have twice the number of threads as available processors.

You can configure this thread pool using `application.yml`, for example:

Configuring Scheduled Task Thread Pool

YAML

```
micronaut:
  executors:
    scheduled:
      type: scheduled
      core-pool-size: 30
```

Unresolved directive in <stdin> - include::/home/runner/work/micronaut-core/micronaut-core/build/generated/configurationProperties/io.micronaut.scheduling.executor.UserExecutorConfiguration.adoc[]

Handling Exceptions

By default, Micronaut includes a **DefaultTaskExceptionHandler** bean that implements the **TaskExceptionHandler** interface and simply logs the exception if an error occurs invoking a scheduled task.

If you have custom requirements you can replace this bean with your own implementation (for example to send an email or shutdown the context to fail fast). To do so, write your own **TaskExceptionHandler** and annotate it with `@Replaces(DefaultTaskExceptionHandler.class)`.

5.9 Bridging Spring AOP

Although Micronaut's design is based on a compile-time approach and does not rely on Spring dependency injection, there is still a lot of value in the Spring ecosystem that does not depend directly on the Spring container.

You may wish to use existing Spring projects within Micronaut and configure beans to be used within Micronaut.

You may also wish to leverage existing AOP advice from Spring. One example of this is Spring's support for declarative transactions with `@Transactional`.

Micronaut provides support for Spring-based transaction management without requiring Spring itself. Simply add the `spring` module to your application dependencies:

Gradle | **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.spring</groupId>
    <artifactId>micronaut-spring</artifactId>
</dependency>
```

Copy to Clipboard

If you use Micronaut's **Hibernate support** you already get this dependency and a `HibernateTransactionManager` is configured for you.

This is done by defining a Micronaut **Transactional** (https://micronaut-projects.github.io/micronaut-spring/latest/api/io/micronaut/spring/tx/annotation/Transactional.html) annotation that uses **@AliasFor** in a manner that every time you set a value with **Transactional** (https://micronaut-projects.github.io/micronaut-spring/latest/api/io/micronaut/spring/tx/annotation/Transactional.html) it aliases the value to the equivalent value in Spring's version of `@Transactional`.

The benefit here is you can use Micronaut's compile-time, reflection-free AOP to declare programmatic Spring transactions. For example:

Using @Transactional

JAVA

```
import io.micronaut.spring.tx.annotation.*;
...

@Transactional
public Book saveBook(String title) {
    ...
}
```

Micronaut's version of **Transactional** (https://micronaut-projects.github.io/micronaut-spring/latest/api/io/micronaut/spring/tx/annotation/Transactional.html) is meta-annotated with **@Blocking**, ensuring that all methods annotated with it use the I/O thread pool when executing within the HTTP server

6 The HTTP Server

Using the CLI

If you create your project using the Micronaut CLI `create-app` command, the `http-server` dependency is included by default.

Micronaut includes both non-blocking HTTP server and client APIs based on **Netty** (https://netty.io).

The design of the HTTP server in Micronaut is optimized for interchanging messages between Microservices, typically in JSON, and is not intended as a full server-side MVC framework. For example, there is currently no support for server-side views or features typical of a traditional server-side MVC framework.

The goal of the HTTP server is to make it as easy as possible to expose APIs to be consumed by HTTP clients, regardless of the language they are written in. To use the HTTP server you need the `http-server-netty` dependency in your build:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut</groupId>
    <artifactId>micronaut-http-server-netty</artifactId>
</dependency>
```

Copy to Clipboard

A "Hello World" server application can be seen below:

Java Groovy Kotlin

JAVA

```
import io.micronaut.http.MediaType;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;

@Controller("/hello")  1
public class HelloController {

    @Get(produces = MediaType.TEXT_PLAIN)  2
    public String index() {
        return "Hello World";  3
    }
}
```

Copy to Clipboard

1. The class is defined as a controller with the **@Controller** annotation mapped to the path `/hello`
2. The method responds to a GET requests to `/hello` and returns a response with a `text/plain` content type
3. By defining a method named `index`, by convention the method is exposed via the `/hello` URI

6.1 Running the Embedded Server

To run the server, create an `Application` class with a `static void main` method, for example:

Java Groovy Kotlin

```java
import io.micronaut.runtime.Micronaut;

public class Application {

    public static void main(String[] args) {
        Micronaut.run(Application.class);
    }
}
```

To run the application from a unit test, use the **EmbeddedServer** interface:

Java | Groovy | Kotlin

```java
import io.micronaut.http.HttpRequest;
import io.micronaut.http.client.HttpClient;
import io.micronaut.http.client.annotation.Client;
import io.micronaut.runtime.server.EmbeddedServer;
import io.micronaut.test.extensions.junit5.annotation.MicronautTest;
import org.junit.jupiter.api.Test;

import jakarta.inject.Inject;

import static org.junit.jupiter.api.Assertions.assertEquals;

@MicronautTest
public class HelloControllerSpec {

    @Inject
    EmbeddedServer server; 1

    @Inject
    @Client("/")
    HttpClient client; 2

    @Test
    void testHelloWorldResponse() {
        String response = client.toBlocking() 3
                .retrieve(HttpRequest.GET("/hello"));
        assertEquals("Hello World", response); 4
    }
}
```

1 The `EmbeddedServer` is run and the Spock `@AutoCleanup` annotation ensures the server is stopped after the specification completes.

2 The `EmbeddedServer` interface provides the `URL` of the server under test which runs on a random port.

3 The test uses the Micronaut HTTP client to make the call

4 The `retrieve` method returns the response of the controller as a `String`

Without explicit port configuration, the port will be 8080, unless the application is run under the `test` environment where the port is random. When the application context starts from the context of a test class, the test environment is added automatically.

6.2 Running Server on a Specific Port

By default the server runs on port 8080. However, you can set the server to run on a specific port:

```yaml
micronaut:
  server:
    port: 8086
```

This is also configurable from an environment variable, e.g. `MICRONAUT_SERVER_PORT=8086`

To run on a random port:

```yaml
micronaut:
  server:
    port: -1
```

Setting an explicit port may cause tests to fail if multiple servers start simultaneously on the same port. To prevent that, specify a random port in the test environment configuration.

6.3 HTTP Routing

The **@Controller** annotation used in the previous section is one of **several annotations** that allow you to control the construction of HTTP routes.

URI Paths

The value of the `@Controller` annotation is a **RFC-6570 URI template** (https://tools.ietf.org/html/rfc6570), so you can embed URI variables within the path using the syntax defined by the URI template specification.

Many other frameworks, including Spring, implement the URI template specification

The actual implementation is handled by the **UriMatchTemplate** class, which extends **UriTemplate**.

You can use this class in your applications to build URIs, for example:

Using a UriTemplate

Java | Groovy | Kotlin

JAVA

```
UriMatchTemplate template = UriMatchTemplate.of("/hello/{name}");

assertTrue(template.match("/hello/John").isPresent());  1
assertEquals("/hello/John", template.expand(  2
        Collections.singletonMap("name", "John")
));
```

Copy to Clipboard

1 Use the `match` method to match a path

2 Use the `expand` method to expand a template into a URI

You can use **UriTemplate** to build paths to include in your responses.

URI Path Variables

URI variables can be referenced via method arguments. For example:

URI Variables Example

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.annotation.PathVariable;

@Controller("/issues")  1
public class IssuesController {

    @Get("/{number}")  2
    public String issue(@PathVariable Integer number) {  3
        return "Issue # " + number + "!";  4
    }
}
```

Copy to Clipboard

1 The `@Controller` annotation is specified with a base URI of `/issues`

2 The **Get** annotation maps the method to an HTTP **GET** with a URI variable embedded in the URI named `number`

3 The method argument can optionally be annotated with **PathVariable**

4 The value of the URI variable is referenced in the implementation

Micronaut maps the URI `/issues/{number}` for the above controller. We can assert this is the case by writing unit tests:

Testing URI Variables

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.ApplicationContext;
import io.micronaut.http.client.HttpClient;
import io.micronaut.http.client.exceptions.HttpClientResponseException;
import io.micronaut.runtime.server.EmbeddedServer;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertThrows;

public class IssuesControllerTest {

    private static EmbeddedServer server;
    private static HttpClient client;

    @BeforeClass 1
    public static void setupServer() {
        server = ApplicationContext.run(EmbeddedServer.class);
        client = server
                    .getApplicationContext()
                    .createBean(HttpClient.class, server.getURL());
    }

    @AfterClass 2
    public static void stopServer() {
        if (server != null) {
            server.stop();
        }
        if (client != null) {
            client.stop();
        }
    }

    @Test
    public void testIssue() {
        String body = client.toBlocking().retrieve("/issues/12"); 3

        assertNotNull(body);
        assertEquals("Issue # 12!", body); 4
    }

    @Test
    public void testShowWithInvalidInteger() {
        HttpClientResponseException e = assertThrows(HttpClientResponseException.class, () ->
                client.toBlocking().exchange("/issues/hello"));

        assertEquals(400, e.getStatus().getCode()); 5
    }

    @Test
    public void testIssueWithoutNumber() {
        HttpClientResponseException e = assertThrows(HttpClientResponseException.class, () ->
                client.toBlocking().exchange("/issues/"));

        assertEquals(404, e.getStatus().getCode()); 6
    }
}
```

Copy to Clipboard

1. The embedded server and HTTP client are started
2. The server and client are cleaned up after the tests finish
3. The tests send a request to the URI `/issues/12`
4. And then asserts the response is "Issue # 12"
5. Another test asserts a 400 response is returned when an invalid number is sent in the URL
6. Another test asserts a 404 response is returned when no number is provided in the URL. The variable being present is required for the route to be executed.

Note that the URI template in the previous example requires that the `number` variable is specified. You can specify optional URI templates with the syntax: `/issues{/number}` and by annotating the `number` parameter with `@Nullable`.

The following table provides examples of URI templates and what they match:

Table 1. URI Template Matching

Template	Description	Matching URI
`/books/{id}`	Simple match	`/books/1`
`/books/{id:2}`	A variable of two characters max	`/books/10`
`/books{/id}`	An optional URI variable	`/books/10` or `/books`
`/book{/id:[a-zA-Z]+}`	An optional URI variable with regex	`/books/foo`
`/books{?max,offset}`	Optional query parameters	`/books?max=10&offset=10`
`/books{/path:.*}{.ext}`	Regex path match with extension	`/books/foo/bar.xml`

URI Reserved Character Matching

By default, URI variables as defined by the **RFC-6570 URI template** (https://tools.ietf.org/html/rfc6570) spec cannot include reserved characters such as `/`, `?` etc.

This can be problematic if you wish to match or expand entire paths. As per **section 3.2.3 of the specification** (https://tools.ietf.org/html/rfc6570#section-3.2.3), you can use reserved expansion or matching using the `+` operator.

For example the URI `/books/{+path}` matches both `/books/foo` and `/books/foo/bar` since the `+` indicates that the variable `path` should include reserved characters (in this case `/`).

Routing Annotations

The previous example uses the **@Get** annotation to add a method that accepts HTTP **GET** requests. The following table summarizes the available annotations and how they map to HTTP methods:

Table 2. HTTP Routing Annotations

Annotation	HTTP Method
@Delete	**DELETE**
@Get	**GET**
@Head	**HEAD**
@Options	**OPTIONS**
@Patch	**PATCH**
@Put	**PUT**
@Post	**POST**
@Trace	**TRACE**

All the method annotations default to `/`.

Multiple URIs

Each of the routing annotations supports multiple URI templates. For each template, a route is created. This feature is useful for example to change the path of the API and leave the existing path as is for backwards compatibility. For example:

Multiple URIs

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;

@Controller("/hello")
public class BackwardCompatibleController {

    @Get(uris = {"/{name}", "/person/{name}"})  1
    public String hello(String name) {  2
        return "Hello, " + name;
    }
}
```

Copy to Clipboard

1 Specify multiple templates

2 Bind to the template arguments as normal

Route validation is more complicated with multiple templates. If a variable that would normally be required does not exist in all templates, that variable is considered optional since it may not exist for every execution of the method.

Building Routes Programmatically

If you prefer to not use annotations and instead declare all routes in code then never fear, Micronaut has a flexible **RouteBuilder** API that makes it a breeze to define routes programmatically.

To start, subclass **DefaultRouteBuilder** and inject the controller to route to into the method, and define your routes:

URI Variables Example

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.ExecutionHandleLocator;
import io.micronaut.web.router.DefaultRouteBuilder;

import jakarta.inject.Inject;
import jakarta.inject.Singleton;

@Singleton
public class MyRoutes extends DefaultRouteBuilder {  1

    public MyRoutes(ExecutionHandleLocator executionHandleLocator,
                    UriNamingStrategy uriNamingStrategy) {
        super(executionHandleLocator, uriNamingStrategy);
    }

    @Inject
    void issuesRoutes(IssuesController issuesController) {  2
        GET("/issues/show/{number}", issuesController, "issue", Integer.class);  3
    }
}
```

Copy to Clipboard

1 Route definitions should subclass **DefaultRouteBuilder**

2 Use `@Inject` to inject a method with the controller to route to

3 Use methods such as `RouteBuilder::GET(String,Class,String,Class…)` to route to controller methods. Note that even though the issues controller is used, the route has no knowledge of its `@Controller` annotation and thus the full path must be specified.

Unfortunately due to type erasure, a Java method lambda reference cannot be used with the API. For Groovy there is a `GroovyRouteBuilder` class which can be subclassed that allows passing Groovy method references.

Route Compile-Time Validation

Micronaut supports validating route arguments at compile time with the validation library. To get started, add the `validation` dependency to your build:

build.gradle

GROOVY

```
annotationProcessor "io.micronaut:micronaut-validation" // Java only
kapt "io.micronaut:micronaut-validation" // Kotlin only
implementation "io.micronaut:micronaut-validation"
```

With the correct dependency on your classpath, route arguments will automatically be checked at compile time. Compilation will fail if any of the following conditions are met:

- The URI template contains a variable that is optional, but the method parameter is not annotated with `@Nullable` or is an `java.util.Optional`.

An optional variable is one that allows the route to match a URI even if the value is not present. For example `/foo{/bar}` matches requests to `/foo` and `/foo/abc`. The non-optional variant would be `/foo/{bar}`. See the **URI Path Variables** section for more information.

- The URI template contains a variable that is missing from the method arguments.

To disable route compile-time validation, set the system property `-Dmicronaut.route.validation=false`. For Java and Kotlin users using Gradle, the same effect can be achieved by removing the `validation` dependency from the `annotationProcessor`/`kapt` scope.

Routing non-standard HTTP methods

The `@CustomHttpMethod` annotation supports non-standard HTTP methods for a client or server. Specifications like [RFC-4918 Webdav](https://tools.ietf.org/html/rfc4918) require additional methods like REPORT or LOCK for example.

RoutingExample

JAVA

```
@CustomHttpMethod(method = "LOCK", value = "/{name}")
String lock(String name)
```

The annotation can be used anywhere the standard method annotations can be used, including controllers and declarative HTTP clients.

6.4 Simple Request Binding

The examples in the previous section demonstrate how Micronaut lets you bind method parameters from URI path variables. This section shows how to bind arguments from other parts of the request.

Binding Annotations

All binding annotations support customization of the name of the variable being bound from with their `name` member.

The following table summarizes the annotations and their purpose, and provides examples:

Table 1. Parameter Binding Annotations

Annotation	Description	Example
@Body	Binds from the body of the request	`@Body String body`
@CookieValue	Binds a parameter from a cookie	`@CookieValue String myCookie`
@Header	Binds a parameter from an HTTP header	`@Header String requestId`
@QueryValue	Binds from a request query parameter	`@QueryValue String myParam`
@Part	Binds from a part of a multipart request	`@Part CompletedFileUpload file`
@RequestAttribute	Binds from an attribute of the request. Attributes are typically created in filters	`@RequestAttribute String myAttribute`
@PathVariable	Binds from the path of the request	`@PathVariable String id`
@RequestBean	Binds any Bindable value to single Bean object	`@RequestBean MyBean bean`

The method parameter name is used when a value is not specified in a binding annotation. In other words the following two methods are equivalent and both bind from a cookie named `myCookie`:

Java | Groovy | Kotlin

JAVA

```
@Get("/cookieName")
public String cookieName(@CookieValue("myCookie") String myCookie) {
    // ...
}

@Get("/cookieInferred")
public String cookieInferred(@CookieValue String myCookie) {
    // ...
}
```

Copy to Clipboard

Because hyphens are not allowed in variable names, it may be necessary to set the name in the annotation. The following definitions are equivalent:

Java | Groovy | Kotlin

JAVA

```
@Get("/headerName")
public String headerName(@Header("Content-Type") String contentType) {
    // ...
}

@Get("/headerInferred")
public String headerInferred(@Header String contentType) {
    // ...
}
```

Copy to Clipboard

Stream Support

Micronaut also supports binding the body to an `InputStream`. If the method is reading the stream, the method execution must be offloaded to another thread pool to avoid blocking the event loop.

Performing Blocking I/O With InputStream

Java | Groovy | Kotlin

JAVA

```
@Post(value = "/read", processes = MediaType.TEXT_PLAIN)
@ExecuteOn(TaskExecutors.IO)  1
String read(@Body InputStream inputStream) throws IOException {  2
    return IOUtils.readText(new BufferedReader(new InputStreamReader(inputStream)));  3
}
```

Copy to Clipboard

1 The controller method is executed on the IO thread pool

2 The body is passed to the method as an input stream

3 The stream is read

Binding from Multiple Query values

Instead of binding from a single section of the request, it may be desirable to bind all query values for example to a POJO. This can be achieved by using the exploded operator (`?pojo*`) in the URI template. For example:

Binding Request parameters to POJO

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.HttpStatus;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.core.annotation.Nullable;

import javax.validation.Valid;

@Controller("/api")
public class BookmarkController {

    @Get("/bookmarks/list{?paginationCommand*}")
    public HttpStatus list(@Valid @Nullable PaginationCommand paginationCommand) {
        return HttpStatus.OK;
    }
}
```

Copy to Clipboard

Binding from Multiple Bindable values

Instead of binding just query values, it is also possible to bind any Bindable value to a POJO (e.g. to bind `HttpRequest`, `@PathVariable`, `@QueryValue` and `@Header` to a single POJO). This can be achieved with the `@RequestBean` annotation and a custom Bean class with fields with Bindable annotations, or fields that can be bound by type (e.g. `HttpRequest`, `BasicAuth`, `Authentication`, etc.).

For example:

Binding Bindable values to POJO

Java | Groovy | Kotlin

JAVA

```
@Controller("/api")
public class MovieTicketController {

    // You can also omit query parameters like:
    // @Get("/movie/ticket/{movieId}
    @Get("/movie/ticket/{movieId}{?minPrice,maxPrice}")
    public HttpStatus list(@Valid @RequestBean MovieTicketBean bean) {
        return HttpStatus.OK;
    }
}
```

Copy to Clipboard

which uses this bean class:

Bean definition

Java | Groovy | Kotlin

JAVA

```
@Introspected
public class MovieTicketBean {

    private HttpRequest<?> httpRequest;

    @PathVariable
    private String movieId;

    @Nullable
    @QueryValue
    @PositiveOrZero
    private Double minPrice;

    @Nullable
    @QueryValue
    @PositiveOrZero
    private Double maxPrice;

    public MovieTicketBean(HttpRequest<?> httpRequest,
                           String movieId,
                           Double minPrice,
                           Double maxPrice) {
        this.httpRequest = httpRequest;
        this.movieId = movieId;
        this.minPrice = minPrice;
        this.maxPrice = maxPrice;
    }

    public HttpRequest<?> getHttpRequest() {
        return httpRequest;
    }

    public String getMovieId() {
        return movieId;
    }

    @Nullable
    public Double getMaxPrice() {
        return maxPrice;
    }

    @Nullable
    public Double getMinPrice() {
        return minPrice;
    }
}
```

Copy to Clipboard

The bean class has to be introspected with `@Introspected`. It can be one of:

1. Mutable Bean class with setters and getters
2. Immutable Bean class with getters and an all-argument constructor (or `@Creator` annotation on a constructor or static method). Arguments of the constructor must match field names so the object can be instantiated without reflection.

Since Java does not retain argument names in bytecode, you must compile code with `-parameters` to use an immutable bean class from another jar. Another option is to extend Bean class in your source.

Bindable Types

Generally any type that can be converted from a String representation to a Java type via the **ConversionService** API can be bound to.

This includes most common Java types, however additional **TypeConverter** instances can be registered by creating `@Singleton` beans of type `TypeConverter`.

The handling of nullability deserves special mention. Consider for example the following example:

Java | Groovy | Kotlin

JAVA

```
    @Get("/headerInferred")
    public String headerInferred(@Header String contentType) {
        // ...
    }
```

Copy to Clipboard

In this case, if the HTTP header `Content-Type` is not present in the request, the route is considered invalid, since it cannot be satisfied, and a HTTP 400 `BAD REQUEST` is returned.

To make the `Content-Type` header optional, you can instead write:

Java | Groovy | Kotlin

JAVA

```
@Get("/headerNullable")
public String headerNullable(@Nullable @Header String contentType) {
    // ...
}
```

Copy to Clipboard

A `null` string is passed if the header is absent from the request.

`java.util.Optional` can also be used, but that is discouraged for method parameters.

Additionally, any `DateTime` that conforms to **RFC-1123** (https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#RFC_1123_DATE_TIME) can be bound to a parameter. Alternatively the format can be customized with the **Format** annotation:

Java | Groovy | Kotlin

JAVA

```
@Get("/date")
public String date(@Header ZonedDateTime date) {
    // ...
}

@Get("/dateFormat")
public String dateFormat(@Format("dd/MM/yyyy hh:mm:ss a z") @Header ZonedDateTime date) {
    // ...
}
```

Copy to Clipboard

Type-Based Binding Parameters

Some parameters are recognized by their type instead of their annotation. The following table summarizes the parameter types, their purpose, and provides an example:

Type	Description	Example
BasicAuth	Allows binding of basic authorization credentials	`BasicAuth basicAuth`

Variable resolution

Micronaut tries to populate method arguments in the following order:

1. URI variables like `/{id}`.
2. From query parameters if the request is a `GET` request (e.g. `?foo=bar`).
3. If there is a `@Body` and request allows the body, bind the body to it.
4. If the request can have a body and no `@Body` is defined then try to parse the body (either JSON or form data) and bind the method arguments from the body.
5. Finally, if the method arguments cannot be populated return `400 BAD REQUEST`.

6.5 Custom Argument Binding

Micronaut uses an **ArgumentBinderRegistry** to look up **ArgumentBinder** beans capable of binding to the arguments in controller methods. The default implementation looks for an annotation on the argument that is meta-annotated with **@Bindable**. If one exists the argument binder registry searches for an argument binder that supports that annotation.

If no fitting annotation is found Micronaut tries to find an argument binder that supports the argument type.

An argument binder returns a **ArgumentBinder.BindingResult**. The binding result gives Micronaut more information than just the value. Binding results are either satisfied or unsatisfied, and either empty or not empty. If an argument binder returns an unsatisfied result, the binder may be called again at different times in request processing. Argument binders are initially called before the body is read and before any filters are executed. If a binder relies on any of that data and it is not present, return a **ArgumentBinder.BindingResult#UNSATISFIED** result. Returning an **ArgumentBinder.BindingResult#EMPTY** or satisfied result will be the final result and the binder will not be called again for that request.

At the end of processing if the result is still **ArgumentBinder.BindingResult#UNSATISFIED**, it is considered **ArgumentBinder.BindingResult#EMPTY**.

Key interfaces are:

AnnotatedRequestArgumentBinder

Argument binders that bind based on the presence of an annotation must implement **AnnotatedRequestArgumentBinder**, and can be used by creating an annotation that is annotated with **@Bindable**. For example:

An example of a binding annotation

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.AliasFor;
import io.micronaut.core.bind.annotation.Bindable;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Target({FIELD, PARAMETER, ANNOTATION_TYPE})
@Retention(RUNTIME)
@Bindable  1
public @interface ShoppingCart {

    @AliasFor(annotation = Bindable.class, member = "value")
    String value() default "";
}
```

Copy to Clipboard

1 The binding annotation must itself be annotated as **@Bindable**

Example of annotated data binding

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.core.convert.ArgumentConversionContext;
import io.micronaut.core.convert.ConversionService;
import io.micronaut.core.type.Argument;
import io.micronaut.http.HttpRequest;
import io.micronaut.http.bind.binders.AnnotatedRequestArgumentBinder;
import io.micronaut.http.cookie.Cookie;
import io.micronaut.jackson.serialize.JacksonObjectSerializer;

import jakarta.inject.Singleton;
import java.util.Map;
import java.util.Optional;

@Singleton
public class ShoppingCartRequestArgumentBinder
        implements AnnotatedRequestArgumentBinder<ShoppingCart, Object> {  1

    private final ConversionService<?> conversionService;
    private final JacksonObjectSerializer objectSerializer;

    public ShoppingCartRequestArgumentBinder(ConversionService<?> conversionService,
                                             JacksonObjectSerializer objectSerializer) {
        this.conversionService = conversionService;
        this.objectSerializer = objectSerializer;
    }

    @Override
    public Class<ShoppingCart> getAnnotationType() {
        return ShoppingCart.class;
    }

    @Override
    public BindingResult<Object> bind(
            ArgumentConversionContext<Object> context,
            HttpRequest<?> source) {  2

        String parameterName = context.getAnnotationMetadata()
                .stringValue(ShoppingCart.class)
                .orElse(context.getArgument().getName());

        Cookie cookie = source.getCookies().get("shoppingCart");
        if (cookie == null) {
            return BindingResult.EMPTY;
        }

        Optional<Map<String, Object>> cookieValue = objectSerializer.deserialize(
                cookie.getValue().getBytes(),
                Argument.mapOf(String.class, Object.class));

        return () -> cookieValue.flatMap(map -> {
            Object obj = map.get(parameterName);
            return conversionService.convert(obj, context);
        });
    }
}
```

Copy to Clipboard

1 The custom argument binder must implement **AnnotatedRequestArgumentBinder**, including both the annotation type to trigger the binder (in this case, `MyBindingAnnotation`) and the type of the argument expected (in this case, `Object`)

2 Override the `bind` method with the custom argument binding logic - in this case, we resolve the name of the annotated argument, extract a value from a cookie with that same name, and convert that value to the argument type

It is common to use **ConversionService** to convert the data to the type of the argument.

Once the binder is created, we can annotate an argument in our controller method which will be bound using the custom logic we've specified.

A controller operation with this annotated binding

Java	Groovy	Kotlin

JAVA

```
    @Get("/annotated")
    HttpResponse<String> checkSession(@ShoppingCart Long sessionId) {  1
        return HttpResponse.ok("Session:" + sessionId);
    }
    // end::method
}
```

Copy to Clipboard

1 The parameter is bound with the binder associated with `MyBindingAnnotation`. This takes precedence over a type-based binder, if applicable.

TypedRequestArgumentBinder

Argument binders that bind based on the type of the argument must implement **TypedRequestArgumentBinder**. For example, given this class:

Example of POJO

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.core.annotation.Introspected;

@Introspected
public class ShoppingCart {

    private String sessionId;
    private Integer total;

    public String getSessionId() {
        return sessionId;
    }

    public void setSessionId(String sessionId) {
        this.sessionId = sessionId;
    }

    public Integer getTotal() {
        return total;
    }

    public void setTotal(Integer total) {
        this.total = total;
    }
}
```

Copy to Clipboard

We can define a `TypedRequestArgumentBinder` for this class, as seen below:

Example of typed data binding

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.core.convert.ArgumentConversionContext;
import io.micronaut.core.type.Argument;
import io.micronaut.http.HttpRequest;
import io.micronaut.http.bind.binders.TypedRequestArgumentBinder;
import io.micronaut.http.cookie.Cookie;
import io.micronaut.jackson.serialize.JacksonObjectSerializer;

import jakarta.inject.Singleton;
import java.util.Optional;

@Singleton
public class ShoppingCartRequestArgumentBinder
        implements TypedRequestArgumentBinder<ShoppingCart> {

    private final JacksonObjectSerializer objectSerializer;

    public ShoppingCartRequestArgumentBinder(JacksonObjectSerializer objectSerializer) {
        this.objectSerializer = objectSerializer;
    }

    @Override
    public BindingResult<ShoppingCart> bind(ArgumentConversionContext<ShoppingCart> context,
                                            HttpRequest<?> source) {  1

        Cookie cookie = source.getCookies().get("shoppingCart");
        if (cookie == null) {
            return Optional::empty;
        }

        return () -> objectSerializer.deserialize(  2
                cookie.getValue().getBytes(),
                ShoppingCart.class);
    }

    @Override
    public Argument<ShoppingCart> argumentType() {
        return Argument.of(ShoppingCart.class);  3
    }
}
```

Copy to Clipboard

1. Override the `bind` method with the data type to bind, in this example the `ShoppingCart` type
2. After retrieving the data (in this case, by deserializing JSON text from a cookie), return as a **ArgumentBinder.BindingResult**
3. Also override the `argumentType` method, which is used by the ArgumentBinderRegistry.

Once the binder is created, it is used for any controller argument of the associated type:

A controller operation with this typed binding

Java | Groovy | Kotlin

JAVA

```
    @Get("/typed")
    public HttpResponse<?> loadCart(ShoppingCart shoppingCart) {  1
        Map<String, Object> responseMap = new HashMap<>();
        responseMap.put("sessionId", shoppingCart.getSessionId());
        responseMap.put("total", shoppingCart.getTotal());

        return HttpResponse.ok(responseMap);
    }
```

Copy to Clipboard

1. The parameter is bound using the custom logic defined for this type in our `TypedRequestArgumentBinder`

6.6 Host Resolution

You may need to resolve the host name of the current server. Micronaut includes an implementation of the **HttpHostResolver** interface.

The default implementation looks for host information in the following places in order:

1. The supplied configuration
2. The `Forwarded` header
3. `X-Forwarded-` headers. If the `X-Forwarded-Host` header is not present, the other `X-Forwarded` headers are ignored.
4. The `Host` header
5. The properties on the request URI

6. The properties on the embedded server URI

The behavior of which headers to pull the relevant data can be changed with the following configuration:

Unresolved directive in <stdin> - include::/home/runner/work/micronaut-core/micronaut-core/build/generated/configurationProperties/io.micronaut.http.server.HttpServerConfiguration.HostResolutionConfiguration.adoc[]

The above configuration also supports an allowed host list. Configuring this list ensures any resolved host matches one of the supplied regular expression patterns. That is useful to prevent host cache poisoning attacks and is recommended to be configured.

6.7 Locale Resolution

Micronaut supports several strategies for resolving locales for a given request. The **getLocale--** method is available on the request, however it only supports parsing the `Accept-Language` header. For other use cases where the locale can be in a cookie, the user's session, or should be set to a fixed value, **HttpLocaleResolver** can be used to determine the current locale.

The **LocaleResolver** API does not need to be used directly. Simply define a parameter to a controller method of type `java.util.Locale` and the locale will be resolved and injected automatically.

There are several configuration options to control how to resolve the locale:

Unresolved directive in <stdin> - include::/home/runner/work/micronaut-core/micronaut-core/build/generated/configurationProperties/io.micronaut.http.server.HttpServerConfiguration.HttpLocaleResolutionConfigurationProperties.adoc[]

Locales can be configured in the "en_GB" format, or in the BCP 47 (Language tag) format. If multiple methods are configured, the fixed locale takes precedence, followed by session/cookie, then header.

If any of the built-in methods do not meet your use case, create a bean of type **HttpLocaleResolver** and set its order (through the `getOrder` method) relative to the existing resolvers.

6.8 Client IP Address

You may need to resolve the originating IP address of an HTTP Request. Micronaut includes an implementation of **HttpClientAddressResolver**.

The default implementation resolves the client address in the following places in order:

1. The configured header
2. The `Forwarded` header
3. The `X-Forwarded-For` header
4. The remote address on the request

The first priority header name can be configured with `micronaut.server.client-address-header`.

6.9 The HttpRequest and HttpResponse

If you need more control over request processing you can write a method that receives the complete **HttpRequest**.

In fact, there are several higher-level interfaces that can be bound to controller method parameters. These include:

Table 1. Bindable Micronaut Interfaces

Interface	Description	Example
HttpRequest	The full `HttpRequest`	`String hello(HttpRequest request)`
HttpHeaders	All HTTP headers present in the request	`String hello(HttpHeaders headers)`
HttpParameters	All HTTP parameters (either from URI variables or request parameters) present in the request	`String hello(HttpParameters params)`
Cookies	All Cookies present in the request	`String hello(Cookies cookies)`

The **HttpRequest** should be declared parametrized with a concrete generic type if the request body is needed, e.g. `HttpRequest<MyClass> request`. The body may not be available from the request otherwise.

In addition, for full control over the emitted HTTP response you can use the static factory methods of the **HttpResponse** class which return a **MutableHttpResponse**.

The following example implements the previous `MessageController` example using the **HttpRequest** and **HttpResponse** objects:

Request and Response Example

Java	Groovy	Kotlin

JAVA

```
import io.micronaut.http.HttpRequest;
import io.micronaut.http.HttpResponse;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.context.ServerRequestContext;
import reactor.core.publisher.Mono;

@Controller("/request")
public class MessageController {

    @Get("/hello") 1
    public HttpResponse<String> hello(HttpRequest<?> request) {
        String name = request.getParameters()
                             .getFirst("name")
                             .orElse("Nobody"); 2

        return HttpResponse.ok("Hello " + name + "!!")
                 .header("X-My-Header", "Foo"); 3
    }

}
```

Copy to Clipboard

1 The method is mapped to the URI `/hello` and accepts a **HttpRequest**

2 The **HttpRequest** is used to obtain the value of a query parameter named `name`.

3 The **HttpResponse.ok(T)** method returns a **MutableHttpResponse** with a text body. A header named `X-My-Header` is also added to the response.

The **HttpRequest** is also available from a static context via **ServerRequestContext**.

Using the ServerRequestContext

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.HttpRequest;
import io.micronaut.http.HttpResponse;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.context.ServerRequestContext;
import reactor.core.publisher.Mono;

@Controller("/request")
public class MessageController {

    @Get("/hello-static") 1
    public HttpResponse<String> helloStatic() {
        HttpRequest<?> request = ServerRequestContext.currentRequest() 1
                .orElseThrow(() -> new RuntimeException("No request present"));
        String name = request.getParameters()
                .getFirst("name")
                .orElse("Nobody");

        return HttpResponse.ok("Hello " + name + "!!")
                .header("X-My-Header", "Foo");
    }

}
```

Copy to Clipboard

1 The **ServerRequestContext** is used to retrieve the request.

Generally **ServerRequestContext** is available within reactive flow, but the recommended approach is consume the request as an argument as shown in the previous example. If the request is needed in downstream methods it should be passed as an argument to those methods. There are cases where the context is not propagated because other threads are used to emit the data.

An alternative for users of Project Reactor to using the ServerRequestContext is to use the contextual features of Project Reactor to retrieve the request. Because the Micronaut Framework uses Project Reactor as it's default reactive streams implementation, users of Project Reactor can benefit by being able to access the request in the context. For example:

Using the Project Reactor context

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.HttpRequest;
import io.micronaut.http.HttpResponse;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.context.ServerRequestContext;
import reactor.core.publisher.Mono;

@Controller("/request")
public class MessageController {

    @Get("/hello-reactor")
    public Mono<HttpResponse<String>> helloReactor() {
        return Mono.deferContextual(ctx -> {  1
            HttpRequest<?> request = ctx.get(ServerRequestContext.KEY);  2
            String name = request.getParameters()
                    .getFirst("name")
                    .orElse("Nobody");

            return Mono.just(HttpResponse.ok("Hello " + name + "!!")
                    .header("X-My-Header", "Foo"));
        });
    }

}
```

Copy to Clipboard

1 The Mono is created with a reference to the context

2 The request is retrieved from the context

Using the context to retrieve the request is the best approach for reactive flows because Project Reactor propagates the context and it does not rely on a thread local like [ServerRequestContext](#).

6.10 Response Status

A Micronaut controller action responds with a 200 HTTP status code by default.

If the action returns an `HttpResponse`, configure the status code for the response with the `status` method.

Java | Groovy | Kotlin

JAVA

```
    @Get(value = "/http-response", produces = MediaType.TEXT_PLAIN)
    public HttpResponse httpResponse() {
        return HttpResponse.status(HttpStatus.CREATED).body("success");
    }
```

Copy to Clipboard

You can also use the `@Status` annotation.

Java | Groovy | Kotlin

JAVA

```
    @Status(HttpStatus.CREATED)
    @Get(produces = MediaType.TEXT_PLAIN)
    public String index() {
        return "success";
    }
```

Copy to Clipboard

or even respond with an `HttpStatus`

Java | Groovy | Kotlin

JAVA

```
    @Get("/http-status")
    public HttpStatus httpStatus() {
        return HttpStatus.CREATED;
    }
```

Copy to Clipboard

6.11 Response Content-Type

A Micronaut controller action produces `application/json` by default. However you can change the `Content-Type` of the response with the `@Produces` annotation or the `produces` member of the HTTP method annotations.

Java | Groovy | Kotlin

JAVA

```java
import io.micronaut.context.annotation.Requires;
import io.micronaut.http.HttpResponse;
import io.micronaut.http.MediaType;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.annotation.Produces;

@Controller("/produces")
public class ProducesController {

    @Get  1
    public HttpResponse index() {
        return HttpResponse.ok().body("{\"msg\":\"This is JSON\"}");
    }

    @Produces(MediaType.TEXT_HTML)
    @Get("/html")  2
    public String html() {
        return "<html><title><h1>HTML</h1></title><body></body></html>";
    }

    @Get(value = "/xml", produces = MediaType.TEXT_XML)  3
    public String xml() {
        return "<html><title><h1>XML</h1></title><body></body></html>";
    }
}
```

Copy to Clipboard

1 The default content type is JSON

2 Annotate a controller action with `@Produces` to change the response content type.

3 Setting the `produces` member of the method annotation also changes the content type.

6.12 Accepted Request Content-Type

A Micronaut controller action consumes `application/json` by default. Consuming other content types is supported with the `@Consumes` annotation, or the `consumes` member of any HTTP method annotation.

Java | Groovy | Kotlin

JAVA

```java
import io.micronaut.context.annotation.Requires;
import io.micronaut.http.HttpResponse;
import io.micronaut.http.MediaType;
import io.micronaut.http.annotation.Consumes;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Post;

@Controller("/consumes")
public class ConsumesController {

    @Post  1
    public HttpResponse index() {
        return HttpResponse.ok();
    }

    @Consumes({MediaType.APPLICATION_FORM_URLENCODED, MediaType.APPLICATION_JSON})  2
    @Post("/multiple")
    public HttpResponse multipleConsumes() {
        return HttpResponse.ok();
    }

    @Post(value = "/member", consumes = MediaType.TEXT_PLAIN)  3
    public HttpResponse consumesMember() {
        return HttpResponse.ok();
    }
}
```

Copy to Clipboard

1 By default, a controller action consumes request with `Content-Type` of type `application/json`.

2 The `@Consumes` annotation takes a `String[]` of supported media types for an incoming request.

3 Content types can also be specified with the `consumes` member of the method annotation.

Customizing Processed Content Types

Normally JSON parsing only happens if the content type is `application/json`. The other **MediaTypeCodec** classes behave similarly in that they have predefined content types they can process. To extend the list of media types that a given codec processes, provide configuration that will be stored in **CodecConfiguration**:

YAML

```yaml
micronaut:
  codec:
    json:
      additionalTypes:
        - text/javascript
        - ...
```

The currently supported configuration prefixes are `json`, `json-stream`, `text`, and `text-stream`.

6.13 Reactive HTTP Request Processing

As mentioned previously, Micronaut is built on Netty which is designed around an Event loop model and non-blocking I/O. Micronaut executes code defined in **@Controller** beans in the same thread as the request thread (an Event Loop thread).

This makes it critical that if you do any blocking I/O operations (for example interactions with Hibernate/JPA or JDBC) that you offload those tasks to a separate thread pool that does not block the Event loop.

For example the following configuration configures the I/O thread pool as a fixed thread pool with 75 threads (similar to what a traditional blocking server such as Tomcat uses in the thread-per-request model):

Configuring the IO thread pool

YAML

```yaml
micronaut:
  executors:
    io:
      type: fixed
      nThreads: 75
```

To use this thread pool in a **@Controller** bean you have a number of options. The simplest is to use the **@ExecuteOn** annotation, which can be declared at the type or method level to indicate which configured thread pool to run the method(s) of the controller on:

Using @ExecuteOn

Java | Groovy | Kotlin

JAVA

```java
import io.micronaut.docs.http.server.reactive.PersonService;
import io.micronaut.docs.ioc.beans.Person;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.scheduling.TaskExecutors;
import io.micronaut.scheduling.annotation.ExecuteOn;

@Controller("/executeOn/people")
public class PersonController {

    private final PersonService personService;

    PersonController(PersonService personService) {
        this.personService = personService;
    }

    @Get("/{name}")
    @ExecuteOn(TaskExecutors.IO)  1
    Person byName(String name) {
        return personService.findByName(name);
    }
}
```

Copy to Clipboard

1 The **@ExecuteOn** annotation is used to execute the operation on the I/O thread pool

The value of the **@ExecuteOn** annotation can be any named executor defined under `micronaut.executors`.

Generally speaking for database operations you want a thread pool configured that matches the maximum number of connections specified in the database connection pool.

An alternative to the **@ExecuteOn** annotation is to use the facility provided by the reactive library you have chosen. Reactive implementations such as **Project Reactor** (https://projectreactor.io) or **RxJava** (https://github.com/ReactiveX/RxJava) feature a `subscribeOn` method which lets you alter which thread executes user code. For example:

Reactive subscribeOn Example

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.docs.ioc.beans.Person;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.scheduling.TaskExecutors;
import jakarta.inject.Named;
import org.reactivestreams.Publisher;
import reactor.core.publisher.Mono;
import reactor.core.scheduler.Scheduler;
import reactor.core.scheduler.Schedulers;
import io.micronaut.core.async.annotation.SingleResult;
import java.util.concurrent.ExecutorService;

@Controller("/subscribeOn/people")
public class PersonController {

    private final Scheduler scheduler;
    private final PersonService personService;

    PersonController(
            @Named(TaskExecutors.IO) ExecutorService executorService,  1
            PersonService personService) {
        this.scheduler = Schedulers.fromExecutorService(executorService);
        this.personService = personService;
    }

    @Get("/{name}")
    @SingleResult
    Publisher<Person> byName(String name) {
        return Mono
                .fromCallable(() -> personService.findByName(name))  2
                .subscribeOn(scheduler);  3
    }
}
```

Copy to Clipboard

1. The configured I/O executor service is injected
2. The `Mono::fromCallable` method wraps the blocking operation
3. The **Project Reactor** (https://projectreactor.io) `subscribeOn` method schedules the operation on the I/O thread pool

6.13.1 Using the @Body Annotation

To parse the request body, you first indicate to Micronaut which parameter receives the data with the **Body** annotation.

The following example implements a simple echo server that echoes the body sent in the request:

Using the @Body annotation

Java	Groovy	Kotlin

JAVA

```
import io.micronaut.http.HttpResponse;
import io.micronaut.http.MediaType;
import io.micronaut.http.annotation.Body;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Post;
import javax.validation.constraints.Size;

@Controller("/receive")
public class MessageController {

    @Post(value = "/echo", consumes = MediaType.TEXT_PLAIN)  1
    String echo(@Size(max = 1024) @Body String text) {  2
        return text;  3
    }

}
```

Copy to Clipboard

1. The **Post** annotation is used with a **MediaType** of `text/plain` (the default is `application/json`).
2. The **Body** annotation is used with a `javax.validation.constraints.Size` that limits the size of the body to at most 1MB. This constraint does **not** limit the amount of data read/buffered by the server.
3. The body is returned as the result of the method

Note that reading the request body is done in a non-blocking manner in that the request contents are read as the data becomes available and accumulated into the String passed to the method.

The `micronaut.server.maxRequestSize` setting in `application.yml` limits the size of the data (the default maximum request size is 10MB) read/buffered by the server. `@Size` is **not** a replacement for this setting.

Regardless of the limit, for a large amount of data accumulating the data into a String in-memory may lead to memory strain on the server. A better approach is to include a Reactive library in your project (such as `Reactor`, `RxJava`,or `Akka`) that supports the Reactive streams implementation and stream the data it becomes available:

Using Reactive Streams to Read the request body

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.HttpResponse;
import io.micronaut.http.MediaType;
import io.micronaut.http.annotation.Body;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Post;
import javax.validation.constraints.Size;


@Controller("/receive")
public class MessageController {

    @Post(value = "/echo-publisher", consumes = MediaType.TEXT_PLAIN)  1
    @SingleResult
    Publisher<HttpResponse<String>> echoFlow(@Body Publisher<String> text) {  2
        return Flux.from(text)
                .collect(StringBuffer::new, StringBuffer::append)  3
                .map(buffer -> HttpResponse.ok(buffer.toString()));
    }

}
```

Copy to Clipboard

1 In this case the method is altered to receive and return an `Publisher` type.

2 This example uses **Project Reactor** (https://projectreactor.io) and returns a single item. Because of that the response type is annotated also with **SingleResult**. Micronaut only emits the response once the operation completes without blocking.

3 The `collect` method is used to accumulate the data in this simulated example, but it could for example write the data to a logging service, database, etc. chunk by chunk

Body arguments of types that do not require conversion cause Micronaut to skip decoding of the request!

6.13.2 Reactive Responses

The previous section introduced the notion of Reactive programming using **Project Reactor** (https://projectreactor.io) and Micronaut.

Micronaut supports returning common reactive types such as **Mono** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html) (or **Single** (http://reactivex.io/RxJava/2.x/javadoc/io/reactivex/Single.html) **Maybe** (http://reactivex.io/RxJava/2.x/javadoc/io/reactivex/Maybe.html) **Observable** (http://reactivex.io/RxJava/2.x/javadoc/io/reactivex/Observable.html) types from RxJava), an instance of **Publisher** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) or **CompletableFuture** (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html) from any controller method.

To use **Project Reactor** (https://projectreactor.io)'s `Flux` or `Mono` you need to add the Micronaut Reactor dependency to your project to include the necessary converters.

To use **RxJava** (https://github.com/ReactiveX/RxJava)'s `Flowable`, `Single` or `Maybe` you need to add the Micronaut RxJava dependency to your project to include the necessary converters.

The argument designated as the body of the request using the **Body** annotation can also be a reactive type or a **CompletableFuture** (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html).

When returning a reactive type, Micronaut subscribes to the returned reactive type on the same thread as the request (a Netty Event Loop thread). It is therefore important that if you perform any blocking operations, you offload those operations to an appropriately configured thread pool, for example using the **Project Reactor** (https://projectreactor.io) or **RxJava** (https://github.com/ReactiveX/RxJava) `subscribeOn(..)` facility or **@ExecuteOn**.

See the section on **Configuring Thread Pools** for information on the thread pools that Micronaut sets up and how to configure them.

To summarize, the following table illustrates some common response types and their handling:

Table 1. Micronaut Response Types

Type	Description	Example Signature
Publisher (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html)	Any type that implements the Publisher (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) interface	`Publisher<String> hello()`
CompletableFuture (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html)	A Java `CompletableFuture` instance	`CompletableFuture<String> hello()`
HttpResponse	An HttpResponse and optional response body	`HttpResponse<Publisher<String>> hello()`
CharSequence (https://docs.oracle.com/javase/8/docs/api/java/lang/CharSequence.html)	Any implementation of `CharSequence`	`String hello()`
T	Any simple POJO type	`Book show()`

When returning a Reactive type, its type affects the returned response. For example, when returning a Flux (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html), Micronaut cannot know the size of the response, so `Transfer-Encoding` type of `Chunked` is used. Whilst for types that emit a single result such as Mono (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html) the `Content-Length` header is populated.

6.14 JSON Binding with Jackson

The most common data interchange format nowadays is JSON.

In fact, the defaults in the Controller annotation specify that the controllers in Micronaut consume and produce JSON by default.

To do so in a non-blocking manner, Micronaut builds on the Jackson (https://github.com/FasterXML/jackson) Asynchronous JSON parsing API and Netty, such that the reading of incoming JSON is done in a non-blocking manner.

Binding using Reactive Frameworks

From a developer perspective however, you can generally just work with Plain Old Java Objects (POJOs) and can optionally use a Reactive framework such as RxJava (https://github.com/ReactiveX/RxJava) or Project Reactor (https://projectreactor.io). The following is an example of a controller that reads and saves an incoming POJO in a non-blocking way from JSON:

Using Reactive Streams to Read the JSON

Java | Groovy | Kotlin

JAVA

```
@Controller("/people")
public class PersonController {

    Map<String, Person> inMemoryDatastore = new ConcurrentHashMap<>();

    @Post("/saveReactive")
    @SingleResult
    public Publisher<HttpResponse<Person>> save(@Body Publisher<Person> person) {  1
        return Mono.from(person).map(p -> {
                    inMemoryDatastore.put(p.getFirstName(), p);  2
                    return HttpResponse.created(p);  3
                }
        );
    }

}
```

Copy to Clipboard

1. The method receives a `Publisher` which emits the POJO once the JSON has been read
2. The `map` method stores the instance in a `Map`
3. An HttpResponse is returned

Using cURL from the command line, you can POST JSON to the `/people` URI:

Using cURL to Post JSON

```
$ curl -X POST localhost:8080/people -d '{"firstName":"Fred","lastName":"Flintstone","age":45}'
```

Binding Using CompletableFuture

The same method as the previous example can also be written with the CompletableFuture (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html) API instead:

Using CompletableFuture to Read the JSON

Java | Groovy | Kotlin

JAVA

```
@Controller("/people")
public class PersonController {

    Map<String, Person> inMemoryDatastore = new ConcurrentHashMap<>();

    @Post("/saveFuture")
    public CompletableFuture<HttpResponse<Person>> save(@Body CompletableFuture<Person> person) {
        return person.thenApply(p -> {
                    inMemoryDatastore.put(p.getFirstName(), p);
                    return HttpResponse.created(p);
                }
        );
    }

}
```

Copy to Clipboard

The above example uses the `thenApply` method to achieve the same as the previous example.

Binding using POJOs

Note however you can just as easily write:

Binding JSON POJOs

Java | Groovy | Kotlin

JAVA

```
@Controller("/people")
public class PersonController {

    Map<String, Person> inMemoryDatastore = new ConcurrentHashMap<>();

    @Post
    public HttpResponse<Person> save(@Body Person person) {
        inMemoryDatastore.put(person.getFirstName(), person);
        return HttpResponse.created(person);
    }

}
```

Copy to Clipboard

Micronaut only executes your method once the data has been read in a non-blocking manner.

The output produced by Jackson can be customized in a variety of ways, from defining Jackson modules to using Jackson annotations (https://github.com/FasterXML/jackson-annotations/wiki/Jackson-Annotations)

Jackson Configuration

The Jackson `ObjectMapper` can be configured through configuration with the JacksonConfiguration class.

All Jackson configuration keys start with `jackson.`

dateFormat	String	The date format
locale	String	Uses Locale.forLanguageTag (https://docs.oracle.com/javase/8/docs/api/java/util/Locale.html#forLanguageTag-java.lang.String-) . Example: `en-US`
timeZone	String	Uses TimeZone.getTimeZone (https://docs.oracle.com/javase/8/docs/api/java/util/TimeZone.html#getTimeZone-java.lang.String-) . Example: `PST`
serializationInclusion	String	One of JsonInclude.Include (http://fasterxml.github.io/jackson-annotations/javadoc/2.9/com/fasterxml/jackson/annotation/JsonInclude.Include.html) . Example: `ALWAYS`

propertyNamingStrategy	String	Name of an instance of PropertyNamingStrategy (http://fasterxml.github.io/jackson-databind/javadoc/2.9/com/fasterxml/jackson/databind/PropertyNamingStrategy.html). Example: `SNAKE_CASE`
defaultTyping	String	The global defaultTyping for polymorphic type handling from enum ObjectMapper.DefaultTyping (http://fasterxml.github.io/jackson-databind/javadoc/2.9/com/fasterxml/jackson/databind/ObjectMapper.DefaultTyping.html). Example: `NON_FINAL`

Example:

```yaml
jackson:
  serializationInclusion: ALWAYS
```

Features

All features can be configured with their name as the key and a boolean to indicate enabled or disabled.

serialization	Map	SerializationFeature (http://fasterxml.github.io/jackson-databind/javadoc/2.9/com/fasterxml/jackson/databind/SerializationFeature.html)
deserialization	Map	DeserializationFeature (http://fasterxml.github.io/jackson-databind/javadoc/2.9/com/fasterxml/jackson/databind/DeserializationFeature.html)
mapper	Map	MapperFeature (http://fasterxml.github.io/jackson-databind/javadoc/2.9/com/fasterxml/jackson/databind/MapperFeature.html)
parser	Map	JsonParser.Feature (http://fasterxml.github.io/jackson-core/javadoc/2.9/com/fasterxml/jackson/core/JsonParser.Feature.html)
generator	Map	JsonGenerator.Feature (http://fasterxml.github.io/jackson-core/javadoc/2.9/com/fasterxml/jackson/core/JsonGenerator.Feature.html)

Example:

```yaml
jackson:
  serialization:
    indentOutput: true
    writeDatesAsTimestamps: false
  deserialization:
    useBigIntegerForInts: true
    failOnUnknownProperties: false
```

Support for `@JsonView`

You can use the `@JsonView` annotation on controller methods if you set `jackson.json-view.enabled` to `true` in `application.yml`.

Jackson's `@JsonView` annotation lets you control which properties are exposed on a per-response basis. See Jackson JSON Views (https://www.baeldung.com/jackson-json-view-annotation) for more information.

Beans

In addition to configuration, beans can be registered to customize Jackson. All beans that extend any of the following classes are registered with the object mapper:

- Module (http://fasterxml.github.io/jackson-databind/javadoc/2.9/com/fasterxml/jackson/databind/Module.html)
- JsonDeserializer (http://fasterxml.github.io/jackson-databind/javadoc/2.9/com/fasterxml/jackson/databind/JsonDeserializer.html)
- JsonSerializer (http://fasterxml.github.io/jackson-databind/javadoc/2.9/com/fasterxml/jackson/databind/JsonSerializer.html)
- KeyDeserializer (http://fasterxml.github.io/jackson-databind/javadoc/2.9/com/fasterxml/jackson/databind/KeyDeserializer.html)
- BeanDeserializerModifier (http://fasterxml.github.io/jackson-databind/javadoc/2.9/com/fasterxml/jackson/databind/deser/BeanDeserializerModifier.html)
- BeanSerializerModifier (http://fasterxml.github.io/jackson-databind/javadoc/2.9/com/fasterxml/jackson/databind/ser/BeanSerializerModifier.html)

Service Loader

Any modules registered via the service loader are also added to the default object mapper.

6.15 Data Validation

It is easy to validate incoming data with Micronaut controllers using **Validation Advice**.

Micronaut provides native support for the `javax.validation` annotations with the `micronaut-validation` dependency:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut</groupId>
    <artifactId>micronaut-validation</artifactId>
</dependency>
```

Copy to Clipboard

Or full JSR 380 compliance with the `micronaut-hibernate-validator` dependency:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.beanvalidation</groupId>
    <artifactId>micronaut-hibernate-validator</artifactId>
</dependency>
```

Copy to Clipboard

We can validate parameters using `javax.validation` annotations and the **Validated** annotation at the class level.

Example

Java Groovy Kotlin

JAVA

```
import io.micronaut.http.HttpResponse;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.validation.Validated;

import javax.validation.constraints.NotBlank;
import java.util.Collections;

@Validated  1
@Controller("/email")
public class EmailController {

    @Get("/send")
    public HttpResponse send(@NotBlank String recipient,  2
                             @NotBlank String subject) {  2
        return HttpResponse.ok(Collections.singletonMap("msg", "OK"));
    }
}
```

Copy to Clipboard

1 Annotate controller with **Validated**

2 `subject` and `recipient` cannot be blank.

If a validation error occurs a `javax.validation.ConstraintViolationException` is thrown. By default, the integrated `io.micronaut.validation.exception.ConstraintExceptionHandler` handles the exception, leading to a behaviour as shown in the following test:

Example Test

Java Groovy Kotlin

JAVA

```
    @Test
    public void testParametersAreValidated() {
        HttpClientResponseException e = Assertions.assertThrows(HttpClientResponseException.class, () ->
            client.toBlocking().exchange("/email/send?subject=Hi&recipient="));
        HttpResponse<?> response = e.getResponse();

        assertEquals(HttpStatus.BAD_REQUEST, response.getStatus());

        response = client.toBlocking().exchange("/email/send?subject=Hi&recipient=me@micronaut.example");

        assertEquals(HttpStatus.OK, response.getStatus());
    }
```

Copy to Clipboard

To use your own `ExceptionHandler` to handle the constraint exceptions, annotate it with `@Replaces(ConstraintExceptionHandler.class)`

Often you may want to use POJOs as controller method parameters.

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.core.annotation.Introspected;

import javax.validation.constraints.NotBlank;

@Introspected
public class Email {

    @NotBlank  1
    String subject;

    @NotBlank  1
    String recipient;

    public String getSubject() {
        return subject;
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }

    public String getRecipient() {
        return recipient;
    }

    public void setRecipient(String recipient) {
        this.recipient = recipient;
    }
}
```

Copy to Clipboard

1 You can use `javax.validation` annotations in your POJOs.

Annotate your controller with **Validated**, and annotate the binding POJO with `@Valid`.

Example

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.HttpResponse;
import io.micronaut.http.annotation.Body;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Post;
import io.micronaut.validation.Validated;

import javax.validation.Valid;
import java.util.Collections;

@Validated  1
@Controller("/email")
public class EmailController {

    @Post("/send")
    public HttpResponse send(@Body @Valid Email email) {  2
        return HttpResponse.ok(Collections.singletonMap("msg", "OK"));
    }
}
```

Copy to Clipboard

1 Annotate the controller with **Validated**

2 Annotate the POJO to validate with `@Valid`

Validation of POJOs is shown in the following test:

Java | Groovy | Kotlin

JAVA

```
@Test
public void testPojoValidation() {
    HttpClientResponseException e = assertThrows(HttpClientResponseException.class, () -> {
        Email email = new Email();
        email.subject = "Hi";
        email.recipient = "";
        client.toBlocking().exchange(HttpRequest.POST("/email/send", email));
    });
    HttpResponse<?> response = e.getResponse();

    assertEquals(HttpStatus.BAD_REQUEST, response.getStatus());

    Email email = new Email();
    email.subject = "Hi";
    email.recipient = "me@micronaut.example";
    response = client.toBlocking().exchange(HttpRequest.POST("/email/send", email));

    assertEquals(HttpStatus.OK, response.getStatus());
}
```

Copy to Clipboard

Bean injection is supported in custom constraints with the Hibernate Validator configuration.

6.16 Serving Static Resources

Static resource resolution is enabled by default. Micronaut supports resolving resources from the classpath or the file system.

See the information below for available configuration options:

Unresolved directive in <stdin> - include::/home/runner/work/micronaut-core/micronaut-core/build/generated/configurationProperties/io.micronaut.web.router.resource.StaticResourceConfiguration.adoc[]

6.17 Error Handling

Sometimes with distributed applications, bad things happen. Having a good way to handle errors is important.

6.17.1 Status Handlers

The **@Error** annotation supports defining either an exception class or an HTTP status. Methods annotated with **@Error** must be defined with in a class annotated with **@Controller**. The annotation also supports the notion of global and local, local being the default.

Local error handlers only respond to exceptions thrown as a result of the route being matched to another method in the same controller. Global error handlers can be invoked as a result of any thrown exception. A local error handler is always searched for first when resolving which handler to execute.

When defining an error handler for an exception, you can specify the exception instance as an argument to the method and omit the exception property of the annotation.

6.17.2 Local Error Handling

For example, the following method handles JSON parse exceptions from Jackson for the scope of the declaring controller:

Local exception handler

Java | Groovy | Kotlin

JAVA

```
@Error
public HttpResponse<JsonError> jsonError(HttpRequest request, JsonParseException e) {  1
    JsonError error = new JsonError("Invalid JSON: " + e.getMessage())  2
            .link(Link.SELF, Link.of(request.getUri()));

    return HttpResponse.<JsonError>status(HttpStatus.BAD_REQUEST, "Fix Your JSON")
            .body(error);  3
}
```

Copy to Clipboard

1. A method that explicitly handles `JsonParseException` is declared
2. An instance of **JsonError** is returned.
3. A custom response is returned to handle the error

Local status handler

Java | Groovy | Kotlin

JAVA

```
@Error(status = HttpStatus.NOT_FOUND)
public HttpResponse<JsonError> notFound(HttpRequest request) {  1
    JsonError error = new JsonError("Person Not Found")  2
            .link(Link.SELF, Link.of(request.getUri()));

    return HttpResponse.<JsonError>notFound()
            .body(error);  3
}
```

Copy to Clipboard

1 The **Error** declares which **HttpStatus** error code to handle (in this case 404)

2 A **JsonError** instance is returned for all 404 responses

3 An **NOT_FOUND** response is returned

6.17.3 Global Error Handling

Global error handler

Java | Groovy | Kotlin

JAVA

```
@Error(global = true)  1
public HttpResponse<JsonError> error(HttpRequest request, Throwable e) {
    JsonError error = new JsonError("Bad Things Happened: " + e.getMessage())  2
            .link(Link.SELF, Link.of(request.getUri()));

    return HttpResponse.<JsonError>serverError()
            .body(error);  3
}
```

Copy to Clipboard

1 The **@Error** declares the method a global error handler

2 A **JsonError** instance is returned for all errors

3 An **INTERNAL_SERVER_ERROR** response is returned

Global status handler

Java | Groovy | Kotlin

JAVA

```
@Error(status = HttpStatus.NOT_FOUND)
public HttpResponse<JsonError> notFound(HttpRequest request) {  1
    JsonError error = new JsonError("Person Not Found")  2
            .link(Link.SELF, Link.of(request.getUri()));

    return HttpResponse.<JsonError>notFound()
            .body(error);  3
}
```

Copy to Clipboard

1 The **@Error** declares which **HttpStatus** error code to handle (in this case 404)

2 A **JsonError** instance is returned for all 404 responses

3 An **NOT_FOUND** response is returned

A few things to note about the **@Error** annotation. You cannot declare identical global `@Error` annotations. Identical non-global `@Error` annotations cannot be declared in the same controller. If an `@Error` annotation with the same parameter exists as global and another as local, the local one takes precedence.

6.17.4 ExceptionHandler

Alternatively, you can implement an **ExceptionHandler**, a generic hook for handling exceptions that occur during execution of an HTTP request.

An `@Error` annotation capturing an exception has precedence over an implementation of `ExceptionHandler` capturing the same exception.

6.17.4.1 Built-In Exception Handlers

Micronaut ships with several built-in handlers:

Exception	Handler
`javax.validation.ConstraintViolationException`	**ConstraintExceptionHandler**
ContentLengthExceededException	**ContentLengthExceededHandler**

ConversionErrorException	ConversionErrorHandler
DuplicateRouteException	DuplicateRouteHandler
HttpStatusException	HttpStatusHandler
com.fasterxml.jackson.core.JsonProcessingException	JsonExceptionHandler
java.net.URISyntaxException	URISyntaxHandler
UnsatisfiedArgumentException	UnsatisfiedArgumentHandler
UnsatisfiedRouteException	UnsatisfiedRouteHandler
org.grails.datastore.mapping.validation.ValidationException	ValidationExceptionHandler

6.17.4.2 Custom Exception Handler

Imagine your e-commerce app throws an `OutOfStockException` when a book is out of stock:

Java | Groovy | Kotlin

JAVA

```
public class OutOfStockException extends RuntimeException {
}
```

Copy to Clipboard

Along with `BookController`:

Java | Groovy | Kotlin

JAVA

```
@Controller("/books")
public class BookController {

    @Produces(MediaType.TEXT_PLAIN)
    @Get("/stock/{isbn}")
    Integer stock(String isbn) {
        throw new OutOfStockException();
    }
}
```

Copy to Clipboard

The server returns a 500 (Internal Server Error) status code if you don't handle the exception.

To respond with 400 Bad Request as the response when the `OutOfStockException` is thrown, you can register a `ExceptionHandler`:

Java | Groovy | Kotlin

JAVA

```
@Produces
@Singleton
@Requires(classes = {OutOfStockException.class, ExceptionHandler.class})
public class OutOfStockExceptionHandler implements ExceptionHandler<OutOfStockException, HttpResponse> {

    private final ErrorResponseProcessor<?> errorResponseProcessor;

    public OutOfStockExceptionHandler(ErrorResponseProcessor<?> errorResponseProcessor) {
        this.errorResponseProcessor = errorResponseProcessor;
    }

    @Override
    public HttpResponse handle(HttpRequest request, OutOfStockException e) {
        return errorResponseProcessor.processResponse(ErrorContext.builder(request)
                .cause(e)
                .errorMessage("No stock available")
                .build(), HttpResponse.badRequest()); 1
    }
}
```

Copy to Clipboard

1 The default **ErrorResponseProcessor** is used to create the body of the response

6.17.5 Error Formatting

Micronaut produces error response bodies via beans of type **ErrorResponseProcessor**.

The default response body is **vnd.error** (https://github.com/blongden/vnd.error), however you can create your own implementation of type **ErrorResponseProcessor** to control the responses.

If customization of the response other than items related to the errors is desired, the exception handler that is handling the exception needs to be overridden.

6.18 API Versioning

Since 1.1.x, Micronaut supports API versioning via a dedicated **@Version** annotation.

The following example demonstrates how to version an API:

Versioning an API

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.core.version.annotation.Version;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;

@Controller("/versioned")
class VersionedController {

    @Version("1")  1
    @Get("/hello")
    String helloV1() {
        return "helloV1";
    }

    @Version("2")  2
    @Get("/hello")
    String helloV2() {
        return "helloV2";
    }
```

Copy to Clipboard

1 The `helloV1` method is declared as version `1`

2 The `helloV2` method is declared as version `2`

Then enable versioning by setting `micronaut.router.versioning.enabled` to `true` in `application.yml`:

Enabling Versioning

YAML

```
micronaut:
  router:
    versioning:
      enabled: true
```

By default Micronaut has two strategies for resolving the version based on an HTTP header named `X-API-VERSION` or a request parameter named `api-version`, however this is configurable. A full configuration example can be seen below:

Configuring Versioning

YAML

```
micronaut:
  router:
    versioning:
      enabled: true  1
      parameter:
        enabled: false  2
        names: 'v,api-version'  3
      header:
        enabled: true  4
        names:  5
          - 'X-API-VERSION'
          - 'Accept-Version'
```

1 Enables versioning

2 Enables or disables parameter-based versioning

3 Specify the parameter names as a comma-separated list

4 Enables or disables header-based versioning

5 Specify the header names as a list

If this is not enough you can also implement the **RequestVersionResolver** interface which receives the **HttpRequest** and can implement any strategy you choose.

Default Version

It is possible to supply a default version through configuration.

Configuring Default Version

YAML

```yaml
micronaut:
  router:
    versioning:
      enabled: true
      default-version: 3  1
```

1 Sets the default version

A route is **not** matched if the following conditions are met:

- The default version is configured
- No version is found in the request
- The route defines a version
- The route version does not match the default version

If the incoming request specifies a version, the default version has no effect.

Versioning Client Requests

Micronaut's **Declarative HTTP client** also supports automatic versioning of outgoing requests via the **@Version** annotation.

By default, if you annotate a client interface with **@Version**, the value supplied to the annotation is included using the `X-API-VERSION` header.

For example:

Java | Groovy | Kotlin

JAVA

```java
import io.micronaut.core.version.annotation.Version;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.client.annotation.Client;
import org.reactivestreams.Publisher;
import io.micronaut.core.async.annotation.SingleResult;

@Client("/hello")
@Version("1")  1
public interface HelloClient {

    @Get("/greeting/{name}")
    String sayHello(String name);

    @Version("2")
    @Get("/greeting/{name}")
    @SingleResult
    Publisher<String> sayHelloTwo(String name);  2
}
```

Copy to Clipboard

1 The **@Version** annotation can be used at the type level to specify the version to use for all methods

2 When defined at the method level it is used only for that method

The default behaviour for how the version is sent for each call can be configured with **DefaultClientVersioningConfiguration**:

Unresolved directive in <stdin> - include::/home/runner/work/micronaut-core/micronaut-core/build/generated/configurationProperties/io.micronaut.http.client.interceptor.configuration.DefaultClientVersioningConfiguration.adoc[]

For example to use `Accept-Version` as the header name:

Configuring Client Versioning

YAML

```yaml
micronaut:
  http:
    client:
      versioning:
        default:
          headers:
            - 'Accept-Version'
            - 'X-API-VERSION'
```

The `default` key refers to the default configuration. You can specify client-specific configuration by using the value passed to `@Client` (typically the service ID). For example:

Configuring Versioning

YAML

```yaml
micronaut:
  http:
    client:
      versioning:
        greeting-service:
          headers:
            - 'Accept-Version'
            - 'X-API-VERSION'
```

The above uses a key named `greeting-service` which can be used to configure a client annotated with `@Client('greeting-service')`.

6.19 Handling Form Data

To make data binding model customizations consistent between form data and JSON, Micronaut uses Jackson to implement binding data from form submissions.

The advantage of this approach is that the same Jackson annotations used for customizing JSON binding can be used for form submissions.

In practice this means that to bind regular form data, the only change required to the previous JSON binding code is updating the **MediaType** consumed:

Binding Form Data to POJOs

Java | Groovy | Kotlin

JAVA

```java
@Controller("/people")
public class PersonController {

    Map<String, Person> inMemoryDatastore = new ConcurrentHashMap<>();

    @Post
    public HttpResponse<Person> save(@Body Person person) {
        inMemoryDatastore.put(person.getFirstName(), person);
        return HttpResponse.created(person);
    }

}
```

Copy to Clipboard

To avoid denial of service attacks, collection types and arrays created during binding are limited by the setting `jackson.arraySizeThreshold` in `application.yml`

Alternatively, instead of using a POJO you can bind form data directly to method parameters (which works with JSON too!):

Binding Form Data to Parameters

Java | Groovy | Kotlin

JAVA

```java
@Controller("/people")
public class PersonController {

    Map<String, Person> inMemoryDatastore = new ConcurrentHashMap<>();

    @Post("/saveWithArgs")
    public HttpResponse<Person> save(String firstName, String lastName, Optional<Integer> age) {
        Person p = new Person(firstName, lastName);
        age.ifPresent(p::setAge);
        inMemoryDatastore.put(p.getFirstName(), p);
        return HttpResponse.created(p);
    }

}
```

Copy to Clipboard

As you can see from the example above, this approach lets you use features such as support for **Optional** (https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html) types and restrict the parameters to be bound. When using POJOs you must be careful to use Jackson annotations to exclude properties that should not be bound.

6.20 Writing Response Data

Reactively Writing Response Data

Micronaut's HTTP server supports writing chunks of response data by returning a **Publisher** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) that emits objects that can be encoded to the HTTP response.

The following table summarizes example return type signatures and the behaviour the server exhibits to handle them:

Return Type	Description
`Publisher<String>`	A **Publisher** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) that emits each chunk of content as a String
`Flux<byte[]>`	A **Flux** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html) that emits each chunk of content as a `byte[]` without blocking
`Flux<ByteBuf>`	A Reactor `Flux` that emits each chunk as a Netty `ByteBuf`
`Flux<Book>`	When emitting a POJO, each emitted object is encoded as JSON by default without blocking
`Flowable<byte[]>`	A **Flux** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html) that emits each chunk of content as a `byte[]` without blocking
`Flowable<ByteBuf>`	A Reactor `Flux` that emits each chunk as a Netty `ByteBuf`
`Flowable<Book>`	When emitting a POJO, each emitted object is encoded as JSON by default without blocking

When returning a reactive type, the server uses a `Transfer-Encoding` of `chunked` and keeps writing data until the **Publisher** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) `onComplete` method is called.

The server requests a single item from the **Publisher** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html), writes it, and requests the next, controlling back pressure.

It is up to the implementation of the **Publisher** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) to schedule any blocking I/O work that may be done as a result of subscribing to the publisher.

To use **Project Reactor** (https://projectreactor.io)'s `Flux` or `Mono` you need to add the Micronaut Reactor dependency to your project to include the necessary converters.

To use **RxJava** (https://github.com/ReactiveX/RxJava)'s `Flowable`, `Single` or `Maybe` you need to add the Micronaut RxJava dependency to your project to include the necessary converters.

Performing Blocking I/O

In some cases you may wish to integrate a library that does not support non-blocking I/O.

Writable

In this case you can return a **Writable** object from any controller method. The **Writable** interface has various signatures that allow writing to traditional blocking streams like **Writer** (https://docs.oracle.com/javase/8/docs/api/java/io/Writer.html) or **OutputStream** (https://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html).

When returning a **Writable**, the blocking I/O operation is shifted to the I/O thread pool so the Netty event loop is not blocked.

See the section on configuring **Server Thread Pools** for details on how to configure the I/O thread pool to meet your application requirements.

The following example demonstrates how to use this API with Groovy's `SimpleTemplateEngine` to write a server side template:

Performing Blocking I/O With Writable

Java | Groovy | Kotlin

JAVA

```
import groovy.text.SimpleTemplateEngine;
import groovy.text.Template;
import io.micronaut.core.io.Writable;
import io.micronaut.core.util.CollectionUtils;
import io.micronaut.http.MediaType;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.server.exceptions.HttpServerException;

@Controller("/template")
public class TemplateController {

    private final SimpleTemplateEngine templateEngine = new SimpleTemplateEngine();
    private final Template template = initTemplate();  1

    @Get(value = "/welcome", produces = MediaType.TEXT_PLAIN)
    Writable render() {  2
        return writer -> template.make(  3
            CollectionUtils.mapOf(
                    "firstName", "Fred",
                    "lastName", "Flintstone"
            )
        ).writeTo(writer);
    }

    private Template initTemplate() {
        try {
            return templateEngine.createTemplate(
                    "Dear $firstName $lastName. Nice to meet you."
            );
        } catch (Exception e) {
            throw new HttpServerException("Cannot create template");
        }
    }
}
```

Copy to Clipboard

1 The controller creates a simple template

2 The controller method returns a **Writable**

3 The returned function receives a **Writer** (https://docs.oracle.com/javase/8/docs/api/java/io/Writer.html) and calls `writeTo` on the template.

InputStream

Another option is to return an input stream. This is useful for many scenarios that interact with other APIs that expose a stream.

Performing Blocking I/O With InputStream

Java | Groovy | Kotlin

JAVA

```
    @Get(value = "/write", produces = MediaType.TEXT_PLAIN)
    InputStream write() {
        byte[] bytes = "test".getBytes(StandardCharsets.UTF_8);
        return new ByteArrayInputStream(bytes);  1
    }
```

Copy to Clipboard

1 The input stream is returned and its contents will be the response body

The reading of the stream will be offloaded to the IO thread pool if the controller method is executed on the event loop.

404 Responses

Often, you want to respond 404 (Not Found) when you don't find an item in your persistence layer or in similar scenarios.

See the following example:

Java | Groovy | Kotlin

JAVA

```
@Controller("/books")
public class BooksController {

    @Get("/stock/{isbn}")
    public Map stock(String isbn) {
        return null;  1
    }

    @Get("/maybestock/{isbn}")
    @SingleResult
    public Publisher<Map> maybestock(String isbn) {
        return Mono.empty();  2
    }
}
```

Copy to Clipboard

1 Returning `null` triggers a 404 (Not Found) response.

2 Returning an empty `Mono` triggers a 404 (Not Found) response.

Responding with an empty `Publisher` or `Flux` results in an empty array being returned if the content type is JSON.

6.21 File Uploads

Handling of file uploads has special treatment in Micronaut. Support is provided for streaming of uploads in a non-blocking manner through streaming uploads or completed uploads.

To receive data from a multipart request, set the `consumes` argument of the method annotation to **MULTIPART_FORM_DATA**. For example:

JAVA

```
@Post(consumes = MediaType.MULTIPART_FORM_DATA)
HttpResponse upload( ... )
```

Route Arguments

Method argument types determine how files are received. Data can be received a chunk at a time or when an upload is completed.

If the route argument name cannot or should not match the name of the part in the request, add the **@Part** annotation to the argument and specify the expected name in the request.

Chunk Data Types

PartData represents a chunk of data received in a multipart request. **PartData** interface methods convert the data to a `byte[]`, **InputStream** (https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html), or a **ByteBuffer** (https://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html).

Data can only be retrieved from a **PartData** once. The underlying buffer is released, causing further attempts to fail.

Route arguments of type **Publisher<PartData>** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) are treated as intended to receive a single file, and each chunk of the received file will be sent downstream. If the generic type is other than **PartData**, conversion will be attempted using Micronaut's conversion service. Conversions to `String` and `byte[]` are supported by default.

If you need knowledge about the metadata of an uploaded file, the **StreamingFileUpload** class is a **Publisher<PartData>** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) that also has file information such as the content type and file name.

Streaming file upload

Java	Groovy	Kotlin

JAVA

```java
import io.micronaut.http.HttpResponse;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Post;
import io.micronaut.http.multipart.StreamingFileUpload;
import org.reactivestreams.Publisher;
import reactor.core.publisher.Mono;
import io.micronaut.core.async.annotation.SingleResult;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.IOException;
import java.io.OutputStream;
import java.nio.file.Files;

import static io.micronaut.http.HttpStatus.CONFLICT;
import static io.micronaut.http.MediaType.MULTIPART_FORM_DATA;
import static io.micronaut.http.MediaType.TEXT_PLAIN;

@Controller("/upload")
public class UploadController {

    @Post(value = "/", consumes = MULTIPART_FORM_DATA, produces = TEXT_PLAIN)  1
    @SingleResult
    public Publisher<HttpResponse<String>> upload(StreamingFileUpload file) {  2

        File tempFile;
        try {
            tempFile = File.createTempFile(file.getFilename(), "temp");
        } catch (IOException e) {
            return Mono.error(e);
        }
        Publisher<Boolean> uploadPublisher = file.transferTo(tempFile);  3

        return Mono.from(uploadPublisher)  4
            .map(success -> {
                if (success) {
                    return HttpResponse.ok("Uploaded");
                } else {
                    return HttpResponse.<String>status(CONFLICT)
                                       .body("Upload Failed");
                }
            });
    }

    @Post(value = "/outputStream", consumes = MULTIPART_FORM_DATA, produces = TEXT_PLAIN)  1
    @SingleResult
    public Mono<HttpResponse<String>> uploadOutputStream(StreamingFileUpload file) {  2

        OutputStream outputStream = new ByteArrayOutputStream();  3

        Publisher<Boolean> uploadPublisher = file.transferTo(outputStream);  4

        return Mono.from(uploadPublisher)  5
                .map(success -> {
                    if (success) {
                        return HttpResponse.ok("Uploaded");
                    } else {
                        return HttpResponse.<String>status(CONFLICT)
                                .body("Upload Failed");
                    }
                });
    }
}
```

Copy to Clipboard

1. The method consumes **MULTIPART_FORM_DATA**
2. The method parameters match form attribute names. In this case `file` will match for example an `<input type="file" name="file">`
3. The **StreamingFileUpload.transferTo(File)** method transfers the file to the server. The method returns a **Publisher** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html)
4. The returned **Mono** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html) subscribes to the **Publisher** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) and outputs a response once the upload is complete, without blocking.

It is also possible to pass an output stream with the `transferTo` method.

The reading of the file or stream will be offloaded to the IO thread pool to prevent the possibility of blocking the event loop.

Streaming file upload

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.HttpResponse;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Post;
import io.micronaut.http.multipart.StreamingFileUpload;
import org.reactivestreams.Publisher;
import reactor.core.publisher.Mono;
import io.micronaut.core.async.annotation.SingleResult;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.IOException;
import java.io.OutputStream;
import java.nio.file.Files;

import static io.micronaut.http.HttpStatus.CONFLICT;
import static io.micronaut.http.MediaType.MULTIPART_FORM_DATA;
import static io.micronaut.http.MediaType.TEXT_PLAIN;

@Controller("/upload")
public class UploadController {

    @Post(value = "/", consumes = MULTIPART_FORM_DATA, produces = TEXT_PLAIN)  1
    @SingleResult
    public Publisher<HttpResponse<String>> upload(StreamingFileUpload file) {  2

        File tempFile;
        try {
            tempFile = File.createTempFile(file.getFilename(), "temp");
        } catch (IOException e) {
            return Mono.error(e);
        }
        Publisher<Boolean> uploadPublisher = file.transferTo(tempFile);  3

        return Mono.from(uploadPublisher)  4
            .map(success -> {
                if (success) {
                    return HttpResponse.ok("Uploaded");
                } else {
                    return HttpResponse.<String>status(CONFLICT)
                                       .body("Upload Failed");
                }
            });
    }

    @Post(value = "/outputStream", consumes = MULTIPART_FORM_DATA, produces = TEXT_PLAIN)  1
    @SingleResult
    public Mono<HttpResponse<String>> uploadOutputStream(StreamingFileUpload file) {  2

        OutputStream outputStream = new ByteArrayOutputStream();  3

        Publisher<Boolean> uploadPublisher = file.transferTo(outputStream);  4

        return Mono.from(uploadPublisher)  5
                .map(success -> {
                    if (success) {
                        return HttpResponse.ok("Uploaded");
                    } else {
                        return HttpResponse.<String>status(CONFLICT)
                                .body("Upload Failed");
                    }
                });
    }
}
```

Copy to Clipboard

1 The method consumes **MULTIPART_FORM_DATA**

2 The method parameters match form attribute names. In this case `file` will match for example an `<input type="file" name="file">`

3 A stream is created to output the data to. In real world scenarios this would come from some other source.

4 The **StreamingFileUpload.transferTo(OutputStream)** method transfers the file to the server. The method returns a **Publisher** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html)

5 The returned **Mono** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html) subscribes to the **Publisher** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) and outputs a response once the upload is complete, without blocking.

Whole Data Types

Route arguments that are not publishers cause route execution to be delayed until the upload has finished. The received data will attempt to be converted to the requested type. Conversions to a `String` or `byte[]` are supported by default. In addition, the file can be converted to a POJO if a media type codec is registered that supports the media type of the file. A media type codec is included by default that allows conversion of JSON files to POJOs.

Receiving a byte array

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.HttpResponse;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Post;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

import static io.micronaut.http.MediaType.MULTIPART_FORM_DATA;
import static io.micronaut.http.MediaType.TEXT_PLAIN;

@Controller("/upload")
public class BytesUploadController {

    @Post(value = "/bytes", consumes = MULTIPART_FORM_DATA, produces = TEXT_PLAIN) 1
    public HttpResponse<String> uploadBytes(byte[] file, String fileName) {  2
        try {
            File tempFile = File.createTempFile(fileName, "temp");
            Path path = Paths.get(tempFile.getAbsolutePath());
            Files.write(path, file);  3
            return HttpResponse.ok("Uploaded");
        } catch (IOException e) {
            return HttpResponse.badRequest("Upload Failed");
        }
    }
}
```

Copy to Clipboard

If you need knowledge about the metadata of an uploaded file, the **CompletedFileUpload** class has methods to retrieve the data of the file, and also file information such as the content type and file name.

File upload with metadata

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.HttpResponse;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Post;
import io.micronaut.http.multipart.CompletedFileUpload;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

import static io.micronaut.http.MediaType.MULTIPART_FORM_DATA;
import static io.micronaut.http.MediaType.TEXT_PLAIN;

@Controller("/upload")
public class CompletedUploadController {

    @Post(value = "/completed", consumes = MULTIPART_FORM_DATA, produces = TEXT_PLAIN) 1
    public HttpResponse<String> uploadCompleted(CompletedFileUpload file) { 2
        try {
            File tempFile = File.createTempFile(file.getFilename(), "temp"); 3
            Path path = Paths.get(tempFile.getAbsolutePath());
            Files.write(path, file.getBytes()); 3
            return HttpResponse.ok("Uploaded");
        } catch (IOException e) {
            return HttpResponse.badRequest("Upload Failed");
        }
    }
}
```

Copy to Clipboard

1 The method consumes **MULTIPART_FORM_DATA**

2 The method parameters match form attribute names. In this case the `file` will match for example an `<input type="file" name="file">`

3 The `CompletedFileUpload` instance gives access to metadata about the upload as well as access to the file contents.

If a file will not be read, the `discard` method on the file object **must** be called to prevent memory leaks.

Multiple Uploads

Different Names

If a multipart request has multiple uploads that have different part names, create an argument to your route that receives each part. For example:

JAVA

```
HttpResponse upload(String title, String name)
```

A route method signature like the above expects two different parts, one named "title" and the other "name".

Same Name

To receive multiple parts with the same part name, the argument must be a **Publisher** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html). When used in one of the following ways, the publisher emits one item per part found with the specified name. The publisher must accept one of the following types:

- **StreamingFileUpload**
- **CompletedFileUpload**
- **CompletedPart** for attributes
- Any POJO, assuming a media codec that supports the content type exists
- Another **Publisher** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) that accepts one of the chunked data types described above

For example:

JAVA

```
HttpResponse upload(Publisher<StreamingFileUpload> files)
HttpResponse upload(Publisher<CompletedFileUpload> files)
HttpResponse upload(Publisher<MyObject> files)
HttpResponse upload(Publisher<Publisher<PartData>> files)
HttpResponse upload(Publisher<CompletedPart> attributes)
```

Whole Body Binding

When request part names aren't known ahead of time, or to read the entire body, a special type can be used to indicate the entire body is desired.

If a route has an argument of type **MultipartBody** (not to be confused with the class for the client) annotated with **@Body**, each part of the request will be emitted through the argument. A **MultipartBody** is a publisher of **CompletedPart** instances.

For example:

Binding to the entire multipart body

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.annotation.Body;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Post;
import io.micronaut.http.multipart.CompletedFileUpload;
import io.micronaut.http.multipart.CompletedPart;
import io.micronaut.http.server.multipart.MultipartBody;
import org.reactivestreams.Publisher;
import org.reactivestreams.Subscriber;
import org.reactivestreams.Subscription;
import reactor.core.publisher.Mono;
import io.micronaut.core.async.annotation.SingleResult;
import static io.micronaut.http.MediaType.MULTIPART_FORM_DATA;
import static io.micronaut.http.MediaType.TEXT_PLAIN;

@Controller("/upload")
public class WholeBodyUploadController {

    @Post(value = "/whole-body", consumes = MULTIPART_FORM_DATA, produces = TEXT_PLAIN) 1
    @SingleResult
    public Publisher<String> uploadBytes(@Body MultipartBody body) { 2

        return Mono.create(emitter -> {
            body.subscribe(new Subscriber<CompletedPart>() {
                private Subscription s;

                @Override
                public void onSubscribe(Subscription s) {
                    this.s = s;
                    s.request(1);
                }

                @Override
                public void onNext(CompletedPart completedPart) {
                    String partName = completedPart.getName();
                    if (completedPart instanceof CompletedFileUpload) {
                        String originalFileName = ((CompletedFileUpload) completedPart).getFilename();
                    }
                }

                @Override
                public void onError(Throwable t) {
                    emitter.error(t);
                }

                @Override
                public void onComplete() {
                    emitter.success("Uploaded");
                }
            });
        });
    }
}
```

Copy to Clipboard

6.22 File Transfers

Micronaut supports sending files to the client in a couple of easy ways.

Sending File Objects

It is possible to return a **File** (https://docs.oracle.com/javase/8/docs/api/java/io/File.html) object from your controller method, and the data will be returned to the client. The `Content-Type` header of file responses is calculated based on the name of the file.

To control either the media type of the file being sent, or to set the file to be downloaded (i.e. using the `Content-Disposition` header), instead construct a **SystemFile** with the file to use. For example:

Sending a SystemFile

JAVA

```
@Get
public SystemFile download() {
    File file = ...
    return new SystemFile(file).attach("myfile.txt");
    // or new SystemFile(file, MediaType.TEXT_HTML_TYPE)
}
```

Sending an InputStream

For cases where a reference to a `File` object is not possible (for example resources in JAR files), Micronaut supports transferring input streams. To return a stream of data from the controller method, construct a **StreamedFile**.

The constructor for `StreamedFile` also accepts a `java.net.URL` for your convenience.

Sending a StreamedFile

JAVA

```
@Get
public StreamedFile download() {
    InputStream inputStream = ...
    return new StreamedFile(inputStream, MediaType.TEXT_PLAIN_TYPE)
    // An attach(String filename) method is also available to set the Content-Disposition
}
```

The server supports returning `304` (Not Modified) responses if the files being transferred have not changed, and the request contains the appropriate header. In addition, if the client accepts encoded responses, Micronaut encodes the file if appropriate. Encoding happens if the file is text-based and larger than 1KB by default. The threshold at which data is encoded is configurable. See the server configuration reference for details.

To use a custom data source to send data through an input stream, construct a **PipedInputStream (https://docs.oracle.com/javase/8/docs/api/java/io/PipedInputStream.html)** and **PipedOutputStream (https://docs.oracle.com/javase/8/docs/api/java/io/PipedOutputStream.html)** to write data from the output stream to the input. Make sure to do the work on a separate thread so the file can be returned immediately.

Cache Configuration

By default, file responses include caching headers. The following options determine how the `Cache-Control` header is built.

Unresolved directive in <stdin> - include::/home/runner/work/micronaut-core/micronaut-core/build/generated/configurationProperties/io.micronaut.http.server.netty.types.files.FileTypeHandlerConfiguration.adoc[]

Unresolved directive in <stdin> - include::/home/runner/work/micronaut-core/micronaut-core/build/generated/configurationProperties/io.micronaut.http.server.netty.types.files.FileTypeHandlerConfiguration.CacheControlConfiguration.adoc[]

6.23 HTTP Filters

The Micronaut HTTP server supports applying filters to request/response processing in a similar (but reactive) way to Servlet filters in traditional Java applications.

Filters support the following use cases:

- Decoration of the incoming **HttpRequest**
- Modification of the outgoing **HttpResponse**
- Implementation of cross-cutting concerns such as security, tracing, etc.

For a server application, the **HttpServerFilter** interface `doFilter` method can be implemented.

The `doFilter` method accepts the **HttpRequest** and an instance of **ServerFilterChain**.

The `ServerFilterChain` interface contains a resolved chain of filters where the final entry in the chain is the matched route. The **ServerFilterChain.proceed(io.micronaut.http.HttpRequest)** method resumes processing of the request.

The `proceed(..)` method returns a Reactive Streams **Publisher (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html)** that emits the response to be returned to the client. Implementors of filters can subscribe to the **Publisher (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html)** and mutate the emitted **MutableHttpResponse** to modify the response prior to returning the response to the client.

To put these concepts into practice lets look at an example.

Filters execute in the event loop, so blocking operations must be offloaded to another thread pool.

Writing a Filter

Suppose you wish to trace each request to the Micronaut "Hello World" example using some external system. This system could be a database or a distributed tracing service, and may require I/O operations.

You should not block the underlying Netty event loop in your filter; instead the filter should proceed with execution once any I/O is complete.

As an example, consider this `TraceService` that uses **Project Reactor** (https://projectreactor.io) to compose an I/O operation:

A TraceService Example using Reactive Streams

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.HttpRequest;
import org.reactivestreams.Publisher;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import jakarta.inject.Singleton;
import reactor.core.publisher.Mono;
import reactor.core.scheduler.Schedulers;

@Singleton
public class TraceService {

    private static final Logger LOG = LoggerFactory.getLogger(TraceService.class);

    Publisher<Boolean> trace(HttpRequest<?> request) {
        return Mono.fromCallable(() -> {  1
            LOG.debug("Tracing request: {}", request.getUri());
            // trace logic here, potentially performing I/O  2
            return true;
        }).subscribeOn(Schedulers.boundedElastic())  3
                .flux();
    }
}
```

Copy to Clipboard

1. The **Mono** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html) type creates logic that executes potentially blocking operations to write the trace data from the request
2. Since this is just an example, the logic does nothing yet
3. The `Schedulers.boundedElastic` executes the logic

You can then inject this implementation into your filter definition:

An Example HttpServerFilter

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.HttpRequest;
import io.micronaut.http.MutableHttpResponse;
import io.micronaut.http.annotation.Filter;
import io.micronaut.http.filter.HttpServerFilter;
import io.micronaut.http.filter.ServerFilterChain;
import org.reactivestreams.Publisher;
import reactor.core.publisher.Flux;

@Filter("/hello/**")  1
public class TraceFilter implements HttpServerFilter {  2

    private final TraceService traceService;

    public TraceFilter(TraceService traceService) {  3
        this.traceService = traceService;
    }

}
```

Copy to Clipboard

1. The **Filter** annotation defines the URI pattern(s) the filter matches
2. The class implements the **HttpServerFilter** interface
3. The previously defined `TraceService` is injected via constructor

The final step is to write the `doFilter` implementation of the **HttpServerFilter** interface.

The doFilter implementation

Java | Groovy | Kotlin

JAVA

```
@Override
public Publisher<MutableHttpResponse<?>> doFilter(HttpRequest<?> request,
                                                  ServerFilterChain chain) {
    return Flux.from(traceService
            .trace(request)) 1
            .switchMap(aBoolean -> chain.proceed(request)) 2
            .doOnNext(res ->
                res.getHeaders().add("X-Trace-Enabled", "true") 3
            );
}
```

Copy to Clipboard

1 `TraceService` is invoked to trace the request

2 If the call succeeds, the filter resumes request processing using **Project Reactor** (https://projectreactor.io)'s `switchMap` method, which invokes the `proceed` method of the **ServerFilterChain**

3 Finally, the **Project Reactor** (https://projectreactor.io)'s `doOnNext` method adds a `X-Trace-Enabled` header to the response.

The previous example demonstrates some key concepts such as executing logic in a non-blocking manner before proceeding with the request and modifying the outgoing response.

The examples use **Project Reactor** (https://projectreactor.io), however you can use any reactive framework that supports the Reactive streams specifications

The **Filter** can use different styles of pattern for path matching by setting `patternStyle`. By default, it uses **AntPathMatcher** for path matching. When using Ant, the mapping matches URLs using the following rules:

- ? matches one character
- * matches zero or more characters
- ** matches zero or more subdirectories in a path

Table 1. @Filter Annotation Path Matching Examples

Pattern	Example Matched Paths
`/**`	any path
`customer/j?y`	customer/joy, customer/jay
`customer/*/id`	customer/adam/id, com/amy/id
`customer/**`	customer/adam, customer/adam/id, customer/adam/name
`customer/*/.html`	customer/index.html, customer/adam/profile.html, customer/adam/job/description.html The other option is regular expression based matching. To use regular expressions, set `patternStyle = FilterPatternStyle.REGEX`. The `pattern` attribute is expected to contain a regular expression which will be expected to match the provided URLs exactly (using **Matcher#matches** (https://docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html#matches--)). NOTE: Using `FilterPatternStyle.ANT` is preferred as the pattern matching is more performant than using regular expressions. `FilterPatternStyle.REGEX` should be used when your pattern cannot be written properly using Ant. == Error States The publisher returned from `chain.proceed` should never emit an error. In the cases where an upstream filter emitted an error or the route itself threw an exception, the error response should be emitted instead of the exception. In some cases it may be desirable to know the cause of the error response and for this purpose an attribute exists on the response if it was created as a result of an exception being emitted or thrown. The original cause is stored as the attribute **EXCEPTION**.

6.24 HTTP Sessions

By default Micronaut is a stateless HTTP server, however depending on your application requirements you may need the notion of HTTP sessions.

Micronaut includes a `session` module inspired by **Spring Session** (https://projects.spring.io/spring-session/) that enables this which currently has two implementations:

- In-Memory sessions - which you should combine with an a sticky session proxy if you plan to run multiple instances.

- Redis sessions - In this case **Redis** (https://redis.io) stores sessions, and non-blocking I/O is used to read/write sessions to Redis.

Enabling Sessions

To enable support for in-memory sessions you just need the `session` dependency:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut</groupId>
    <artifactId>micronaut-session</artifactId>
</dependency>
```

Copy to Clipboard

Redis Sessions

To store **Session** instances in Redis, use the **Micronaut Redis** (https://micronaut-projects.github.io/micronaut-redis/latest/guide/#sessions) module which includes detailed instructions.

To quickly get up and running with Redis sessions you must also have the `redis-lettuce` dependency in your build:

build.gradle

GROOVY

```
compile "io.micronaut:micronaut-session"
compile "io.micronaut.redis:micronaut-redis-lettuce"
```

And enable Redis sessions via configuration in `application.yml`:

Enabling Redis Sessions

YAML

```
redis:
  uri: redis://localhost:6379
micronaut:
  session:
    http:
      redis:
        enabled: true
```

Configuring Session Resolution

Session resolution can be configured with **HttpSessionConfiguration**.

By default, sessions are resolved using an **HttpSessionFilter** that looks for session identifiers via either an HTTP header (using the `Authorization-Info` or `X-Auth-Token` headers) or via a Cookie named `SESSION`.

You can disable either header resolution or cookie resolution via configuration in `application.yml`:

Disabling Cookie Resolution

YAML

```
micronaut:
  session:
    http:
      cookie: false
      header: true
```

The above configuration enables header resolution, but disables cookie resolution. You can also configure header and cookie names.

Working with Sessions

A **Session** can be retrieved with a parameter of type **Session** in a controller method. For example consider the following controller:

Java Groovy Kotlin

JAVA

```
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.annotation.Post;
import io.micronaut.session.Session;
import io.micronaut.session.annotation.SessionValue;
import io.micronaut.core.annotation.Nullable;

import javax.validation.constraints.NotBlank;

@Controller("/shopping")
public class ShoppingController {
    private static final String ATTR_CART = "cart";  1

    @Post("/cart/{name}")
    Cart addItem(Session session, @NotBlank String name) {  2
        Cart cart = session.get(ATTR_CART, Cart.class).orElseGet(() -> {  3
            Cart newCart = new Cart();
            session.put(ATTR_CART, newCart);  4
            return newCart;
        });
        cart.getItems().add(name);
        return cart;
    }

}
```

Copy to Clipboard

1 `ShoppingController` declares a **Session** attribute named `cart`

2 The **Session** is declared as a method parameter

3 The `cart` attribute is retrieved

4 Otherwise a new `Cart` instance is created and stored in the session

Note that because the **Session** is declared as a required parameter, to execute the controller action a **Session** will be created and saved to the **SessionStore**.

If you don't want to create unnecessary sessions, declare the **Session** as `@Nullable` in which case a session will not be created and saved unnecessarily. For example:

Using @Nullable with Sessions

Java | Groovy | Kotlin

JAVA

```
    @Post("/cart/clear")
    void clearCart(@Nullable Session session) {
        if (session != null) {
            session.remove(ATTR_CART);
        }
    }
```

Copy to Clipboard

The above method only injects a new **Session** if one already exists.

Session Clients

If the client is a web browser, sessions should work if cookies are enabled. However, for programmatic HTTP clients you need to propagate the session ID between HTTP calls.

For example, when invoking the `viewCart` method of the `StoreController` in the previous example, the HTTP client receives by default a `AUTHORIZATION_INFO` header. The following example, using a Spock test, demonstrates this:

Java | Groovy | Kotlin

JAVA

```
        HttpResponse<Cart> response = Flux.from(client.exchange(HttpRequest.GET("/shopping/cart"), Cart.class))  1
                                          .blockFirst();
        Cart cart = response.body();

        assertNotNull(response.header(HttpHeaders.AUTHORIZATION_INFO));  2
        assertNotNull(cart);
        assertTrue(cart.getItems().isEmpty());
```

Copy to Clipboard

1 A request is made to `/shopping/cart`

2 The `AUTHORIZATION_INFO` header is present in the response

You can then pass this `AUTHORIZATION_INFO` in subsequent requests to reuse the existing **Session**:

Java | Groovy | Kotlin

JAVA

```
String sessionId = response.header(HttpHeaders.AUTHORIZATION_INFO);  1

response = Flux.from(client.exchange(HttpRequest.POST("/shopping/cart/Apple", "")
                .header(HttpHeaders.AUTHORIZATION_INFO, sessionId), Cart.class))  2
                .blockFirst();
cart = response.body();
```

Copy to Clipboard

1 The `AUTHORIZATION_INFO` is retrieved from the response

2 And then sent as a header in the subsequent request

Using @SessionValue

Rather than explicitly injecting the **Session** into a controller method, you can instead use **@SessionValue**. For example:

Using @SessionValue

Java | Groovy | Kotlin

JAVA

```
@Get("/cart")
@SessionValue(ATTR_CART)  1
Cart viewCart(@SessionValue @Nullable Cart cart) {  2
    if (cart == null) {
        cart = new Cart();
    }
    return cart;
}
```

Copy to Clipboard

1 **@SessionValue** is declared on the method resulting in the return value being stored in the **Session**. Note that you must specify the attribute name when used on a return value

2 **@SessionValue** is used on a `@Nullable` parameter which results in looking up the value from the **Session** in a non-blocking way and supplying it if present. In the case a value is not specified to **@SessionValue** resulting in the parameter name being used to lookup the attribute.

Session Events

You can register **ApplicationEventListener** beans to listen for **Session** related events located in the **io.micronaut.session.event** package.

The following table summarizes the events:

Table 1. Session Events

Type	Description
SessionCreatedEvent	Fired when a **Session** is created
SessionDeletedEvent	Fired when a **Session** is deleted
SessionExpiredEvent	Fired when a **Session** expires
SessionDestroyedEvent	Parent of both `SessionDeletedEvent` and `SessionExpiredEvent`

6.25 Server Sent Events

The Micronaut HTTP server supports emitting **Server Sent Events (SSE)** (https://en.wikipedia.org/wiki/Server-sent_events) using the **Event** API.

To emit events from the server, return a Reactive Streams **Publisher** (https://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) that emits objects of type **Event**.

The **Publisher** (https://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) itself could publish events from a background task, via an event system, etc.

Imagine for an example a event stream of news headlines; you may define a data class as follows:

Headline

Java | Groovy | Kotlin

JAVA

```
public class Headline {

    private String title;
    private String description;

    public Headline() {}

    public Headline(String title, String description) {
        this.title = title;
        this.description = description;
    }

    public String getTitle() {
        return title;
    }

    public String getDescription() {
        return description;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}
```

Copy to Clipboard

To emit news headline events, write a controller that returns a **Publisher** (https://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) of **Event** instances using whichever Reactive library you prefer. The example below uses **Project Reactor** (https://projectreactor.io)'s **Flux** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html) via the `generate` method:

Publishing Server Sent Events from a Controller

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.MediaType;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.sse.Event;
import io.micronaut.scheduling.TaskExecutors;
import io.micronaut.scheduling.annotation.ExecuteOn;
import org.reactivestreams.Publisher;
import reactor.core.publisher.Flux;

@Controller("/headlines")
public class HeadlineController {

    @ExecuteOn(TaskExecutors.IO)
    @Get(produces = MediaType.TEXT_EVENT_STREAM)
    public Publisher<Event<Headline>> index() {  1
        String[] versions = {"1.0", "2.0"};  2
        return Flux.generate(() -> 0, (i, emitter) -> {  3
            if (i < versions.length) {
                emitter.next(  4
                    Event.of(new Headline("Micronaut " + versions[i] + " Released", "Come and get it"))
                );
            } else {
                emitter.complete();  5
            }
            return ++i;
        });
    }
}
```

Copy to Clipboard

1 The controller method returns a **Publisher** (https://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) of **Event**

2 A headline is emitted for each version of Micronaut

3 The **Flux** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html) type's `generate` method generates a **Publisher** (https://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html). The `generate` method accepts an initial value and a lambda that accepts the value and a **Emitter** (http://reactivex.io/RxJava/2.x/javadoc/io/reactivex/Emitter.html). Note that this example executes on the same thread as the controller action, but you could use `subscribeOn` or map an existing "hot" **Flux** (http://reactivex.io/RxJava/2.x/javadoc/io/reactivex/Flux.html).

4 The **Emitter** (http://reactivex.io/RxJava/2.x/javadoc/io/reactivex/Emitter.html) interface `onNext` method emits objects of type **Event**. The **Event.of(ET)** factory method constructs the event.

5 The **Emitter** (http://reactivex.io/RxJava/2.x/javadoc/io/reactivex/Emitter.html) interface `onComplete` method indicates when to finish sending server sent events.

You typically want to schedule SSE event streams on a separate executor. The previous example uses **@ExecuteOn** to execute the stream on the I/O executor.

The above example sends back a response of type `text/event-stream` and for each **Event** emitted the `Headline` type previously will be converted to JSON resulting in responses such as:

Server Sent Event Response Output

JSON

```
data: {"title":"Micronaut 1.0 Released","description":"Come and get it"}
data: {"title":"Micronaut 2.0 Released","description":"Come and get it"}
```

You can use the methods of the **Event** interface to customize the Server Sent Event data sent back, including associating event ids, comments, retry timeouts, etc.

6.26 WebSocket Support

Micronaut features dedicated support for creating WebSocket clients and servers. The **io.micronaut.websocket.annotation** package includes annotations for defining both clients and servers.

6.26.1 Using @ServerWebSocket

The **@ServerWebSocket** annotation can be applied to any class that should map to a WebSocket URI. The following example is a simple chat WebSocket implementation:

WebSocket Chat Example

Java	Groovy	Kotlin

JAVA

```
import io.micronaut.websocket.WebSocketBroadcaster;
import io.micronaut.websocket.WebSocketSession;
import io.micronaut.websocket.annotation.OnClose;
import io.micronaut.websocket.annotation.OnMessage;
import io.micronaut.websocket.annotation.OnOpen;
import io.micronaut.websocket.annotation.ServerWebSocket;

import java.util.function.Predicate;

@ServerWebSocket("/chat/{topic}/{username}")  1
public class ChatServerWebSocket {

    private final WebSocketBroadcaster broadcaster;

    public ChatServerWebSocket(WebSocketBroadcaster broadcaster) {
        this.broadcaster = broadcaster;
    }

    @OnOpen  2
    public void onOpen(String topic, String username, WebSocketSession session) {
        String msg = "[" + username + "] Joined!";
        broadcaster.broadcastSync(msg, isValid(topic, session));
    }

    @OnMessage  3
    public void onMessage(String topic, String username,
                          String message, WebSocketSession session) {
        String msg = "[" + username + "] " + message;
        broadcaster.broadcastSync(msg, isValid(topic, session));  4
    }

    @OnClose  5
    public void onClose(String topic, String username, WebSocketSession session) {
        String msg = "[" + username + "] Disconnected!";
        broadcaster.broadcastSync(msg, isValid(topic, session));
    }

    private Predicate<WebSocketSession> isValid(String topic, WebSocketSession session) {
        return s -> s != session &&
                topic.equalsIgnoreCase(s.getUriVariables().get("topic", String.class, null));
    }
}
```

Copy to Clipboard

1. The **@ServerWebSocket** annotation defines the path the WebSocket is mapped under. The URI can be a URI template.
2. The **@OnOpen** annotation declares the method to invoke when the WebSocket is opened.
3. The **@OnMessage** annotation declares the method to invoke when a message is received.
4. You can use a **WebSocketBroadcaster** to broadcast messages to every WebSocket session. You can filter which sessions to send to with a `Predicate`. Also, you could use the **WebSocketSession** instance to send a message to it with `WebSocketSession::send`.
5. The **@OnClose** annotation declares the method to invoke when the WebSocket is closed.

A working example of WebSockets in action can be found in the **Micronaut Examples (https://github.com/micronaut-projects/micronaut-examples/tree/master/websocket-chat)** GitHub repository.

For binding, method arguments to each WebSocket method can be:

- A variable from the URI template (in the above example `topic` and `username` are URI template variables)
- An instance of **WebSocketSession**

The @OnClose Method

The **@OnClose** method can optionally receive a **CloseReason**. The `@OnClose` method is invoked prior to the session closing.

The @OnMessage Method

The **@OnMessage** method can define a parameter for the message body. The parameter can be one of the following:

- A Netty `WebSocketFrame`
- Any Java primitive or simple type (such as `String`). In fact, any type that can be converted from `ByteBuf` (you can register additional **TypeConverter** beans to support a custom type).
- A `byte[]`, a `ByteBuf` or a Java NIO `ByteBuffer`.
- A POJO. In this case, it will be decoded by default as JSON using **JsonMediaTypeCodec**. You can register a custom codec and define the content type of the handler using the **@Consumes** annotation.

The @OnError Method

A method annotated with **@OnError** can be added to implement custom error handling. The `@OnError` method can define a parameter that receives the exception type to be handled. If no `@OnError` handling is present and an unrecoverable exception occurs, the WebSocket is automatically closed.

Non-Blocking Message Handling

The previous example uses the `broadcastSync` method of the **WebSocketBroadcaster** interface which blocks until the broadcast is complete. A similar `sendSync` method exists in **WebSocketSession** to send a message to a single receiver in a blocking manner. You can however implement non-blocking WebSocket servers by instead returning a **Publisher** (https://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) or a **Future** (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html) from each WebSocket handler method. For example:

WebSocket Chat Example

Java | Groovy | Kotlin

JAVA

```
@OnMessage
public Publisher<Message> onMessage(String topic, String username,
                                    Message message, WebSocketSession session) {
    String text = "[" + username + "] " + message.getText();
    Message newMessage = new Message(text);
    return broadcaster.broadcast(newMessage, isValid(topic, session));
}
```

Copy to Clipboard

The example above uses `broadcast`, which creates an instance of **Publisher** (https://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) and returns the value to Micronaut. Micronaut sends the message asynchronously based on the Publisher interface. The similar `send` method sends a single message asynchronously via Micronaut return value.

For sending messages asynchronously outside Micronaut annotated handler methods, you can use `broadcastAsync` and `sendAsync` methods in their respective **WebSocketBroadcaster** and **WebSocketSession** interfaces. For blocking sends, the `broadcastSync` and `sendSync` methods can be used.

@ServerWebSocket and Scopes

By default, the `@ServerWebSocket` instance is shared for all WebSocket connections. Extra care must be taken to synchronize local state to avoid thread safety issues.

If you prefer to have an instance for each connection, annotate the class with **@Prototype**. This lets you retrieve the **WebSocketSession** from the `@OnOpen` handler and assign it to a field of the `@ServerWebSocket` instance.

Sharing Sessions with the HTTP Session

The **WebSocketSession** is by default backed by an in-memory map. If you add the `session` module you can however share **sessions** between the HTTP server and the WebSocket server.

When sessions are backed by a persistent store such as Redis, after each message is processed the session is updated to the backing store.

Using the CLI

If you created your project using Application Type `Micronaut Application`, you can use the `create-websocket-server` command with the Micronaut CLI to create a class annotated with **ServerWebSocket**.

```
$ mn create-websocket-server MyChat
| Rendered template WebsocketServer.java to destination src/main/java/example/MyChatServer.java
```

Connection Timeouts

By default, Micronaut times out idle connections with no activity after five minutes. Normally this is not a problem as browsers automatically reconnect WebSocket sessions, however you can control this behaviour by setting the `micronaut.server.idle-timeout` setting (a negative value results in no timeout):

Setting the Connection Timeout for the Server

YAML

```
micronaut:
  server:
    idle-timeout: 30m # 30 minutes
```

If you use Micronaut's WebSocket client you may also wish to set the timeout on the client:

Setting the Connection Timeout for the Client

YAML

```yaml
micronaut:
  http:
    client:
      read-idle-timeout: 30m # 30 minutes
```

6.26.2 Using @ClientWebSocket

The **@ClientWebSocket** annotation can be used with the **WebSocketClient** interface to define WebSocket clients.

You can inject a reference to a **WebSocketClient** using the **@Client** annotation:

JAVA

```java
@Inject
@Client("http://localhost:8080")
WebSocketClient webSocketClient;
```

This lets you use the same service discovery and load balancing features for WebSocket clients.

Once you have a reference to the **WebSocketClient** interface you can use the `connect` method to obtain a connected instance of a bean annotated with **@ClientWebSocket**.

For example consider the following implementation:

WebSocket Chat Example

Java	Groovy	Kotlin

JAVA

```
import io.micronaut.http.HttpRequest;
import io.micronaut.websocket.WebSocketSession;
import io.micronaut.websocket.annotation.ClientWebSocket;
import io.micronaut.websocket.annotation.OnMessage;
import io.micronaut.websocket.annotation.OnOpen;
import org.reactivestreams.Publisher;
import io.micronaut.core.async.annotation.SingleResult;
import java.util.Collection;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.Future;

@ClientWebSocket("/chat/{topic}/{username}")  1
public abstract class ChatClientWebSocket implements AutoCloseable {  2

    private WebSocketSession session;
    private HttpRequest request;
    private String topic;
    private String username;
    private Collection<String> replies = new ConcurrentLinkedQueue<>();

    @OnOpen
    public void onOpen(String topic, String username,
                       WebSocketSession session, HttpRequest request) {  3
        this.topic = topic;
        this.username = username;
        this.session = session;
        this.request = request;
    }

    public String getTopic() {
        return topic;
    }

    public String getUsername() {
        return username;
    }

    public Collection<String> getReplies() {
        return replies;
    }

    public WebSocketSession getSession() {
        return session;
    }

    public HttpRequest getRequest() {
        return request;
    }

    @OnMessage
    public void onMessage(String message) {
        replies.add(message);  4
    }
```

Copy to Clipboard

1. The class is abstract (more on that later) and is annotated with **@ClientWebSocket**
2. The client must implement `AutoCloseable` and you should ensure that the connection is closed at some point.
3. You can use the same annotations as on the server, in this case `@OnOpen` to obtain a reference to the underlying session.
4. The `@OnMessage` annotation defines the method that receives responses from the server.

You can also define abstract methods that start with either `send` or `broadcast` and these methods will be implemented for you at compile time. For example:

WebSocket Send Methods

JAVA

```
public abstract void send(String message);
```

Note by returning `void` this tells Micronaut that the method is a blocking send. You can instead define methods that return either futures or a **Publisher** (https://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html):

WebSocket Send Methods

JAVA

```
public abstract reactor.core.publisher.Mono<String> send(String message);
```

The above example defines a send method that returns a **Mono** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html).

WebSocket Send Methods

JAVA

```
public abstract java.util.concurrent.Future<String> sendAsync(String message);
```

The above example defines a send method that executes asynchronously and returns a **Future** (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html) to access the results.

Once you have defined a client class you can connect to the client socket and start sending messages:

Connecting a Client WebSocket

JAVA

```
ChatClientWebSocket chatClient = webSocketClient
    .connect(ChatClientWebSocket.class, "/chat/football/fred")
    .blockFirst();
chatClient.send("Hello World!");
```

For illustration purposes we use `blockFirst()` to obtain the client. It is however possible to combine `connect` (which returns a **Flux** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html)) to perform non-blocking interaction via WebSocket.

Using the CLI

If you created your project using the Micronaut CLI and the default (`service`) profile, you can use the `create-websocket-client` command to create an abstract class with **WebSocketClient**.

```
$ mn create-websocket-client MyChat
| Rendered template WebsocketClient.java to destination src/main/java/example/MyChatClient.java
```

6.27 HTTP/2 Support

Since Micronaut 2.x, Micronaut's Netty-based HTTP server can be configured to support HTTP/2.

Configuring the Server for HTTP/2

The first step is to set the supported HTTP version in the server configuration:

Enabling HTTP/2 Support

YAML

```
micronaut:
  server:
    http-version: 2.0
```

With this configuration, Micronaut enables support for the `h2c` protocol (see **HTTP/2 over cleartext** (https://http2.github.io/http2-spec/#discover-http)) which is fine for development.

Since browsers don't support `h2c` and in general **HTTP/2 over TLS** (https://http2.github.io/http2-spec/#discover-https) (the `h2` protocol), it is recommended for production that you enable **HTTPS support**. For development this can be done with:

Enabling `h2` Protocol Support

YAML

```
micronaut:
  ssl:
    enabled: true
    buildSelfSigned: true
  server:
    http-version: 2.0
```

For production, see the **configuring HTTPS** section of the documentation.

Note that if your deployment environment uses JDK 8, or for improved support for OpenSSL, define the following dependencies on Netty Tomcat Native:

Gradle	**Maven**

MAVEN

```
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-tcnative</artifactId>
    <version>2.0.40.Final</version>
    <scope>runtime</scope>
</dependency>
```

Copy to Clipboard

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-tcnative-boringssl-static</artifactId>
    <version>2.0.40.Final</version>
    <scope>runtime</scope>
</dependency>
```

Copy to Clipboard

In addition to a dependency on the appropriate native library for your architecture. For example:

Configuring Tomcat Native

GROOVY

```
runtimeOnly "io.netty:netty-tcnative-boringssl-static:2.0.40.Final:${Os.isFamily(Os.FAMILY_MAC) ? 'osx-x86_64' : 'linux
```

See the documentation on **Tomcat Native** (https://netty.io/wiki/forked-tomcat-native.html) for more information.

HTTP/2 Clients

By default, Micronaut's HTTP client is configured to support HTTP 1.1. To enable support for HTTP/2, set the supported HTTP version in configuration:

Enabling HTTP/2 in Clients

YAML

```
micronaut:
  http:
    client:
      http-version: 2.0
```

Or by specifying the HTTP version to use when injecting the client:

Injecting a HTTP/2 Client

JAVA

```
@Inject
@Client(httpVersion=HttpVersion.HTTP_2_0)
ReactorHttpClient client;
```

6.28 Server Events

The HTTP server emits a number of **Bean Events**, defined in the **io.micronaut.runtime.server.event** package, that you can write listeners for. The following table summarizes these:

Table 1. Server Events

Event	Description
ServerStartupEvent	Emitted when the server completes startup
ServerShutdownEvent	Emitted when the server shuts down
ServiceReadyEvent	Emitted after all **ServerStartupEvent** listeners have been invoked and exposes the **EmbeddedServerInstance**
ServiceStoppedEvent	Emitted after all **ServerShutdownEvent** listeners have been invoked and exposes the **EmbeddedServerInstance**

Doing significant work within a listener for a **ServerStartupEvent** will increase startup time.

The following example defines a **ApplicationEventListener** that listens for **ServerStartupEvent**:

Listening for Server Startup Events

JAVA

```
import io.micronaut.context.event.ApplicationEventListener;
...
@Singleton
public class StartupListener implements ApplicationEventListener<ServerStartupEvent> {
    @Override
    public void onApplicationEvent(ServerStartupEvent event) {
        // logic here
        ...
    }
}
```

Alternatively, you can also use the **@EventListener** annotation on a method of any bean that accepts `ServerStartupEvent`:

Using `@EventListener` *with* `ServerStartupEvent`

JAVA

```
import io.micronaut.runtime.event.annotation.EventListener;
import io.micronaut.runtime.server.event.ServerStartupEvent;
import javax.inject.Singleton;
...
@Singleton
public class MyBean {

    @EventListener
    public void onStartup(ServerStartupEvent event) {
        // logic here
        ...
    }
}
```

6.29 Configuring the HTTP Server

The HTTP server features a number of configuration options. They are defined in the **NettyHttpServerConfiguration** configuration class, which extends **HttpServerConfiguration**.

The following example shows how to tweak configuration options for the server via `application.yml`:

Configuring HTTP server settings

YAML

```
micronaut:
  server:
    maxRequestSize: 1MB
    host: localhost  1
    netty:
      maxHeaderSize: 500KB  2
      worker:
        threads: 8  3
      childOptions:
        autoRead: true  4
```

1. By default Micronaut binds to all network interfaces. Use `localhost` to bind only to loopback network interface
2. Maximum size for headers
3. Number of Netty worker threads
4. Auto read request body

Unresolved directive in <stdin> - include::/home/runner/work/micronaut-core/micronaut-core/build/generated/configurationProperties/io.micronaut.http.server.netty.configuration.NettyHttpServerConfiguration.adoc[]

Using Native Transports

The native Netty transports add features specific to a particular platform, generate less garbage, and generally improve performance when compared to the NIO-based transport.

To enable native transports, first add a dependency:

For macOS:

Gradle	**Maven**

MAVEN

```
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-transport-native-kqueue</artifactId>
    <scope>runtime</scope>
    <classifier>osx-x86_64</classifier>
</dependency>
```

Copy to Clipboard

For Linux on x86:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-transport-native-epoll</artifactId>
    <scope>runtime</scope>
    <classifier>linux-x86_64</classifier>
</dependency>
```

Copy to Clipboard

For Linux on ARM64:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-transport-native-epoll</artifactId>
    <scope>runtime</scope>
    <classifier>linux-aarch_64</classifier>
</dependency>
```

Copy to Clipboard

Then configure the default event loop group to prefer native transports:

Configuring The Default Event Loop to Prefer Native Transports

YAML

```
micronaut:
  netty:
    event-loops:
      default:
        prefer-native-transport: true
```

Netty enables simplistic sampling resource leak detection which reports there is a leak or not, at the cost of small overhead. You can disable it or enable more advanced detection by setting property `netty.resource-leak-detector-level` to one of: `SIMPLE` (default), `DISABLED`, `PARANOID` or `ADVANCED`.

6.29.1 Configuring Server Thread Pools

The HTTP server is built on **Netty** (http://netty.io) which is designed as a non-blocking I/O toolkit in an event loop model.

The Netty worker event loop uses the "default" named event loop group. This can be configured through `micronaut.netty.event-loops.default`.

The event loop configuration under `micronaut.server.netty.worker` is only used if the `event-loop-group` is set to a name which doesn't correspond to any `micronaut.netty.event-loops` configuration. This behavior is deprecated and will be removed in a future version. Use `micronaut.netty.event-loops.*` for any event loop group configuration beyond setting the name through `event-loop-group`. This does not apply to the parent event loop configuration (`micronaut.server.netty.parent`).

Unresolved directive in <stdin> - include::/home/runner/work/micronaut-core/micronaut-core/build/generated/configurationProperties/io.micronaut.http.server.netty.configuration.NettyHttpServerConfiguration.Worker.adoc[]

The parent event loop can be configured with `micronaut.server.netty.parent` with the same configuration options.

The server can also be configured to use a different named worker event loop:

Using a different event loop for the server

YAML

```yaml
micronaut:
  server:
    netty:
      worker:
        event-loop-group: other
  netty:
    event-loops:
      other:
        num-threads: 10
```

The default value for the number of threads is the value of the system property `io.netty.eventLoopThreads`, or if not specified, the available processors x 2.

See the following table for configuring event loops:

Unresolved directive in <stdin> - include::/home/runner/work/micronaut-core/micronaut-core/build/generated/configurationProperties/io.micronaut.http.netty.channel.DefaultEventLoopGroupConfiguration.adoc[]

Blocking Operations

When dealing with blocking operations, Micronaut shifts the blocking operations to an unbound, caching I/O thread pool by default. You can configure the I/O thread pool using the **ExecutorConfiguration** named `io`. For example:

Configuring the Server I/O Thread Pool

YAML

```yaml
micronaut:
  executors:
    io:
      type: fixed
      nThreads: 75
```

The above configuration creates a fixed thread pool with 75 threads.

6.29.2 Configuring the Netty Pipeline

You can customize the Netty pipeline by writing a **Bean Event Listener** that listens for the creation of **ChannelPipelineCustomizer**.

Both the Netty HTTP server and client implement this interface and it lets you customize the Netty `ChannelPipeline` and add additional handlers.

The **ChannelPipelineCustomizer** interface defines constants for the names of the various handlers Micronaut registers.

As an example the following code sample demonstrates registering the **Logbook** (https://github.com/zalando/logbook) library which includes additional Netty handlers to perform request and response logging:

Customizing the Netty pipeline for Logbook

Java	Groovy	Kotlin

JAVA

```
import io.micronaut.context.annotation.Requires;
import io.micronaut.context.event.BeanCreatedEvent;
import io.micronaut.context.event.BeanCreatedEventListener;
import io.micronaut.http.netty.channel.ChannelPipelineCustomizer;
import org.zalando.logbook.Logbook;
import org.zalando.logbook.netty.LogbookClientHandler;
import org.zalando.logbook.netty.LogbookServerHandler;

import jakarta.inject.Singleton;

@Requires(beans = Logbook.class)
@Singleton
public class LogbookPipelineCustomizer
        implements BeanCreatedEventListener<ChannelPipelineCustomizer> {  1

    private final Logbook logbook;

    public LogbookPipelineCustomizer(Logbook logbook) {
        this.logbook = logbook;
    }

    @Override
    public ChannelPipelineCustomizer onCreated(BeanCreatedEvent<ChannelPipelineCustomizer> event) {
        ChannelPipelineCustomizer customizer = event.getBean();

        if (customizer.isServerChannel()) {  2
            customizer.doOnConnect(pipeline -> {
                pipeline.addAfter(
                    ChannelPipelineCustomizer.HANDLER_HTTP_SERVER_CODEC,
                    "logbook",
                    new LogbookServerHandler(logbook)
                );
                return pipeline;
            });
        } else {  3
            customizer.doOnConnect(pipeline -> {
                pipeline.addAfter(
                        ChannelPipelineCustomizer.HANDLER_HTTP_CLIENT_CODEC,
                        "logbook",
                        new LogbookClientHandler(logbook)
                );
                return pipeline;
            });
        }
        return customizer;
    }
}
```

Copy to Clipboard

1 `LogbookPipelineCustomizer` implements **ChannelPipelineCustomizer** and requires the definition of a `Logbook` bean

2 If the bean being created is the server, the server handler is registered

3 if the bean being created is the client, the client handler is registered

6.29.3 Configuring CORS

Micronaut supports CORS (**Cross Origin Resource Sharing** (https://www.w3.org/TR/cors/)) out of the box. By default, CORS requests are rejected. To enable processing of CORS requests, modify your configuration. For example with `application.yml`:

CORS Configuration Example

YAML

```
micronaut:
  server:
    cors:
      enabled: true
```

By only enabling CORS processing, a "wide open" strategy is adopted that allows requests from any origin.

To change the settings for all origins or a specific origin, change the configuration to provide one or more "configurations". By providing any configuration, the default "wide open" configuration is not configured.

CORS Configurations

YAML

```yaml
micronaut:
  server:
    cors:
      enabled: true
      configurations:
        all:
          ...
        web:
          ...
        mobile:
          ...
```

In the above example, three configurations are provided. Their names (`all`, `web`, `mobile`) are not important and have no significance inside Micronaut. They are there purely to be able to easily recognize the intended user of the configuration.

The same configuration properties can be applied to each configuration. See **CorsOriginConfiguration** for properties that can be defined. The values of each configuration supplied will default to the default values of the corresponding fields.

When a CORS request is made, configurations are searched for allowed origins that match exactly or match the request origin through a regular expression.

Allowed Origins

To allow any origin for a given configuration, don't include the `allowedOrigins` key in your configuration.

For multiple valid origins, set the `allowedOrigins` key of the configuration to a list of strings. Each value can either be a static value (`http://www.foo.com`) or a regular expression (`^http(|s)://www\.google\.com$`).

Regular expressions are passed to **Pattern#compile** (https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html#compile-java.lang.String-) and compared to the request origin with **Matcher#matches** (https://docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html#matches--).

Example CORS Configuration

YAML

```yaml
micronaut:
  server:
    cors:
      enabled: true
      configurations:
        web:
          allowedOrigins:
            - http://foo.com
            - ^http(|s):\/\/www\.google\.com$
```

Allowed Methods

To allow any request method for a given configuration, don't include the `allowedMethods` key in your configuration.

For multiple allowed methods, set the `allowedMethods` key of the configuration to a list of strings.

Example CORS Configuration

YAML

```yaml
micronaut:
  server:
    cors:
      enabled: true
      configurations:
        web:
          allowedMethods:
            - POST
            - PUT
```

Allowed Headers

To allow any request header for a given configuration, don't include the `allowedHeaders` key in your configuration.

For multiple allowed headers, set the `allowedHeaders` key of the configuration to a list of strings.

Example CORS Configuration

YAML

```yaml
micronaut:
  server:
    cors:
      enabled: true
      configurations:
        web:
          allowedHeaders:
            - Content-Type
            - Authorization
```

Exposed Headers

To configure the headers that are sent in the response to a CORS request through the `Access-Control-Expose-Headers` header, include a list of strings for the `exposedHeaders` key in your configuration. None are exposed by default.

Example CORS Configuration

YAML

```yaml
micronaut:
  server:
    cors:
      enabled: true
      configurations:
        web:
          exposedHeaders:
            - Content-Type
            - Authorization
```

Allow Credentials

Credentials are allowed by default for CORS requests. To disallow credentials, set the `allowCredentials` option to `false`.

Example CORS Configuration

YAML

```yaml
micronaut:
  server:
    cors:
      enabled: true
      configurations:
        web:
          allowCredentials: false
```

Max Age

The default maximum age that preflight requests can be cached is 30 minutes. To change that behavior, specify a value in seconds.

Example CORS Configuration

YAML

```yaml
micronaut:
  server:
    cors:
      enabled: true
      configurations:
        web:
          maxAge: 3600 # 1 hour
```

Multiple Header Values

By default, when a header has multiple values, multiple headers are sent, each with a single value. It is possible to change the behavior to send a single header with a comma-separated list of values by setting a configuration option.

YAML

```yaml
micronaut:
  server:
    cors:
      single-header: true
```

6.29.4 Securing the Server with HTTPS

Micronaut supports HTTPS out of the box. By default HTTPS is disabled and all requests are served using HTTP. To enable HTTPS support, modify your configuration. For example with `application.yml`:

HTTPS Configuration Example

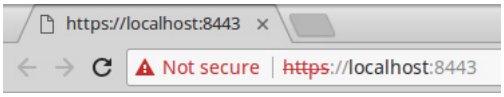
YAML

```
micronaut:
  ssl:
    enabled: true
    buildSelfSigned: true  1
```

1 Micronaut will create a self-signed certificate.

By default Micronaut with HTTPS support starts on port `8443` but you can change the port with the property `micronaut.ssl.port`.

This configuration will generate a warning in the browser.



Using a valid x509 certificate

It is also possible to configure Micronaut to use an existing valid x509 certificate, for example one created with **Let's Encrypt** (https://letsencrypt.org/). You will need the `server.crt` and `server.key` files and to convert them to a PKCS #12 file.

BASH

```
$ openssl pkcs12 -export \
                 -in server.crt \  1
                 -inkey server.key \  2
                 -out server.p12 \  3
                 -name someAlias \  4
                 -chain -CAfile ca.crt -caname root
```

1 The original `server.crt` file

2 The original `server.key` file

3 The `server.p12` file to create

4 The alias for the certificate

During the creation of the `server.p12` file it is necessary to define a password that will be required later when using the certificate in Micronaut.

Now modify your configuration:

HTTPS Configuration Example

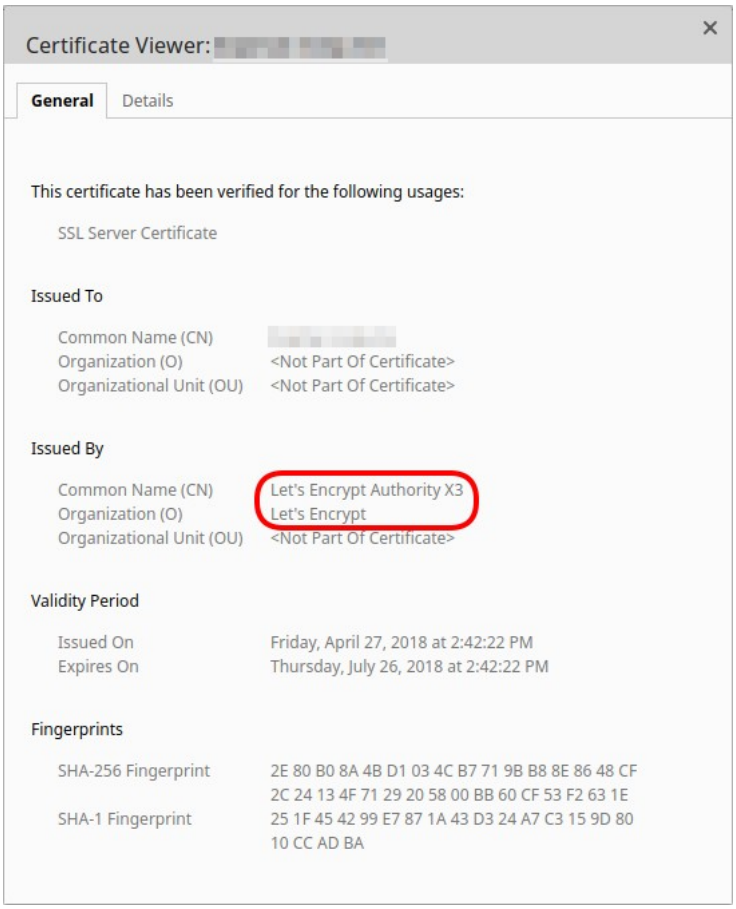
YAML

```
micronaut:
  ssl:
    enabled: true
    keyStore:
      path: classpath:server.p12  1
      password: mypassword  2
      type: PKCS12
```

1 The `p12` file. It can also be referenced as `file:/path/to/the/file`

2 The password defined during the export

With this configuration, if we start Micronaut and connect to **`https://localhost:8443`** we still see the warning in the browser, but if we inspect the certificate we can check that it is the one generated by Let's Encrypt.

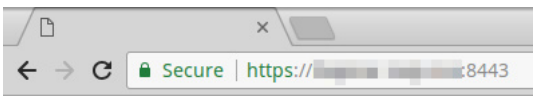


Finally, we can test that the certificate is valid for the browser by adding an alias to the domain in `/etc/hosts` file:

BASH

```
$ cat /etc/hosts
...
127.0.0.1   my-domain.org
...
```

Now we can connect to **`https://my-domain.org:8443`**:



Using Java Keystore (JKS)

Using this type of certificate is not recommended because the format is proprietary - PKCS12 format is preferred. Regardless, Micronaut also supports it.

Convert the `p12` certificate to a JKS one:

BASH

```
$ keytool -importkeystore \
          -deststorepass newPassword -destkeypass newPassword \  1
          -destkeystore server.keystore \  2
          -srckeystore server.p12 -srcstoretype PKCS12 -srcstorepass mypassword \  3
          -alias someAlias  4
```

1. It is necessary to define the password for the keystore
2. The file to create
3. The PKCS12 file created previously, and the password defined during the creation
4. The alias used before

If either `srcstorepass` or `alias` are not the same as defined in the `p12` file, the conversion will fail.

Now modify your configuration:

HTTPS Configuration Example

YAML

```yaml
micronaut:
  ssl:
    enabled: true
    keyStore:
      path: classpath:server.keystore
      password: newPassword
      type: JKS
```

Start Micronaut, and the application will run on `https://localhost:8443` using the certificate in the keystore.

Refreshing/Reloading HTTPS Certificates

Keeping HTTPS certificates up-to-date after expiry can be a challenge. A great solution to this is **Automated Certificate Management Environment (https://en.wikipedia.org/wiki/Automated_Certificate_Management_Environment)** (ACME) and the **Micronaut ACME Module (https://micronaut-projects.github.io/micronaut-acme/latest/guide/index.html)** which provides support for automatically refreshing certificates from a certificate authority.

If the use of a certificate authority is not possible and you need to manually update certificates from disk then you should fire a **RefreshEvent** using Micronaut's support for **Application Events** containing the keys where your HTTPS configuration is defined and Micronaut will reload the certificates from disk and apply the new configuration to the server.

You can also use the **Refresh Management Endpoint**, however this will only apply if the physical location of certificate on disk has changed

For example the following will reload the previously listed HTTPS configuration from disk and apply it to new incoming requests (this code could run for a **scheduled job** that polled certificates for changes for example):

Manually Refreshing HTTPS configuration

JAVA

```java
import jakarta.inject.Inject;
import io.micronaut.context.event.ApplicationEventPublisher;
import io.micronaut.runtime.context.scope.refresh.RefreshEvent;
import java.util.Collections;
...

@Inject ApplicationEventPublisher<RefreshEvent> eventPublisher;

...

eventPublisher.publishEvent(new RefreshEvent(
    Collections.singletonMap("micronaut.ssl", "*")
));
```

6.29.5 Enabling HTTP and HTTPS

Micronaut supports binding both HTTP and HTTPS. To enable dual protocol support, modify your configuration. For example with `application.yml`:

Dual Protocol Configuration Example

YAML

```yaml
micronaut:
  ssl:
    enabled: true
    build-self-signed: true  1
  server:
    dual-protocol : true  2
```

1. You must configure SSL for HTTPS to work. In this example we are just using a self-signed certificate, but see **Securing the Server with HTTPS** for other configurations
2. Enabling both HTTP and HTTPS is an opt-in feature - setting the `dualProtocol` flag enables it. By default Micronaut only enables one

It is also possible to redirect automatically all HTTP request to HTTPS. Besides the previous configuration, you need to enable this option. For example, with `application.yml`:

Enable HTTP to HTTPS Redirects

YAML

```yaml
micronaut:
  ssl:
    enabled: true
    build-self-signed: true
  server:
    dual-protocol : true
    http-to-https-redirect: true  1
```

1 Enable HTTP to HTTPS redirects

6.29.6 Enabling Access Logger

In the spirit of [apache mod_log_config (http://httpd.apache.org/docs/current/mod/mod_log_config.html)](http://httpd.apache.org/docs/current/mod/mod_log_config.html) and [Tomcat Access Log Valve (https://tomcat.apache.org/tomcat-10.0-doc/config/valve.html#Access_Logging)](https://tomcat.apache.org/tomcat-10.0-doc/config/valve.html#Access_Logging), it is possible to enable an access logger for the HTTP server (this works for both HTTP/1 and HTTP/2).

To enable and configure the access logger, in `application.yml` set:

Enabling the access logger

YAML

```yaml
micronaut:
  server:
    netty:
      access-logger:
        enabled: true # Enables the access logger
        logger-name: my-access-logger # A logger name, optional, default is `HTTP_ACCESS_LOGGER`
        log-format: common # A log format, optional, default is Common Log Format
```

Logback Configuration

In addition to enabling the access logger, you must add a logger for the specified or default logger name. For instance using the default logger name for logback:

Logback configuration

XML

```xml
<appender
    name="httpAccessLogAppender"
    class="ch.qos.logback.core.rolling.RollingFileAppender">
    <append>true</append>
    <file>log/http-access.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <!-- daily rollover -->
        <fileNamePattern>log/http-access-%d{yyyy-MM-dd}.log
        </fileNamePattern>
        <maxHistory>7</maxHistory>
    </rollingPolicy>
    <encoder>
        <charset>UTF-8</charset>
        <pattern>%msg%n</pattern>
    </encoder>
    <immediateFlush>true</immediateFlush>
</appender>

<logger name="HTTP_ACCESS_LOGGER" additivity="false" level="info">
    <appender-ref ref="httpAccessLogAppender" />
</logger>
```

The pattern should only have the message marker, as other elements will be processed by the access logger.

Log Format

The syntax is based on [Apache httpd log format (http://httpd.apache.org/docs/current/mod/mod_log_config.html)](http://httpd.apache.org/docs/current/mod/mod_log_config.html).

These are the supported markers:

- **%a** - Remote IP address
- **%A** - Local IP address
- **%b** - Bytes sent, excluding HTTP headers, or '-' if no bytes were sent
- **%B** - Bytes sent, excluding HTTP headers
- **%h** - Remote host name
- **%H** - Request protocol
- **%{<header>}i** - Request header. If the argument is omitted (**%i**) all headers are printed
- **%{<header>}o** - Response header. If the argument is omitted (**%o**) all headers are printed

- **%{<cookie>}C** - Request cookie (COOKIE). If the argument is omitted (**%C**) all cookies are printed
- **%{<cookie>}c** - Response cookie (SET_COOKIE). If the argument is omitted (**%c**) all cookies are printed
- **%l** - Remote logical username from identd (always returns '-')
- **%m** - Request method
- **%p** - Local port
- **%q** - Query string (excluding the '?' character)
- **%r** - First line of the request
- **%s** - HTTP status code of the response
- **%{<format>}t** - Date and time. If the argument is omitted, Common Log Format is used ("'['dd/MMM/yyyy:HH:mm:ss Z']'").
  - If the format starts with begin: (default) the time is taken at the beginning of the request processing. If it starts with end: it is the time when the log entry gets written, close to the end of the request processing.
  - The format should follow `DateTimeFormatter` syntax.
- **%{property}u** - Remote authenticated user. When **micronaut-session** is on the classpath, returns the session id if the argument is omitted, or the specified property otherwise prints '-'
- **%U** - Requested URI
- **%v** - Local server name
- **%D** - Time taken to process the request, in milliseconds
- **%T** - Time taken to process the request, in seconds

In addition, you can use the following aliases for common patterns:

- **common** - `%h %l %u %t "%r" %s %b` for **Common Log Format** (https://httpd.apache.org/docs/1.3/logs.html#common) (CLF)
- **combined** - `%h %l %u %t "%r" %s %b "%{Referer}i" "%{User-Agent}i"` for **Combined Log Format** (https://httpd.apache.org/docs/1.3/logs.html#combined)

6.29.7 Starting Secondary Servers

Micronaut supports the programmatic creation of additional Netty servers through the **NettyEmbeddedServerFactory** interface.

This is useful in cases where you, for example, need to expose distinct servers over different ports with potentially differing configurations (HTTPS, thread resources etc.).

The following example demonstrates how to define a **Factory Bean** that starts an additional server using a programmatically created configuration:

Programmatically creating Secondary servers

Java	Groovy	Kotlin

JAVA

```
import java.util.Collections;
import io.micronaut.context.annotation.Bean;
import io.micronaut.context.annotation.Context;
import io.micronaut.context.annotation.Factory;
import io.micronaut.context.annotation.Requires;
import io.micronaut.context.env.Environment;
import io.micronaut.discovery.ServiceInstanceList;
import io.micronaut.discovery.StaticServiceInstanceList;
import io.micronaut.http.server.netty.NettyEmbeddedServer;
import io.micronaut.http.server.netty.NettyEmbeddedServerFactory;
import io.micronaut.http.server.netty.configuration.NettyHttpServerConfiguration;
import jakarta.inject.Named;

@Factory
public class SecondaryNettyServer {
    public static final String SERVER_ID = "another";  1

    @Named(SERVER_ID)
    @Context
    @Bean(preDestroy = "close")  2
    @Requires(beans = Environment.class)
    NettyEmbeddedServer nettyEmbeddedServer(NettyEmbeddedServerFactory serverFactory) {  3
        final NettyHttpServerConfiguration configuration =
                new NettyHttpServerConfiguration();  4
        // configure server programmatically
        final NettyEmbeddedServer embeddedServer = serverFactory.build(configuration);  5
        embeddedServer.start();  6
        return embeddedServer;  7
    }

    @Bean
    ServiceInstanceList serviceInstanceList(  8
            @Named(SERVER_ID) NettyEmbeddedServer nettyEmbeddedServer) {
        return new StaticServiceInstanceList(
                SERVER_ID,
                Collections.singleton(nettyEmbeddedServer.getURI())
        );
    }
}
```

Copy to Clipboard

1. Define a unique name for the server
2. Define a **@Context** scoped bean using the server name and including `preDestroy="close"` to ensure the server is shutdown when the context is closed
3. Inject the **NettyEmbeddedServerFactory** into a **Factory Bean**
4. Programmatically create the **NettyHttpServerConfiguration**
5. Use the `build` method to build the server instance
6. Start the server with the `start` method
7. Return the server instance as a managed bean
8. Optionally define an instance of **ServiceInstanceList** if you wish to inject **HTTP Clients** by the server name

With this class in place when the **ApplicationContext** starts the server will also be started with the appropriate configuration.

Thanks to the definition of the **ServiceInstanceList** in step 8, you can then inject a client into your tests to test the secondary server:

Injecting the server or client

Java	Groovy	Kotlin

JAVA

```
@Client(path = "/", id = SecondaryNettyServer.SERVER_ID)
@Inject
HttpClient httpClient;  1

@Named(SecondaryNettyServer.SERVER_ID)
EmbeddedServer embeddedServer;  2
```

Copy to Clipboard

1. Use the server name to inject a client by ID
2. Use the `@Named` annotation as a qualifier to inject the embedded server instance

6.30 Server Side View Rendering

Micronaut supports Server Side View Rendering.

See the documentation for **Micronaut Views** (https://micronaut-projects.github.io/micronaut-views/latest/guide) for more information.

6.31 OpenAPI / Swagger Support

To configure Micronaut integration with **OpenAPI/Swagger** (https://swagger.io/docs/specification/about/), please follow **these instructions** (https://micronaut-projects.github.io/micronaut-openapi/latest/guide/index.html)

6.32 GraphQL Support

GraphQL (https://graphql.org/) is a query language for building APIs that provides an intuitive and flexible syntax and a system for describing data requirements and interactions.

See the documentation for **Micronaut GraphQL** (https://micronaut-projects.github.io/micronaut-graphql/latest/guide) for more information on how to build GraphQL applications with Micronaut.

7 The HTTP Client

Using the CLI

If you create your project using the Micronaut CLI, the `http-client` dependency is included by default.

Client communication between Microservices is a critical component of any Microservice architecture. With that in mind, Micronaut includes an HTTP client that has both a low-level API and a higher-level AOP-driven API.

Regardless whether you choose to use Micronaut's HTTP server, you may wish to use the Micronaut HTTP client in your application since it is a feature-rich client implementation.

To use the HTTP client, add the `http-client` dependency to your build:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut</groupId>
    <artifactId>micronaut-http-client</artifactId>
</dependency>
```

Copy to Clipboard

Since the higher level API is built on the low-level HTTP client, we first introduce the low-level client.

7.1 Using the Low-Level HTTP Client

The **HttpClient** interface forms the basis for the low-level API. This interfaces declares methods to help ease executing HTTP requests and receiving responses.

The majority of the methods in the **HttpClient** interface return Reactive Streams **Publisher** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) instances, which is not always the most useful interface to work against.

Micronaut's Reactor HTTP Client dependency ships with a sub-interface named **ReactorHttpClient** (https://micronaut-projects.github.io/micronaut-reactor/latest/api/io/micronaut/reactor/client/ReactorHttpClient). It provides a variation of the **HttpClient** interface that returns **Project Reactor** (https://projectreactor.io) **Flux** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html) types.

7.1.1 Sending your first HTTP request

Obtaining a HttpClient

There are a few ways to obtain a reference to an **HttpClient**. The most common is to use the **Client** annotation. For example:

Injecting an HTTP client

JAVA

```
@Client("https://api.twitter.com/1.1") @Inject HttpClient httpClient;
```

The above example injects a client that targets the Twitter API.

KOTLIN

```
@field:Client("\${myapp.api.twitter.url}") @Inject lateinit var httpClient: HttpClient
```

The above Kotlin example injects a client that targets the Twitter API using a configuration path. Note the required escaping (backslash) on `"\${path.to.config}"` which is necessary due to Kotlin string interpolation.

The **Client** annotation is also a custom scope that manages the creation of **HttpClient** instances and ensures they are stopped when the application shuts down.

The value you pass to the **Client** annotation can be one of the following:

- An absolute URI, e.g. **`https://api.twitter.com/1.1`**

- A relative URI, in which case the targeted server will be the current server (useful for testing)
- A service identifier. See the section on **Service Discovery** for more information on this topic.

Another way to create an `HttpClient` is with the static `create` method of **HttpClient**, however this approach is not recommended as you must ensure you manually shutdown the client, and of course no dependency injection will occur for the created client.

Performing an HTTP GET

Generally there are two methods of interest when working with the `HttpClient`. The first is `retrieve`, which executes an HTTP request and returns the body in whichever type you request (by default a `String`) as `Publisher`.

The `retrieve` method accepts an **HttpRequest** or a `String` URI to the endpoint you wish to request.

The following example shows how to use `retrieve` to execute an HTTP `GET` and receive the response body as a `String`:

Using retrieve

Java | Groovy | Kotlin

JAVA

```
String uri = UriBuilder.of("/hello/{name}")
                       .expand(Collections.singletonMap("name", "John"))
                       .toString();
assertEquals("/hello/John", uri);

String result = client.toBlocking().retrieve(uri);

assertEquals("Hello John", result);
```

Copy to Clipboard

Note that in this example, for illustration purposes we call `toBlocking()` to return a blocking version of the client. However, in production code you should not do this and instead rely on the non-blocking nature of the Micronaut HTTP server.

For example the following `@Controller` method calls another endpoint in a non-blocking manner:

Using the HTTP client without blocking

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.annotation.Body;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.annotation.Post;
import io.micronaut.http.annotation.Status;
import io.micronaut.http.client.HttpClient;
import io.micronaut.http.client.annotation.Client;
import org.reactivestreams.Publisher;
import reactor.core.publisher.Mono;
import io.micronaut.core.async.annotation.SingleResult;
import static io.micronaut.http.HttpRequest.GET;
import static io.micronaut.http.HttpStatus.CREATED;
import static io.micronaut.http.MediaType.TEXT_PLAIN;

    @Get("/hello/{name}")
    @SingleResult
    Publisher<String> hello(String name) {  1
        return Mono.from(httpClient.retrieve(GET("/hello/" + name)));  2
    }
```

Copy to Clipboard

1. The `hello` method returns a **Mono (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html)** which may or may not emit an item. If an item is not emitted, a 404 is returned.
2. The `retrieve` method is called which returns a **Flux (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html)**. This has a `firstElement` method that returns the first emitted item or nothing

Using Reactor (or RxJava if you prefer) you can easily and efficiently compose multiple HTTP client calls without blocking (which limits the throughput and scalability of your application).

Debugging / Tracing the HTTP Client

To debug requests being sent and received from the HTTP client you can enable tracing logging via your `logback.xml` file:

logback.xml

XML

```
<logger name="io.micronaut.http.client" level="TRACE"/>
```

Client Specific Debugging / Tracing

To enable client-specific logging you can configure the default logger for all HTTP clients. You can also configure different loggers for different clients using **Client-Specific Configuration**. For example, in `application.yml`:

application.yml

```
micronaut:
  http:
    client:
      logger-name: mylogger
    services:
      otherClient:
        logger-name: other.client
```

XML

Then enable logging in `logback.xml`:

logback.xml

```
<logger name="mylogger" level="DEBUG"/>
<logger name="other.client" level="TRACE"/>
```

XML

Customizing the HTTP Request

The previous example demonstrates using the static methods of the **HttpRequest** interface to construct a **MutableHttpRequest** instance. Like the name suggests, a `MutableHttpRequest` can be mutated, including the ability to add headers, customize the request body, etc. For example:

Passing an HttpRequest to retrieve

Java | Groovy | Kotlin

```
Flux<String> response = Flux.from(client.retrieve(
        GET("/hello/John")
        .header("X-My-Header", "SomeValue")
));
```

JAVA

Copy to Clipboard

The above example adds a header (`X-My-Header`) to the response before it is sent. The **MutableHttpRequest** interface has more convenience methods that make it easy to modify the request in common ways.

Reading JSON Responses

Microservices typically use a message encoding format such as JSON. Micronaut's HTTP client leverages Jackson for JSON parsing, hence any type Jackson can decode can be passed as a second argument to `retrieve`.

For example consider the following `@Controller` method that returns a JSON response:

Returning JSON from a controller

Java | Groovy | Kotlin

```
@Get("/greet/{name}")
Message greet(String name) {
    return new Message("Hello " + name);
}
```

JAVA

Copy to Clipboard

The method above returns a POJO of type `Message` which looks like:

Message POJO

Java | Groovy | Kotlin

```
import com.fasterxml.jackson.annotation.JsonCreator;
import com.fasterxml.jackson.annotation.JsonProperty;

public class Message {

    private final String text;

    @JsonCreator
    public Message(@JsonProperty("text") String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }
}
```

JAVA

Jackson annotations are used to map the constructor

Copy to Clipboard

On the client you can call this endpoint and decode the JSON into a map using the `retrieve` method as follows:

Decoding the response body to a Map

Java | Groovy | Kotlin

JAVA

```
Flux<Map> response = Flux.from(client.retrieve(
        GET("/greet/John"), Map.class
));
```

Copy to Clipboard

The above examples decodes the response into a **Map (https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)** representing the JSON. You can use the `Argument.of(..)` method to customize the type of the key and string:

Decoding the response body to a Map

Java | Groovy | Kotlin

JAVA

```
response = Flux.from(client.retrieve(
        GET("/greet/John"),
        Argument.of(Map.class, String.class, String.class)  1
));
```

Copy to Clipboard

1 The `Argument.of` method returns a `Map` where the key and value types are `String`

Whilst retrieving JSON as a map can be desirable, typically you want to decode objects into POJOs. To do that, pass the type instead:

Decoding the response body to a POJO

Java | Groovy | Kotlin

JAVA

```
Flux<Message> response = Flux.from(client.retrieve(
        GET("/greet/John"), Message.class
));

assertEquals("Hello John", response.blockFirst().getText());
```

Copy to Clipboard

Note how you can use the same Java type on both the client and the server. The implication of this is that typically you define a common API project where you define the interfaces and types that define your API.

Decoding Other Content Types

If the server you communicate with uses a custom content type that is not JSON, by default Micronaut's HTTP client will not know how to decode this type.

To resolve this, register **MediaTypeCodec** as a bean, and it will be automatically picked up and used to decode (or encode) messages.

Receiving the Full HTTP Response

Sometimes receiving just the body of the response is not enough, and you need other information from the response such as headers, cookies, etc. In this case, instead of `retrieve` use the `exchange` method:

Receiving the Full HTTP Response

Java | Groovy | Kotlin

JAVA

```
Flux<HttpResponse<Message>> call = Flux.from(client.exchange(
        GET("/greet/John"), Message.class  1
));

HttpResponse<Message> response = call.blockFirst();
Optional<Message> message = response.getBody(Message.class);  2
// check the status
assertEquals(HttpStatus.OK, response.getStatus());  3
// check the body
assertTrue(message.isPresent());
assertEquals("Hello John", message.get().getText());
```

Copy to Clipboard

1 The `exchange` method receives the **HttpResponse**

2 The body is retrieved using the `getBody(..)` method of the response

3 Other aspects of the response such as the **HttpStatus** can be checked

The above example receives the full **HttpResponse** from which you can obtain headers and other useful information.

7.1.2 Posting a Request Body

All the examples so far have used the same HTTP method i.e `GET`. The **HttpRequest** interface has factory methods for all the different HTTP methods. The following table summarizes them:

Table 1. HttpRequest Factory Methods

Method	Description	Allows Body
HttpRequest.GET(java.lang.String)	Constructs an HTTP `GET` request	`false`
HttpRequest.OPTIONS(java.lang.String)	Constructs an HTTP `OPTIONS` request	`false`
HttpRequest.HEAD(java.lang.String)	Constructs an HTTP `HEAD` request	`false`
HttpRequest.POST(java.lang.String,T)	Constructs an HTTP `POST` request	`true`
HttpRequest.PUT(java.lang.String,T)	Constructs an HTTP `PUT` request	`true`
HttpRequest.PATCH(java.lang.String,T)	Constructs an HTTP `PATCH` request	`true`
HttpRequest.DELETE(java.lang.String)	Constructs an HTTP `DELETE` request	`true`

A `create` method also exists to construct a request for any **HttpMethod** type. Since the `POST`, `PUT` and `PATCH` methods require a body, a second argument which is the body object is required.

The following example demonstrates how to send a simple `String` body:

Sending a `String` body

Java | Groovy | Kotlin

JAVA

```
Flux<HttpResponse<String>> call = Flux.from(client.exchange(
        POST("/hello", "Hello John")  1
            .contentType(MediaType.TEXT_PLAIN_TYPE)
            .accept(MediaType.TEXT_PLAIN_TYPE),  2
        String.class  3
));
```

Copy to Clipboard

1. The `POST` method is used; the first argument is the URI and the second is the body
2. The content type and accepted type are set to `text/plain` (the default is `application/json`)
3. The expected response type is a `String`

Sending JSON

The previous example sends plain text. To send JSON, pass the object to encode to JSON (whether that be a Map or a POJO) as long as Jackson is able to encode it.

For example, you can create a `Message` from the previous section and pass it to the `POST` method:

Sending a JSON body

Java | Groovy | Kotlin

JAVA

```
Flux<HttpResponse<Message>> call = Flux.from(client.exchange(
        POST("/greet", new Message("Hello John")),  1
        Message.class  2
));
```

Copy to Clipboard

1. An instance of `Message` is created and passed to the `POST` method
2. The same class decodes the response

With the above example the following JSON is sent as the body of the request:

Resulting JSON

JSON

```
{"text":"Hello John"}
```

The JSON can be customized using **Jackson Annotations** (https://github.com/FasterXML/jackson-annotations).

Using a URI Template

If include some properties of the object in the URI, you can use a URI template.

For example imagine you have a `Book` class with a `title` property. You can include the `title` in the URI template and then populate it from an instance of `Book`. For example:

Sending a JSON body with a URI template

Java | Groovy | Kotlin

JAVA

```
Flux<HttpResponse<Book>> call = Flux.from(client.exchange(
        POST("/amazon/book/{title}", new Book("The Stand")),
        Book.class
));
```

Copy to Clipboard

In the above case the `title` property is included in the URI.

Sending Form Data

You can also encode a POJO or a map as form data instead of JSON. Just set the content type to `application/x-www-form-urlencoded` on the post request:

Sending a Form Data

Java | Groovy | Kotlin

JAVA

```
Flux<HttpResponse<Book>> call = Flux.from(client.exchange(
        POST("/amazon/book/{title}", new Book("The Stand"))
        .contentType(MediaType.APPLICATION_FORM_URLENCODED),
        Book.class
));
```

Copy to Clipboard

Note that Jackson can bind form data too, so to customize the binding process use [Jackson annotations (https://github.com/FasterXML/jackson-annotations)](https://github.com/FasterXML/jackson-annotations).

7.1.3 Multipart Client Uploads

The Micronaut HTTP Client supports multipart requests. To build a multipart request, set the content type to `multipart/form-data` and set the body to an instance of MultipartBody.

For example:

Creating the body

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.client.multipart.MultipartBody;

        String toWrite = "test file";
        File file = File.createTempFile("data", ".txt");
        FileWriter writer = new FileWriter(file);
        writer.write(toWrite);
        writer.close();

        MultipartBody requestBody = MultipartBody.builder()      1
                .addPart(                                        2
                    "data",
                    file.getName(),
                    MediaType.TEXT_PLAIN_TYPE,
                    file
                ).build();                                       3
```

Copy to Clipboard

1. Create a MultipartBody builder for adding parts to the body.
2. Add a part to the body, in this case a file. There are different variations of this method in MultipartBody.Builder.
3. The build method assembles all parts from the builder into a MultipartBody. At least one part is required.

Creating a request

Java | Groovy | Kotlin

JAVA

```
HttpRequest.POST("/multipart/upload", requestBody)      1
           .contentType(MediaType.MULTIPART_FORM_DATA_TYPE)  2
```

Copy to Clipboard

1. The multipart request body with different types of data.
2. Set the content-type header of the request to `multipart/form-data`.

7.1.4 Streaming JSON over HTTP

Micronaut's HTTP client includes support for streaming data over HTTP via the **ReactorStreamingHttpClient** (https://micronaut-projects.github.io/micronaut-reactor/latest/api/io/micronaut/reactor/client/ReactorStreamingHttpClient) interface which includes methods specific to streaming including:

Table 1. HTTP Streaming Methods

Method	Description
`dataStream(HttpRequest<I> request)`	Returns a stream of data as a **Flux** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html) of **ByteBuffer**
`exchangeStream(HttpRequest<I> request)`	Returns the **HttpResponse** wrapping a **Flux** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html) of **ByteBuffer**
`jsonStream(HttpRequest<I> request)`	Returns a non-blocking stream of JSON objects

To use JSON streaming, declare a controller method on the server that returns a `application/x-json-stream` of JSON objects. For example:

Streaming JSON on the Server

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.MediaType;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import org.reactivestreams.Publisher;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
import java.time.Duration;
import java.time.ZonedDateTime;
import java.time.temporal.ChronoUnit;

    @Get(value = "/headlines", processes = MediaType.APPLICATION_JSON_STREAM)  1
    Publisher<Headline> streamHeadlines() {
        return Mono.fromCallable(() -> {   2
            Headline headline = new Headline();
            headline.setText("Latest Headline at " + ZonedDateTime.now());
            return headline;
        }).repeat(100)  3
          .delayElements(Duration.of(1, ChronoUnit.SECONDS));  4
    }
```

Copy to Clipboard

1. The `streamHeadlines` method produces `application/x-json-stream`
2. A **Flux** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html) is created from a `Callable` function (note no blocking occurs within the function, so this is ok, otherwise you should `subscribeOn` an I/O thread pool).
3. The **Flux** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html) repeats 100 times
4. The **Flux** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html) emits items with a delay of one second between each

The server does not have to be written in Micronaut, any server that supports JSON streaming will do.

Then on the client, subscribe to the stream using `jsonStream` and every time the server emits a JSON object the client will decode and consume it:

Streaming JSON on the Client

Java | Groovy | Kotlin

JAVA

```
Flux<Headline> headlineStream = Flux.from(client.jsonStream(
        GET("/streaming/headlines"), Headline.class));  1
CompletableFuture<Headline> future = new CompletableFuture<>();  2
headlineStream.subscribe(new Subscriber<Headline>() {
    @Override
    public void onSubscribe(Subscription s) {
        s.request(1);  3
    }

    @Override
    public void onNext(Headline headline) {
        System.out.println("Received Headline = " + headline.getText());
        future.complete(headline);  4
    }

    @Override
    public void onError(Throwable t) {
        future.completeExceptionally(t);  5
    }

    @Override
    public void onComplete() {
        // no-op  6
    }
});
```

Copy to Clipboard

1. The `jsonStream` method returns a **Flux** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html)
2. A `CompletableFuture` is used to receive a value, but what you do with each emitted item is application-specific
3. The **Subscription** (https://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Subscription.html) requests a single item. You can use the **Subscription** (https://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Subscription.html) to regulate back pressure and demand.
4. The `onNext` method is called when an item is emitted
5. The `onError` method is called when an error occurs
6. The `onComplete` method is called when all `Headline` instances have been emitted

Note neither the server nor the client in the example above perform any blocking I/O.

7.1.5 Configuring HTTP clients

Global Configuration for All Clients

The default HTTP client configuration is a **Configuration Properties** named **DefaultHttpClientConfiguration** that allows configuring the default behaviour for all HTTP clients. For example, in `application.yml`:

Altering default HTTP client configuration

YAML

```
micronaut:
  http:
    client:
      read-timeout: 5s
```

The above example sets the `readTimeout` property of the **HttpClientConfiguration** class.

Client Specific Configuration

To have separate configuration per-client, there are a couple of options. You can configure **Service Discovery** manually in `application.yml` and apply per-client configuration:

Manually configuring HTTP services

YAML

```
micronaut:
  http:
    services:
      foo:
        urls:
          - http://foo1
          - http://foo2
        read-timeout: 5s  1
```

1. The read timeout is applied to the `foo` client.

WARN: This client configuration can be used in conjunction with the `@Client` annotation, either by injecting an `HttpClient` directly or use on a client interface. In any case, all other attributes on the annotation **will be ignored** except the service id.

Then, inject the named client configuration:

Injecting an HTTP client

JAVA

```
@Client("foo") @Inject ReactorHttpClient httpClient;
```

You can also define a bean that extends from **HttpClientConfiguration** and ensure that the `javax.inject.Named` annotation names it appropriately:

Defining an HTTP client configuration bean

JAVA

```
@Named("twitter")
@Singleton
class TwitterHttpClientConfiguration extends HttpClientConfiguration {
   public TwitterHttpClientConfiguration(ApplicationConfiguration configuration) {
        super(configuration);
    }
}
```

This configuration will be picked up if you inject a service named `twitter` with `@Client` using **Service Discovery**:

Injecting an HTTP client

JAVA

```
@Client("twitter") @Inject ReactorHttpClient httpClient;
```

Alternatively, if you don't use service discovery you can use the `configuration` member of `@Client` to refer to a specific type:

Injecting an HTTP client

JAVA

```
@Client(value = "https://api.twitter.com/1.1",
        configuration = TwitterHttpClientConfiguration.class)
@Inject
ReactorHttpClient httpClient;
```

Using HTTP Client Connection Pooling

A client that handles a significant number of requests will benefit from enabling HTTP client connection pooling. The following configuration enables pooling for the `foo` client:

Manually configuring HTTP services

YAML

```
micronaut:
  http:
    services:
      foo:
        urls:
          - http://foo1
          - http://foo2
        pool:
          enabled: true  1
          max-connections: 50  2
```

1 Enables the pool

2 Sets the maximum number of connections in the pool

See the API for **ConnectionPoolConfiguration** for details on available pool configuration options.

Configuring Event Loop Groups

By default, Micronaut shares a common Netty `EventLoopGroup` for worker threads and all HTTP client threads.

This `EventLoopGroup` can be configured via the `micronaut.netty.event-loops.default` property:

Configuring The Default Event Loop

YAML

```
micronaut:
  netty:
    event-loops:
      default:
        num-threads: 10
        prefer-native-transport: true
```

You can also use the `micronaut.netty.event-loops` setting to configure one or more additional event loops. The following table summarizes the properties:

Unresolved directive in <stdin> - include::/home/runner/work/micronaut-core/micronaut-core/build/generated/configurationProperties/io.micronaut.http.netty.channel.DefaultEventLoopGroupConfiguration.adoc[]

For example, if your interactions with an HTTP client involve CPU-intensive work, it may be worthwhile configuring a separate `EventLoopGroup` for one or all clients.

The following example configures an additional event loop group called "other" with 10 threads:

Configuring Additional Event Loops

YAML

```yaml
micronaut:
  netty:
    event-loops:
      other:
        num-threads: 10
        prefer-native-transport: true
```

Once an additional event loop has been configured you can alter the HTTP client configuration to use it:

Altering the Event Loop Group used by Clients

YAML

```yaml
micronaut:
  http:
    client:
      event-loop-group: other
```

7.1.6 Error Responses

If an HTTP response is returned with a code of 400 or higher, an **HttpClientResponseException** is created. The exception contains the original response. How that exception gets thrown depends on the return type of the method.

For blocking clients, the exception is thrown and should be caught and handled by the caller. For reactive clients, the exception is passed through the publisher as an error.

7.1.7 Bind Errors

Often you want to consume an endpoint and bind to a POJO if the request is successful and bind to a different POJO if an error occurs. The following example shows how to invoke `exchange` with a success and error type.

Java | Groovy | Kotlin

JAVA

```java
@Controller("/books")
public class BooksController {

    @Get("/{isbn}")
    public HttpResponse find(String isbn) {
        if (isbn.equals("1680502395")) {
            Map<String, Object> m = new HashMap<>();
            m.put("status", 401);
            m.put("error", "Unauthorized");
            m.put("message", "No message available");
            m.put("path", "/books/" + isbn);
            return HttpResponse.status(HttpStatus.UNAUTHORIZED).body(m);
        }

        return HttpResponse.ok(new Book("1491950358", "Building Microservices"));
    }
}
```

Copy to Clipboard

Java | Groovy | Kotlin

JAVA

```java
@Test
public void afterAnHttpClientExceptionTheResponseBodyCanBeBoundToAPOJO() {
    try {
        client.toBlocking().exchange(HttpRequest.GET("/books/1680502395"),
                Argument.of(Book.class),  1
                Argument.of(CustomError.class));  2
    } catch (HttpClientResponseException e) {
        assertEquals(HttpStatus.UNAUTHORIZED, e.getResponse().getStatus());
        Optional<CustomError> jsonError = e.getResponse().getBody(CustomError.class);
        assertTrue(jsonError.isPresent());
        assertEquals(401, jsonError.get().status);
        assertEquals("Unauthorized", jsonError.get().error);
        assertEquals("No message available", jsonError.get().message);
        assertEquals("/books/1680502395", jsonError.get().path);
    }
}
```

Copy to Clipboard

1 Success Type

2 Error Type

7.2 Proxying Requests with ProxyHttpClient

A common requirement in Microservice environments is to proxy requests in a Gateway Microservice to other backend Microservices.

The regular **HttpClient** API is designed around simplifying message exchange and is not designed for proxying requests. For this case, use the **ProxyHttpClient**, which can be used from a **HTTP Server Filter** to proxy requests to backend Microservices.

The following example demonstrates rewriting requests under the URI `/proxy` to the URI `/real` onto the same server (although in a real environment you generally proxy to another server):

Customizing the Netty pipeline for Logbook

Java	Groovy	Kotlin

JAVA

```
import io.micronaut.core.async.publisher.Publishers;
import io.micronaut.core.util.StringUtils;
import io.micronaut.http.HttpRequest;
import io.micronaut.http.MutableHttpResponse;
import io.micronaut.http.annotation.Filter;
import io.micronaut.http.client.ProxyHttpClient;
import io.micronaut.http.filter.HttpServerFilter;
import io.micronaut.http.filter.OncePerRequestHttpServerFilter;
import io.micronaut.http.filter.ServerFilterChain;
import io.micronaut.runtime.server.EmbeddedServer;
import org.reactivestreams.Publisher;

@Filter("/proxy/**")
public class ProxyFilter implements HttpServerFilter {  1

    private final ProxyHttpClient client;
    private final EmbeddedServer embeddedServer;

    public ProxyFilter(ProxyHttpClient client,
                       EmbeddedServer embeddedServer) {  2
        this.client = client;
        this.embeddedServer = embeddedServer;
    }

    @Override
    public Publisher<MutableHttpResponse<?>> doFilter(HttpRequest<?> request,
                                                      ServerFilterChain chain) {
        return Publishers.map(client.proxy(  3
                request.mutate()  4
                        .uri(b -> b  5
                                .scheme("http")
                                .host(embeddedServer.getHost())
                                .port(embeddedServer.getPort())
                                .replacePath(StringUtils.prependUri(
                                        "/real",
                                        request.getPath().substring("/proxy".length())
                                ))
                        )
                        .header("X-My-Request-Header", "XXX")  6
        ), response -> response.header("X-My-Response-Header", "YYY"));
    }
}
```

Copy to Clipboard

1 The filter extends **HttpServerFilter**

2 The **ProxyHttpClient** is injected into the constructor.

3 The `proxy` method proxies the request

4 The request is mutated to modify the URI and include an additional header

5 The **UriBuilder** API rewrites the URI

6 Additional request and response headers are included

The **ProxyHttpClient** API is a low-level API that can be used to build a higher-level abstraction such as an API Gateway.

7.3 Declarative HTTP Clients with @Client

Now that you have an understanding of the workings of the lower-level HTTP client, let's take a look at Micronaut's support for declarative clients via the **Client** annotation.

Essentially, the `@Client` annotation can be declared on any interface or abstract class, and through the use of **Introduction Advice** the abstract methods are implemented for you at compile time, greatly simplifying the creation of HTTP clients.

Let's start with a simple example. Given the following class:

Pet.java

Java	Groovy	Kotlin

JAVA

```
public class Pet {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

Copy to Clipboard

You can define a common interface for saving new `Pet` instances:

PetOperations.java

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.annotation.Post;
import io.micronaut.validation.Validated;
import org.reactivestreams.Publisher;
import io.micronaut.core.async.annotation.SingleResult;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotBlank;

@Validated
public interface PetOperations {
    @Post
    @SingleResult
    Publisher<Pet> save(@NotBlank String name, @Min(1L) int age);
}
```

Copy to Clipboard

Note how the interface uses Micronaut's HTTP annotations which are usable on both the server and client side. You can also use `javax.validation` constraints to validate arguments.

Be aware that some annotations, such as **Produces** and **Consumes**, have different semantics between server and client side usage. For example, `@Produces` on a controller method (server side) indicates how the method's **return value** is formatted, while `@Produces` on a client indicates how the method's **parameters** are formatted when sent to the server. While this may seem a little confusing, it is logical considering the different semantics between a server producing/consuming vs a client: a server consumes an argument and **returns** a response to the client, whereas a client consumes an argument and **sends** output to a server.

Additionally, to use the `javax.validation` features, add the `validation` module to your build:

Gradle | **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut</groupId>
    <artifactId>micronaut-validator</artifactId>
</dependency>
```

Copy to Clipboard

On the server-side of Micronaut you can implement the `PetOperations` interface:

PetController.java

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.annotation.Controller;
import org.reactivestreams.Publisher;
import reactor.core.publisher.Mono;
import io.micronaut.core.async.annotation.SingleResult;

@Controller("/pets")
public class PetController implements PetOperations {

    @Override
    @SingleResult
    public Publisher<Pet> save(String name, int age) {
        Pet pet = new Pet();
        pet.setName(name);
        pet.setAge(age);
        // save to database or something
        return Mono.just(pet);
    }
}
```

Copy to Clipboard

You can then define a declarative client in `src/test/java` that uses `@Client` to automatically implement a client at compile time:

PetClient.java

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.client.annotation.Client;
import org.reactivestreams.Publisher;
import io.micronaut.core.async.annotation.SingleResult;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotBlank;

@Client("/pets")  1
public interface PetClient extends PetOperations {  2

    @Override
    @SingleResult
    Publisher<Pet> save(@NotBlank String name, @Min(1L) int age);  3
}
```

Copy to Clipboard

1. The **Client** annotation is used with a value relative to the current server, in this case `/pets`
2. The interface extends from `PetOperations`
3. The `save` method is overridden. See the warning below.

Notice in the above example we override the `save` method. This is necessary if you compile without the `-parameters` option since Java does not retain parameters names in bytecode otherwise. Overriding is not necessary if you compile with `-parameters`. In addition, when overriding methods you should ensure any validation annotations are declared again since these are not **Inherited** **(https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Inherited.html)** annotations.

Once you have defined a client you can `@Inject` it wherever you need it.

Recall that the value of `@Client` can be:

- An absolute URI, e.g. **`https://api.twitter.com/1.1`**
- A relative URI, in which case the server targeted is the current server (useful for testing)
- A service identifier. See the section on **Service Discovery** for more information on this topic.

In production you typically use a service ID and **Service Discovery** to discover services automatically.

Another important thing to notice regarding the `save` method in the example above is that it returns a **Single** **(http://reactivex.io/RxJava/2.x/javadoc/io/reactivex/Single.html)** type.

This is a non-blocking reactive type - typically you want your HTTP clients to not block. There are cases where you may want an HTTP client that does block (such as in unit tests), but this is rare.

The following table illustrates common return types usable with **@Client**:

Table 1. Micronaut Response Types

Type	Description	Example Signature

Type	Description	Example Signature
Publisher (https://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html)	Any type that implements the Publisher (https://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) interface	`Flux<String> hello()`
HttpResponse	An HttpResponse and optional response body type	`Mono<HttpResponse<String>> hello()`
Publisher (https://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html)	A Publisher (https://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) implementation that emits a POJO	`Mono<Book> hello()`
CompletableFuture (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html)	A Java `CompletableFuture` instance	`CompletableFuture<String> hello()`
CharSequence (https://docs.oracle.com/javase/8/docs/api/java/lang/CharSequence.html)	A blocking native type. Such as `String`	`String hello()`
T	Any simple POJO type.	`Book show()`

Generally, any reactive type that can be converted to the Publisher (https://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) interface is supported as a return type, including (but not limited to) the reactive types defined by RxJava 1.x, RxJava 2.x, and Reactor 3.x.

Returning CompletableFuture (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html) instances is also supported. Note that returning any other type *results in a blocking request* and is not recommended other than for testing.

7.3.1 Customizing Parameter Binding

The previous example presented a simple example using method parameters to represent the body of a `POST` request:

PetOperations.java

JAVA

```
@Post
@SingleResult
Publisher<Pet> save(@NotBlank String name, @Min(1L) int age);
```

The `save` method performs an HTTP `POST` with the following JSON by default:

Example Produced JSON

JSON

```
{"name":"Dino", "age":10}
```

You may however want to customize what is sent as the body, the parameters, URI variables, etc. The @Client annotation is very flexible in this regard and supports the same HTTP Annotations as Micronaut's HTTP server.

For example, the following defines a URI template, and the `name` parameter is used as part of the URI template, whilst @Body declares that the contents to send to the server are represented by the `Pet` POJO:

PetOperations.java

JAVA

```
@Post("/{name}")
Mono<Pet> save(
    @NotBlank String name,  1
    @Body @Valid Pet pet)  2
```

1. The name parameter, included as part of the URI, and declared `@NotBlank`
2. The `pet` parameter, used to encode the body and declared `@Valid`

The following table summarizes the parameter annotations and their purpose, and provides an example:

Table 1. Parameter Binding Annotations

Annotation	Description	Example
@Body	Specifies the parameter for the body of the request	`@Body String body`
@CookieValue	Specifies parameters to be sent as cookies	`@CookieValue String myCookie`
@Header	Specifies parameters to be sent as HTTP headers	`@Header String requestId`

Annotation	Description	Example
@QueryValue	Customizes the name of the URI parameter to bind from	`@QueryValue("userAge") Integer age`
@PathVariable	Binds a parameter exclusively from a Path Variable.	`@PathVariable Long id`
@RequestAttribute	Specifies parameters to be set as request attributes	`@RequestAttribute Integer locationId`

Always use @Produces or @Consumes instead of supplying a header for `Content-Type` or `Accept`.

Query values formatting

The Format annotation can be used together with `@QueryValue` annotation to format query values.

The supported values are: `"csv"`, `"ssv"`, `"pipes"`, `"multi"` and `"deep-object"`, where the meaning is similar to Open API v3 (https://swagger.io/docs/specification/serialization/) query parameter's `style` attribute.

The format can only be applied to `java.lang.Iterable`, `java.util.Map` or POJO with Introspected annotation. Examples of how different values will be formatted are given in the table below:

Format	Iterable example	Map or POJO example
Original value	`["Mike", "Adam", "Kate"]`	`{"name": "Mike", "age": 30"}`
`"CSV"`	`"param=Mike,Adam,Kate"`	`"param=name,Mike,age,30"`
`"SSV"`	`"param=Mike Adam Kate"`	`"param=name Mike age 30"`
`"PIPES"`	`"param=Mike|Adam|Kate"`	`"param=name|Mike|age|30"`
`"MULTI"`	`"param=Mike&param=Adam&param=Kate"`	`"name=Mike&age=30"`
`"DEEP_OBJECT"`	`"param[0]=Mike&param[1]=Adam&param[2]=Kate"`	`"param[name]=Mike&param[age]=30"`

Type-Based Binding Parameters

Some parameters are recognized by their type instead of their annotation. The following table summarizes these parameter types and their purpose, and provides an example:

Type	Description	Example
BasicAuth	Binds Basic Authorization credentials	`BasicAuth basicAuth`

Custom Binding

The ClientArgumentRequestBinder API binds client arguments to the request. Custom binder classes registered as beans are automatically used during the binding process. Annotation-based binders are searched for first, with type-based binders being searched if a binder was not found.

This is an experimental feature in 2.1 and subject to change! One limitation of binding is that binders may not manipulate or set the request URI because it can be a combination of many arguments.

Binding By Annotation

To control how an argument is bound to the request based on an annotation on the argument, create a bean of type AnnotatedClientArgumentRequestBinder. Any custom annotations must be annotated with @Bindable.

In this example, see the following client:

Client With @Metadata Argument

Java | Groovy | Kotlin

JAVA

```
@Client("/")
public interface MetadataClient {

    @Get("/client/bind")
    String get(@Metadata Map<String, Object> metadata);
}
```

Copy to Clipboard

The argument is annotated with the following annotation:

@Metadata Annotation

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.core.bind.annotation.Bindable;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Documented
@Retention(RUNTIME)
@Target(PARAMETER)
@Bindable
public @interface Metadata {
}
```

Copy to Clipboard

Without any additional code, the client attempts to convert the metadata to a string and append it as a query parameter. In this case that isn't the desired effect, so a custom binder is needed.

The following binder handles arguments passed to clients that are annotated with the `@Metadata` annotation, and mutate the request to contain the desired headers. The implementation could be modified to accept more types of data other than `Map`.

Annotation Argument Binder

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.core.annotation.NonNull;
import io.micronaut.core.convert.ArgumentConversionContext;
import io.micronaut.core.naming.NameUtils;
import io.micronaut.core.util.StringUtils;
import io.micronaut.http.MutableHttpRequest;
import io.micronaut.http.client.bind.AnnotatedClientArgumentRequestBinder;
import io.micronaut.http.client.bind.ClientRequestUriContext;
import org.jetbrains.annotations.NotNull;

import jakarta.inject.Singleton;
import java.util.Map;

@Singleton
public class MetadataClientArgumentBinder implements AnnotatedClientArgumentRequestBinder<Metadata> {

    @NotNull
    @Override
    public Class<Metadata> getAnnotationType() {
        return Metadata.class;
    }

    @Override
    public void bind(@NotNull ArgumentConversionContext<Object> context,
                     @NonNull ClientRequestUriContext uriContext,
                     @NotNull Object value,
                     @NotNull MutableHttpRequest<?> request) {
        if (value instanceof Map) {
            for (Map.Entry<?, ?> entry: ((Map<?, ?>) value).entrySet()) {
                String key = NameUtils.hyphenate(StringUtils.capitalize(entry.getKey().toString()), false);
                request.header("X-Metadata-" + key, entry.getValue().toString());
            }
        }
    }
}
```

Copy to Clipboard

Binding By Type

To bind to the request based on the type of the argument, create a bean of type [TypedClientArgumentRequestBinder].

In this example, see the following client:

Client With Metadata Argument

Java | Groovy | Kotlin

```java
@Client("/")
public interface MetadataClient {

    @Get("/client/bind")
    String get(Metadata metadata);
}
```

Without any additional code, the client attempts to convert the metadata to a string and append it as a query parameter. In this case that isn't the desired effect, so a custom binder is needed.

The following binder handles arguments passed to clients of type `Metadata` and mutate the request to contain the desired headers.

Typed Argument Binder

Java | Groovy | Kotlin

```java
import io.micronaut.core.annotation.NonNull;
import io.micronaut.core.convert.ArgumentConversionContext;
import io.micronaut.core.type.Argument;
import io.micronaut.http.MutableHttpRequest;
import io.micronaut.http.client.bind.ClientRequestUriContext;
import io.micronaut.http.client.bind.TypedClientArgumentRequestBinder;

import jakarta.inject.Singleton;

@Singleton
public class MetadataClientArgumentBinder implements TypedClientArgumentRequestBinder<Metadata> {

    @Override
    @NonNull
    public Argument<Metadata> argumentType() {
        return Argument.of(Metadata.class);
    }

    @Override
    public void bind(@NonNull ArgumentConversionContext<Metadata> context,
                     @NonNull ClientRequestUriContext uriContext,
                     @NonNull Metadata value,
                     @NonNull MutableHttpRequest<?> request) {
        request.header("X-Metadata-Version", value.getVersion().toString());
        request.header("X-Metadata-Deployment-Id", value.getDeploymentId().toString());
    }
}
```

Binding On Method

It is also possible to create a binder, that would change the request with an annotation on the method. For example:

Client With Annotated Method

Java | Groovy | Kotlin

```java
@Client("/")
public interface NameAuthorizedClient {

    @Get("/client/authorized-resource")
    @NameAuthorization(name="Bob")  1
    String get();
}
```

1 The `@NameAuthorization` is annotating a method

The annotation is defined as:

Annotation Definition

Java | Groovy | Kotlin

JAVA

```
@Documented
@Retention(RUNTIME)
@Target(METHOD)  1
@Bindable
public @interface NameAuthorization {
    @AliasFor(member = "name")
    String value() default "";

    @AliasFor(member = "value")
    String name() default "";
}
```

Copy to Clipboard

1 It is defined to be used on methods

The following binder specifies the behaviour:

Annotation Definition

Java | Groovy | Kotlin

JAVA

```
@Singleton  1
public class NameAuthorizationBinder implements AnnotatedClientRequestBinder<NameAuthorization> {  2
    @NotNull
    @Override
    public Class<NameAuthorization> getAnnotationType() {
        return NameAuthorization.class;
    }

    @Override
    public void bind(  3
            @NonNull MethodInvocationContext<Object, Object> context,
            @NonNull ClientRequestUriContext uriContext,
            @NonNull MutableHttpRequest<?> request
    ) {
        context.getValue(NameAuthorization.class)
                .ifPresent(name -> uriContext.addQueryParameter("name", String.valueOf(name)));

    }
}
```

Copy to Clipboard

1 The `@Singleton` annotation registers it in Micronaut context

2 It implements the `AnnotatedClientRequestBinder<NameAuthorization>`

3 The custom `bind` method is used to implement the behaviour of the binder

7.3.2 Streaming with @Client

The **@Client** annotation can also handle streaming HTTP responses.

Streaming JSON with @Client

For example, to write a client that streams data from the controller defined in the **JSON Streaming** section of the documentation, you can define a client that returns an unbound **Publisher** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) such as Reactor's **Flux** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html) or a RxJava's **Flowable** (http://reactivex.io/RxJava/2.x/javadoc/io/reactivex/Flowable.html):

HeadlineClient.java

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.annotation.Get;
import io.micronaut.http.client.annotation.Client;
import org.reactivestreams.Publisher;
import reactor.core.publisher.Flux;
import static io.micronaut.http.MediaType.APPLICATION_JSON_STREAM;

@Client("/streaming")
public interface HeadlineClient {

    @Get(value = "/headlines", processes = APPLICATION_JSON_STREAM)  1
    Publisher<Headline> streamHeadlines();  2
```

Copy to Clipboard

1 The `@Get` method processes responses of type `APPLICATION_JSON_STREAM`

2 The return type is `Publisher`

The following example shows how the previously defined `HeadlineClient` can be invoked from a test:

Streaming HeadlineClient

Java | Groovy | Kotlin

JAVA

```
@Test
public void testClientAnnotationStreaming() {
    try(EmbeddedServer embeddedServer = ApplicationContext.run(EmbeddedServer.class)) {
        HeadlineClient headlineClient = embeddedServer
                                        .getApplicationContext()
                                        .getBean(HeadlineClient.class);  1

        Mono<Headline> firstHeadline = Mono.from(headlineClient.streamHeadlines());  2

        Headline headline = firstHeadline.block();  3

        assertNotNull(headline);
        assertTrue(headline.getText().startsWith("Latest Headline"));
    }
}
```

Copy to Clipboard

1 The client is retrieved from the **ApplicationContext**

2 The `next` method emits the first element emmited by the **Flux** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html) into a **Mono** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html).

3 The `block()` method retrieves the result in the test.

Streaming Clients and Response Types

The example defined in the previous section expects the server to respond with a stream of JSON objects, and the content type to be `application/x-json-stream`. For example:

A JSON Stream

JSON

```
{"title":"The Stand"}
{"title":"The Shining"}
```

The reason for this is simple; a sequence of JSON object is not, in fact, valid JSON and hence the response content type cannot be `application/json`. For the JSON to be valid it would have to return an array:

A JSON Array

JSON

```
[
    {"title":"The Stand"},
    {"title":"The Shining"}
]
```

Micronaut's client does however support streaming of both individual JSON objects via `application/x-json-stream` and also JSON arrays defined with `application/json`.

If the server returns `application/json` and a non-single **Publisher** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) is returned (such as a Reactor's **Flux** (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html) or a RxJava's **Flowable** (http://reactivex.io/RxJava/2.x/javadoc/io/reactivex/Flowable.html)), the client streams the array elements as they become available.

Streaming Clients and Read Timeout

When streaming responses from servers, the underlying HTTP client will not apply the default `readTimeout` setting (which defaults to 10 seconds) of the **HttpClientConfiguration** since the delay between reads for streaming responses may differ from normal reads.

Instead, the `read-idle-timeout` setting (which defaults to 5 minutes) dictates when to close a connection after it becomes idle.

If you stream data from a server that defines a longer delay than 5 minutes between items, you should adjust `readIdleTimeout`. The following configuration in `application.yml` demonstrates how:

Adjusting the readIdleTimeout

YAML

```
micronaut:
  http:
    client:
      read-idle-timeout: 10m
```

The above example sets the `readIdleTimeout` to ten minutes.

Streaming Server Sent Events

Micronaut features a native client for Server Sent Events (SSE) defined by the interface **SseClient**.

You can use this client to stream SSE events from any server that emits them.

Although SSE streams are typically consumed by a browser `EventSource`, there are cases where you may wish to consume an SSE stream via **SseClient**, such as in unit tests or when a Micronaut service acts as a gateway for another service.

The **@Client** annotation also supports consuming SSE streams. For example, consider the following controller method that produces a stream of SSE events:

SSE Controller

Java | Groovy | Kotlin

JAVA

```
@Get(value = "/headlines", processes = MediaType.TEXT_EVENT_STREAM)  1
Publisher<Event<Headline>> streamHeadlines() {
    return Flux.<Event<Headline>>create((emitter) -> {   2
        Headline headline = new Headline();
        headline.setText("Latest Headline at " + ZonedDateTime.now());
        emitter.next(Event.of(headline));
        emitter.complete();
    }, FluxSink.OverflowStrategy.BUFFER)
            .repeat(100)  3
            .delayElements(Duration.of(1, ChronoUnit.SECONDS));  4
}
```

Copy to Clipboard

1. The controller defines a **@Get** annotation that produces a `MediaType.TEXT_EVENT_STREAM`
2. The method uses Reactor to emit a `Headline` object
3. The `repeat` method repeats the emission 100 times
4. With a delay of one second between each

Notice that the return type of the controller is also **Event** and that the `Event.of` method creates events to stream to the client.

To define a client that consumes the events, define a method that processes `MediaType.TEXT_EVENT_STREAM`:

SSE Client

Java | Groovy | Kotlin

JAVA

```
@Client("/streaming/sse")
public interface HeadlineClient {

    @Get(value = "/headlines", processes = TEXT_EVENT_STREAM)
    Publisher<Event<Headline>> streamHeadlines();
}
```

Copy to Clipboard

The generic type of the `Flux` can be either an **Event**, in which case you will receive the full event object, or a POJO, in which case you will receive only the data contained within the event converted from JSON.

7.3.3 Error Responses

If an HTTP response is returned with a code of 400 or higher, an **HttpClientResponseException** is created. The exception contains the original response. How that exception is thrown depends on the method return type.

- For reactive response types, the exception is passed through the publisher as an error.
- For blocking response types, the exception is thrown and should be caught and handled by the caller.

The one exception to this rule is HTTP Not Found (404) responses. This exception only applies to the declarative client.

HTTP Not Found (404) responses for blocking return types is **not** considered an error condition and the client exception will **not** be thrown. That behavior includes methods that return `void`.

If the method returns an `HttpResponse`, the original response is returned. If the return type is `Optional`, an empty optional is returned. For all other types, `null` is returned.

7.3.4 Customizing Request Headers

Customizing the request headers deserves special mention as there are several ways that can be accomplished.

Populating Headers Using Configuration

The **@Header** annotation can be declared at the type level and is repeatable such that it is possible to drive the request headers sent via configuration using annotation metadata.

The following example serves to illustrate this:

Defining Headers via Configuration

Java | Groovy | Kotlin

JAVA

```
@Client("/pets")
@Header(name="X-Pet-Client", value="${pet.client.id}")
public interface PetClient extends PetOperations {

    @Override
    @SingleResult
    Publisher<Pet> save(String name, int age);

    @Get("/{name}")
    @SingleResult
    Publisher<Pet> get(String name);
}
```

Copy to Clipboard

The above example defines a **@Header** annotation on the `PetClient` interface that reads the `pet.client.id` property using property placeholder configuration.

Then set the following in `application.yml` to populate the value:

Configuring Headers in YAML

YAML

```
pet:
  client:
    id: foo
```

Alternatively you can supply a `PET_CLIENT_ID` environment variable and the value will be populated.

Populating Headers using a Client Filter

Alternatively, to dynamically populate headers, another option is to use a **Client Filter**.

For more information on writing client filters see the **Client Filters** section of the guide.

7.3.5 Customizing Jackson Settings

As mentioned previously, Jackson is used for message encoding to JSON. A default Jackson `ObjectMapper` is configured and used by Micronaut HTTP clients.

You can override the settings used to construct the `ObjectMapper` with properties defined by the **JacksonConfiguration** class in `application.yml`.

For example, the following configuration enables indented output for Jackson:

Example Jackson Configuration

YAML

```
jackson:
  serialization:
    indentOutput: true
```

However, these settings apply globally and impact both how the HTTP server renders JSON and how JSON is sent from the HTTP client. Given that, sometimes it is useful to provide client-specific Jackson settings. You can do this with the **@JacksonFeatures** annotation on a client:

As an example, the following snippet is taken from Micronaut's native Eureka client (which of course uses Micronaut's HTTP client):

Example of JacksonFeatures

JAVA

```
@Client(id = EurekaClient.SERVICE_ID,
        path = "/eureka",
        configuration = EurekaConfiguration.class)
@JacksonFeatures(
    enabledSerializationFeatures = WRAP_ROOT_VALUE,
    disabledSerializationFeatures = WRITE_SINGLE_ELEM_ARRAYS_UNWRAPPED,
    enabledDeserializationFeatures = {UNWRAP_ROOT_VALUE, ACCEPT_SINGLE_VALUE_AS_ARRAY}
)
public interface EurekaClient {
    ...
}
```

The Eureka serialization format for JSON uses the `WRAP_ROOT_VALUE` serialization feature of Jackson, hence it is enabled just for that client.

If the customization offered by `JacksonFeatures` is not enough, you can also write a **BeanCreatedEventListener** for the `ObjectMapper` and add whatever customizations you need.

7.3.6 Retry and Circuit Breaker

Recovering from failure is critical for HTTP clients, and that is where Micronaut's integrated **Retry Advice** comes in handy.

You can declare the **@Retryable** or **@CircuitBreaker** annotations on any **@Client** interface and the retry policy will be applied, for example:

Declaring @Retryable

Java | Groovy | Kotlin

JAVA

```
@Client("/pets")
@Retryable
public interface PetClient extends PetOperations {

    @Override
    @SingleResult
    Publisher<Pet> save(String name, int age);
}
```

Copy to Clipboard

For more information on customizing retry, see the section on **Retry Advice**.

7.3.7 Client Fallbacks

In distributed systems, failure happens and it is best to be prepared for it and handle it gracefully.

In addition, when developing Microservices it is quite common to work on a single Microservice without other Microservices the project requires being available.

With that in mind, Micronaut features a native fallback mechanism that is integrated into **Retry Advice** that allows falling back to another implementation in the case of failure.

Using the **@Fallback** annotation, you can declare a fallback implementation of a client to be used when all possible retries have been exhausted.

In fact the mechanism is not strictly linked to Retry; you can declare any class as **@Recoverable**, and if a method call fails (or, in the case of reactive types, an error is emitted) a class annotated with `@Fallback` will be searched for.

To illustrate this, consider again the `PetOperations` interface declared earlier. You can define a `PetFallback` class that will be called in the case of failure:

Defining a Fallback

Java | Groovy | Kotlin

JAVA

```
@Fallback
public class PetFallback implements PetOperations {
    @Override
    @SingleResult
    public Publisher<Pet> save(String name, int age) {
        Pet pet = new Pet();
        pet.setAge(age);
        pet.setName(name);
        return Mono.just(pet);
    }
}
```

Copy to Clipboard

If you only need fallbacks to help with testing against external Microservices, you can define fallbacks in the `src/test/java` directory so they are not included in production code. You will have to specify `@Recoverable(api = PetOperations.class)` on the declarative client if you are using fallbacks without hystrix.

As you can see the fallback does not perform any network operations and is quite simple, hence will provide a successful result in the case of an external system being down.

Of course, the actual behaviour of the fallback is up to you. You can for example implement a fallback that pulls data from a local cache when real data is not available, and sends alert emails or other notifications to operations about downtime.

7.3.8 Netflix Hystrix Support

Using the CLI

If you create your project using the Micronaut CLI, supply the `netflix-hystrix` feature to configure Hystrix in your project:

```
$ mn create-app my-app --features netflix-hystrix
```

Netflix Hystrix (https://github.com/Netflix/Hystrix) is a fault tolerance library developed by the Netflix team and is designed to improve resilience of interprocess communication.

Micronaut integrates with Hystrix through the `netflix-hystrix` module, which you can add to your build:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.netflix</groupId>
    <artifactId>micronaut-netflix-hystrix</artifactId>
</dependency>
```

Copy to Clipboard

Using the @HystrixCommand Annotation

With the above dependency declared you can annotate any method (including methods defined on `@Client` interfaces) with the HystrixCommand (https://micronaut-projects.github.io/micronaut-netflix/latest/api/io/micronaut/configuration/hystrix/annotation.HystrixCommand.html) annotation, and method's execution will be wrapped in a Hystrix command. For example:

Using @HystrixCommand

GROOVY

```
@HystrixCommand
String hello(String name) {
    return "Hello $name"
}
```

This works for reactive return types such as Flux (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html), and the reactive type will be wrapped in a `HystrixObservableCommand`.

The HystrixCommand (https://micronaut-projects.github.io/micronaut-netflix/latest/api/io/micronaut/configuration/hystrix/annotation.HystrixCommand.html) annotation also integrates with Micronaut's support for **Retry Advice** and **Fallbacks**

For information on how to customize the Hystrix thread pool, group, and properties, see the Javadoc for HystrixCommand (https://micronaut-projects.github.io/micronaut-netflix/latest/api/io/micronaut/configuration/hystrix/annotation.HystrixCommand.html).

Enabling Hystrix Stream and Dashboard

You can enable a Server Sent Event stream to feed into the Hystrix Dashboard (https://github.com/Netflix-Skunkworks/hystrix-dashboard) by setting the `hystrix.stream.enabled` setting to `true` in `application.yml`:

Enabling Hystrix Stream

YAML

```
hystrix:
  stream:
    enabled: true
```

This exposes a `/hystrix.stream` endpoint with the format the Hystrix Dashboard (https://github.com/Netflix-Skunkworks/hystrix-dashboard) expects.

7.4 HTTP Client Filters

Often, you need to include the same HTTP headers or URL parameters in a set of requests against a third-party API or when calling another Microservice.

To simplify this, you can define **HttpClientFilter** classes that are applied to all matching HTTP client requests.

As an example, say you want to build a client to communicate with the **Bintray REST API** (https://bintray.com/docs/api/). It would be tedious to specify authentication for every single HTTP call.

To resolve this you can define a filter. The following is an example `BintrayService`:

Java | Groovy | Kotlin

JAVA

```
class BintrayApi {
    public static final String URL = 'https://api.bintray.com'
}

@Singleton
class BintrayService {
    final HttpClient client;
    final String org;

    BintrayService(
            @Client(BintrayApi.URL) HttpClient client,           1
            @Value("${bintray.organization}") String org ) {
        this.client = client;
        this.org = org;
    }

    Flux<HttpResponse<String>> fetchRepositories() {
        return Flux.from(client.exchange(HttpRequest.GET(
                "/repos/" + org), String.class));  2
    }

    Flux<HttpResponse<String>> fetchPackages(String repo) {
        return Flux.from(client.exchange(HttpRequest.GET(
                "/repos/" + org + "/" + repo + "/packages"), String.class));  2
    }
}
```

Copy to Clipboard

1. An **ReactorHttpClient** (https://micronaut-projects.github.io/micronaut-reactor/latest/api/io/micronaut/reactor/client/ReactorHttpClient) is injected for the Bintray API
2. The organization is configurable via configuration

The Bintray API is secured. To authenticate you add an `Authorization` header for every request. You can modify `fetchRepositories` and `fetchPackages` methods to include the necessary HTTP Header for each request, but using a filter is much simpler:

Java | Groovy | Kotlin

JAVA

```
@Filter("/repos/**")  1
class BintrayFilter implements HttpClientFilter {

    final String username;
    final String token;

    BintrayFilter(
            @Value("${bintray.username}") String username,  2
            @Value("${bintray.token}") String token ) {  2
        this.username = username;
        this.token = token;
    }

    @Override
    public Publisher<? extends HttpResponse<?>> doFilter(MutableHttpRequest<?> request,
                                                         ClientFilterChain chain) {
        return chain.proceed(
                request.basicAuth(username, token)  3
        );
    }
}
```

Copy to Clipboard

1. You can match only a subset of paths with a Client filter.
2. The `username` and `token` are injected via configuration
3. The `basicAuth` method includes HTTP Basic credentials

Now when you invoke the `bintrayService.fetchRepositories()` method, the `Authorization` HTTP header is included in the request.

Injecting Another Client into a HttpClientFilter

To create an **ReactorHttpClient** (https://micronaut-projects.github.io/micronaut-reactor/latest/api/io/micronaut/reactor/client/ReactorHttpClient), Micronaut needs to resolve all `HttpClientFilter` instances, which creates a circular dependency when injecting another **https://micronaut-projects.github.io/micronaut-reactor/latest/api/io/micronaut/reactor/client.ReactorHttpClient** or a `@Client` bean into an instance of a `HttpClientFilter`.

To resolve this issue, use the **BeanProvider** interface to inject another **ReactorHttpClient** (https://micronaut-projects.github.io/micronaut-reactor/latest/api/io/micronaut/reactor/client/ReactorHttpClient) or a `@Client` bean into an instance of `HttpClientFilter`.

The following example which implements a filter allowing **authentication between services on Google Cloud Run** (https://cloud.google.com/run/docs/authenticating/service-to-service) demonstrates how to use **BeanProvider** to inject another client:

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.BeanProvider;
import io.micronaut.context.annotation.Requires;
import io.micronaut.context.env.Environment;
import io.micronaut.http.HttpRequest;
import io.micronaut.http.HttpResponse;
import io.micronaut.http.MutableHttpRequest;
import io.micronaut.http.annotation.Filter;
import io.micronaut.http.client.HttpClient;
import io.micronaut.http.filter.ClientFilterChain;
import io.micronaut.http.filter.HttpClientFilter;
import org.reactivestreams.Publisher;
import reactor.core.publisher.Mono;

import java.io.UnsupportedEncodingException;
import java.net.URI;
import java.net.URLEncoder;

@Requires(env = Environment.GOOGLE_COMPUTE)
@Filter(patterns = "/google-auth/api/**")
public class GoogleAuthFilter implements HttpClientFilter {

    private final BeanProvider<HttpClient> authClientProvider;

    public GoogleAuthFilter(BeanProvider<HttpClient> httpClientProvider) {  1
        this.authClientProvider = httpClientProvider;
    }

    @Override
    public Publisher<? extends HttpResponse<?>> doFilter(MutableHttpRequest<?> request,
                                                         ClientFilterChain chain) {
        return Mono.fromCallable(() -> encodeURI(request))
                .flux()
                .flatMap(uri -> authClientProvider.get().retrieve(HttpRequest.GET(uri)  2
                        .header("Metadata-Flavor", "Google")))
                .flatMap(t -> chain.proceed(request.bearerAuth(t)));
    }

    private String encodeURI(MutableHttpRequest<?> request) throws UnsupportedEncodingException {
        URI fullURI = request.getUri();
        String receivingURI = fullURI.getScheme() + "://" + fullURI.getHost();
        return "http://metadata/computeMetadata/v1/instance/service-accounts/default/identity?audience=" +
                URLEncoder.encode(receivingURI, "UTF-8");
    }
}
```

Copy to Clipboard

1 The **BeanProvider** interface is used to inject another client, avoiding a circular reference

2 The `get()` method of the `Provider` interface is used to obtain the client instance.

Filter Matching By Annotation

For cases where a filter should be applied to a client regardless of the URL, filters can be matched by the presence of an annotation applied to both the filter and the client. Given the following client:

Java | Groovy | Kotlin

```java
import io.micronaut.http.annotation.Get;
import io.micronaut.http.client.annotation.Client;

@BasicAuth  1
@Client("/message")
public interface BasicAuthClient {

    @Get
    String getMessage();
}
```

Copy to Clipboard

1 The `@BasicAuth` annotation is applied to the client

The following filter will filter the client requests:

Java | Groovy | Kotlin

```java
import io.micronaut.http.HttpResponse;
import io.micronaut.http.MutableHttpRequest;
import io.micronaut.http.filter.ClientFilterChain;
import io.micronaut.http.filter.HttpClientFilter;
import org.reactivestreams.Publisher;

import jakarta.inject.Singleton;

@BasicAuth  1
@Singleton  2
public class BasicAuthClientFilter implements HttpClientFilter {

    @Override
    public Publisher<? extends HttpResponse<?>> doFilter(MutableHttpRequest<?> request,
                                                         ClientFilterChain chain) {
        return chain.proceed(request.basicAuth("user", "pass"));
    }
}
```

Copy to Clipboard

1 The same annotation, `@BasicAuth`, is applied to the filter

2 Normally the `@Filter` annotation makes filters singletons by default. Because the `@Filter` annotation is not used, the desired scope must be applied

The `@BasicAuth` annotation is just an example and can be replaced with your own.

Java | Groovy | Kotlin

```java
import io.micronaut.http.annotation.FilterMatcher;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@FilterMatcher  1
@Documented
@Retention(RUNTIME)
@Target({TYPE, PARAMETER})
public @interface BasicAuth {
}
```

Copy to Clipboard

1 The only requirement for custom annotations is that the **@FilterMatcher** annotation must be present

7.5 HTTP Client Sample

Read the HTTP Client Guide (**Java** (http://guides.micronaut.io/micronaut-http-client/guide/index.html), **Groovy** (http://guides.micronaut.io/micronaut-http-client-groovy/guide/index.html), **Kotlin** (http://guides.micronaut.io/micronaut-http-client-kotlin/guide/index.html)), a step-by-step tutorial, to learn more.

8 Cloud Native Features

The majority of JVM frameworks in use today were designed before the rise of cloud deployments and microservice architectures. Applications built with these frameworks were intended to be deployed to traditional Java containers. As a result, cloud support in these frameworks typically comes as an add-on rather than as core design features.

Micronaut was designed from the ground up for building microservices for the cloud. As a result, many key features that typically require external libraries or services are available within your application itself. To override one of the industry's current favorite buzzwords, Micronaut applications are "natively cloud-native".

The following are some cloud-specific features that are integrated directly into the Micronaut runtime:

- Distributed Configuration
- Service Discovery
- Client-Side Load-Balancing
- Distributed Tracing
- Serverless Functions

Many features in Micronaut are heavily inspired by features from Spring and Grails. This is by design and helps developers who are already familiar with systems such as Spring Cloud.

The following sections cover these features and how to use them.

8.1 Cloud Configuration

Applications built for the Cloud often need to adapt to running in a Cloud environment, read and share configuration in a distributed manner, and externalize configuration to the environment where necessary.

Micronaut's **Environment** concept is by default Cloud platform-aware and makes a best effort to detect the underlying active environment.

You can then use the **Requires** annotation to **conditionally load bean definitions**.

The following table summarizes the constants in the **Environment** interface and provides an example:

Table 1. Micronaut Environment Detection

Constant	Description	Requires Example	Environment name
ANDROID	The application is running as an Android application	`@Requires(env = Environment.ANDROID)`	`android`
TEST	The application is running within a JUnit or Spock test	`@Requires(env = Environment.TEST)`	`test`
CLOUD	The application is running in a Cloud environment (present for all other cloud platform types)	`@Requires(env = Environment.CLOUD)`	`cloud`
AMAZON_EC2	Running on Amazon EC2 (https://aws.amazon.com/ec2)	`@Requires(env = Environment.AMAZON_EC2)`	`ec2`
GOOGLE_COMPUTE	Running on Google Compute (https://cloud.google.com/compute/)	`@Requires(env = Environment.GOOGLE_COMPUTE)`	`gcp`
KUBERNETES	Running on Kubernetes (https://www.kubernetes.io)	`@Requires(env = Environment.KUBERNETES)`	`k8s`
HEROKU	Running on Heroku (https://heroku.com)	`@Requires(env = Environment.HEROKU)`	`heroku`
CLOUD_FOUNDRY	Running on Cloud Foundry (https://www.cloudfoundry.org)	`@Requires(env = Environment.CLOUD_FOUNDRY)`	`pcf`
AZURE	Running on Microsoft Azure (https://azure.microsoft.com)	`@Requires(env = Environment.AZURE)`	`azure`
IBM	Running on IBM Cloud (https://www.ibm.com/cloud/)	`@Requires(env = Environment.IBM)`	`ibm`
DIGITAL_OCEAN	Running on Digital Ocean (https://www.digitalocean.com/)	`@Requires(env = Environment.DIGITAL_OCEAN)`	`digitalocean`
ORACLE_CLOUD	Running on Oracle Cloud (https://cloud.oracle.com/)	`@Requires(env = Environment.ORACLE_CLOUD)`	`oraclecloud`

Note that you can have multiple environments active, for example when running in Kubernetes on AWS.

In addition, using the value of the constants defined in the table above you can create environment-specific configuration files. For example if you create a `src/main/resources/application-gcp.yml` file, it is only loaded when running on Google Compute.

Any configuration property in the **Environment** can also be set via an environment variable. For example, setting the `CONSUL_CLIENT_HOST` environment variable overrides the `host` property in **ConsulConfiguration** (https://micronaut-projects.github.io/micronaut-discovery-client/latest/api/io/micronaut/discovery/consul/ConsulConfiguration.html).

Using Cloud Instance Metadata

When Micronaut detects it is running on a supported cloud platform, on startup it populates the interface **ComputeInstanceMetadata**.

As of Micronaut 2.1.x this logic depends on the presence of the appropriate core Cloud module for Oracle Cloud, AWS, or GCP.

All this data is merged together into the `metadata` property for the running **ServiceInstance**.

To access the metadata for your application instance you can use the interface **EmbeddedServerInstance**, and call `getMetadata()` which returns a Map of the metadata.

If you connect remotely via a client, the instance metadata can be referenced once you have retrieved a **ServiceInstance** from either the **LoadBalancer** or **DiscoveryClient** APIs.

The Netflix Ribbon client-side load balancer can be configured to use the metadata to do zone-aware client-side load balancing. See **Client-Side Load Balancing**

To obtain metadata for a service via **Service Discovery** use the **LoadBalancerResolver** interface to resolve a **LoadBalancer** and obtain a reference to a service by identifier:

Obtaining Metadata for a Service instance

JAVA

```
LoadBalancer loadBalancer = loadBalancerResolver.resolve("some-service");
Flux.from(
    loadBalancer.select()
).subscribe((instance) ->
    ConvertibleValues<String> metaData = instance.getMetadata();
    ...
);
```

The **EmbeddedServerInstance** is available through event listeners that listen for the **ServiceReadyEvent**. The **@EventListener** annotation makes it easy to listen for the event in your beans.

To obtain metadata for the locally running server, use an **EventListener** for the **ServiceReadyEvent**:

Obtaining Metadata for a Local Server

JAVA

```
@EventListener
void onServiceStarted(ServiceReadyEvent event) {
    ServiceInstance serviceInstance = event.getSource();
    ConvertibleValues<String> metadata = serviceInstance.getMetadata();
}
```

8.1.1 Distributed Configuration

As you can see, Micronaut features a robust system for externalizing and adapting configuration to the environment inspired by similar approaches in Grails and Spring Boot.

However, what if you want multiple microservices to share configuration? Micronaut includes APIs for distributed configuration.

The **ConfigurationClient** interface has a `getPropertySources` method that can be implemented to read and resolve configuration from distributed sources.

The `getPropertySources` returns a **Publisher** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) that emits zero or many **PropertySource** instances.

The default implementation is **DefaultCompositeConfigurationClient** which merges all registered `ConfigurationClient` beans into a single bean.

You can either implement your own **ConfigurationClient** or use the implementations provided by Micronaut. The following sections cover those.

8.1.2 HashiCorp Consul Support

Consul (https://www.consul.io) is a popular Service Discovery and Distributed Configuration server provided by HashiCorp. Micronaut features a native **ConsulClient** (https://micronaut-projects.github.io/micronaut-discovery-client/latest/api/io/micronaut/discovery/consul/client/v1/ConsulClient.html) that uses Micronaut's support for **Declarative HTTP Clients**.

Starting Consul

The quickest way to start using Consul is via Docker:

1. Starting Consul with Docker

```
docker run -p 8500:8500 consul
```

Alternatively you can **install and run a local Consul instance** (https://www.consul.io/docs/install/index.html).

Enabling Distributed Configuration with Consul

Using the CLI

If you create your project using the Micronaut CLI, supply the `config-consul` feature to enable Consul's distributed configuration in your project:

```
$ mn create-app my-app --features config-consul
```

To enable distributed configuration, similar to Spring Boot and Grails, create a `src/main/resources/bootstrap.yml` configuration file and enable the configuration client:

bootstrap.yml

YAML

```yaml
micronaut:
  application:
    name: hello-world
  config-client:
    enabled: true
consul:
  client:
    defaultZone: "${CONSUL_HOST:localhost}:${CONSUL_PORT:8500}"
```

After enabling distributed configuration, store the configuration to share in Consul's key/value store. There are a number of ways to do that.

Storing Configuration as Key/Value Pairs

One way is to store the keys and values directly in Consul. In this case by default Micronaut looks for configuration in the Consul `/config` directory.

You can alter the path searched for by setting `consul.client.config.path`

Within the `/config` directory Micronaut searches values within the following directories in order of precedence:

Table 1. Configuration Resolution Precedence

Directory	Description
`/config/application`	Configuration shared by all applications
`/config/application,prod`	Configuration shared by all applications for the `prod` **Environment**
`/config/[APPLICATION_NAME]`	Application-specific configuration, example `/config/hello-world`
`/config/[APPLICATION_NAME],prod`	Application-specific configuration for an active **Environment**

The value of `APPLICATION_NAME` is whatever your have configured `micronaut.application.name` to be in `bootstrap.yml`.

To see this in action, use the following cURL command to store a property called `foo.bar` with a value of `myvalue` in the directory `/config/application`.

Using cURL to Write a Value

BASH

```
curl -X PUT -d @- localhost:8500/v1/kv/config/application/foo.bar <<< myvalue
```

If you now define a `@Value("${foo.bar}")` or call `environment.getProperty(..)` the value `myvalue` will be resolved from Consul.

Storing Configuration in YAML, JSON etc.

Some Consul users prefer storing configuration in blobs of a certain format, such as YAML. Micronaut supports this mode and supports storing configuration in either YAML, JSON, or Java properties format.

The **ConfigDiscoveryConfiguration** has a number of configuration options for configuring how distributed configuration is discovered.

You can set the `consul.client.config.format` option to configure the format with which properties are read.

For example, to configure JSON:

application.yml

```yaml
consul:
  client:
    config:
      format: JSON
```

Now write your configuration in JSON format to Consul:

Using cURL to write JSON

```bash
curl -X PUT  localhost:8500/v1/kv/config/application \
-d @- << EOF
{ "foo": {  "bar": "myvalue" } }
EOF
```

Storing Configuration as File References

Another popular option is **git2consul** (https://github.com/breser/git2consul) which mirrors the contents of a Git repository to Consul's key/value store.

You can setup a Git repository that contains files like `application.yml`, `hello-world-test.json`, etc., and the contents of these files will be cloned to Consul.

In this case, each key in Consul represents a file with an extension, for example `/config/application.yml`, and you must configure the `FILE` format:

application.yml

```yaml
consul:
  client:
    config:
      format: FILE
```

8.1.3 HashiCorp Vault Support

Micronaut integrates with **HashiCorp Vault** (https://www.vaultproject.io/) as a distributed configuration source.

To enable support for Vault Configuration, create a `src/main/resources/bootstrap.yml` configuration file and add the following configuration:

Integrating with HashiCorp Vault

```yaml
micronaut:
  application:
    name: hello-world
  config-client:
    enabled: true

vault:
  client:
    config:
      enabled: true
```

See the **configuration reference** (https://micronaut-projects.github.io/micronaut-discovery-client/latest/guide/configurationreference.html#io.micronaut.discovery.vault.config.VaultClientConfiguration) for all configuration options.

Micronaut uses the configured `micronaut.application.name` to lookup property sources for the application from Vault.

Table 1. Configuration Resolution Precedence

Directory	Description
`/application`	Configuration shared by all applications
`/[APPLICATION_NAME]`	Application-specific configuration
`/application/[ENV_NAME]`	Configuration shared by all applications for an active environment name
`/[APPLICATION_NAME]/[ENV_NAME]`	Application-specific configuration for an active environment name

See the **Documentation for HashiCorp Vault** (https://www.vaultproject.io/api/secret/kv/index.html) for more information on how to setup the server.

8.1.4 Spring Cloud Config Support

Since 1.1, Micronaut features a native **Spring Cloud Configuration** (https://spring.io/projects/spring-cloud-config) for those who have not switched to a dedicated more complete solution like Consul.

To enable support for Spring Cloud Configuration, create a `src/main/resources/bootstrap.yml` configuration file and add the following configuration:

Integrating with Spring Cloud Configuration

YAML

```yaml
micronaut:
  application:
    name: hello-world
  config-client:
    enabled: true
spring:
  cloud:
    config:
      enabled: true
      uri: http://localhost:8888/
      retry-attempts: 4 # optional, number of times to retry
      retry-delay: 2s # optional, delay between retries
```

Micronaut uses the configured `micronaut.application.name` to look up property sources for the application from Spring Cloud config server configured via `spring.cloud.config.uri`.

See the **Documentation for Spring Cloud Config Server** (https://spring.io/projects/spring-cloud-config#learn) for more information on how to set up the server.

8.1.5 AWS Parameter Store Support

Micronaut supports configuration sharing via AWS System Manager Parameter Store. You need the following dependencies configured:

Gradle **Maven**

MAVEN

```xml
<dependency>
    <groupId>io.micronaut.aws</groupId>
    <artifactId>micronaut-aws-parameter-store</artifactId>
</dependency>
```

Copy to Clipboard

To enable distributed configuration, create a `src/main/resources/bootstrap.yml` configuration file and add Parameter Store configuration:

bootstrap.yml

YAML

```yaml
micronaut:
  application:
    name: hello-world
  config-client:
    enabled: true
aws:
  client:
    system-manager:
      parameterstore:
        enabled: true
```

See the configuration reference (https://micronaut-projects.github.io/micronaut-aws/latest/guide/configurationreference.html#io.micronaut.discovery.aws.parameterstore.AWSParameterStoreConfiguration) for all configuration options.

You can configure shared properties from the AWS Console → System Manager → Parameter Store.

Micronaut uses a hierarchy to read configuration values, and supports `String`, `StringList`, and `SecureString` types.

You can create environment-specific configurations as well by including the environment name after an underscore `_`. For example if `micronaut.application.name` is set to `helloworld`, specifying configuration values under `helloworld_test` will be applied only to the `test` environment.

Table 1. Configuration Resolution Precedence

Directory	Description
`/config/application`	Configuration shared by all applications
`/config/[APPLICATION_NAME]`	Application-specific configuration, example `/config/hello-world`
`/config/application_prod`	Configuration shared by all applications for the `prod` Environment
`/config/[APPLICATION_NAME]_prod`	Application-specific configuration for an active Environment

For example, if the configuration name `/config/application_test/server.url` is configured in AWS Parameter Store, any application connecting to that parameter store can retrieve the value using `server.url`. If the application has `micronaut.application.name` configured to be `myapp`, a value with the name `/config/myapp_test/server.url` overrides the value just for that application.

Each level of the tree can be composed of key=value pairs. For multiple key/value pairs, set the type to `StringList`.

For special secure information, such as keys or passwords, use the type `SecureString`. KMS will be automatically invoked when you add and retrieve values, and will decrypt them with the default key store for your account. If you set the configuration to not use secure strings, they will be returned to you encrypted and you must manually decrypt them.

8.1.6 Oracle Cloud Vault Support

See the **Secure Distributed Configuration with Oracle Cloud Vault** (https://micronaut-projects.github.io/micronaut-oracle-cloud/latest/guide/#vault) documentation.

8.1.7 Google Cloud Pub/Sub Support

See the **Micronaut GCP Pub/Sub** (https://micronaut-projects.github.io/micronaut-gcp/latest/guide/#distributedConfiguration) documentation.

8.1.8 Kubernetes Support

See the **Kubernetes Configuration Client** (https://micronaut-projects.github.io/micronaut-kubernetes/latest/guide/#config-client) documentation.

8.2 Service Discovery

Service Discovery enables Microservices to find each other without knowing the physical location or IP address of associated services.

Micronaut integrates with multiple tools and libraries. See **Micronaut Service Discovery documentation** (https://micronaut-projects.github.io/micronaut-discovery-client/latest/guide/) for more details.

8.2.1 Consul Support

See the **Micronaut Consul documentation** (https://micronaut-projects.github.io/micronaut-discovery-client/latest/guide/index.html#serviceDiscoveryConsul).

8.2.2 Eureka Support

See the **Micronaut Eureka documentation** (https://micronaut-projects.github.io/micronaut-discovery-client/latest/guide/index.html#serviceDiscoveryEureka).

8.2.3 Kubernetes Support

Kubernetes is a container runtime with many features including integrated Service Discovery and Distributed Configuration.

Micronaut includes first-class integration with Kubernetes. See the **Micronaut Kubernetes documentation** (https://micronaut-projects.github.io/micronaut-kubernetes/latest/guide/index.html) for more details.

8.2.4 AWS Route 53 Support

To use **Route 53 Service Discovery** (https://aws.amazon.com/route53/), you must meet the following criteria:

- Run EC2 instances of some type
- Have a domain name hosted in Route 53
- Have a newer version of AWS-CLI (such as 14+)

Assuming you have those things, you are ready. It is not as fancy as Consul or Eureka, but other than some initial setup with the AWS-CLI, there is no other software running to go wrong. You can even support health checks if you add a custom health check to your service. To test if your account can create and use Service Discovery, see the Integration Test section. More information is available at **https://docs.aws.amazon.com/Route53/latest/APIReference/overview-service-discovery.html**.

Here are the steps:

1. Use AWS-CLI to create a namespace. You can make either a public or private one depending on the IPs or subnets you use
2. Create a service with DNS Records with AWS-CLI command
3. Add health checks or custom health checks (optional)
4. Add Service ID to your application configuration file like so:

Sample application.yml

YAML

```yaml
aws:
  route53:
    registration
    enabled: true
    aws-service-id: srv-978fs98fsdf
    namespace: micronaut.io
micronaut:
  application:
    name: something
```

1. Make sure you have the following dependencies in your build file:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.aws</groupId>
    <artifactId>micronaut-aws-route53</artifactId>
</dependency>
```

Copy to Clipboard

1. On the client side, you need the same dependencies and fewer configuration options:

Sample application.yml

YAML

```
aws:
  route53:
    discovery:
      client:
        enabled: true
        aws-service-id: srv-978fs98fsdf
        namespace-id: micronaut.io
```

You can then use the **DiscoveryClient** API to find other services registered via Route 53. For example:

Sample code for client

JAVA

```
DiscoveryClient discoveryClient = embeddedServer.getApplicationContext().getBean(DiscoveryClient.class);
List<String> serviceIds = Flux.from(discoveryClient.getServiceIds()).blockFirst();
List<ServiceInstance> instances = Flux.from(discoveryClient.getInstances(serviceIds.get(0))).blockFirst();
```

Creating the Namespace

Namespaces are similar to a regular Route53 hosted zone, and they appear in the Route53 console, but the console doesn't support modifying them. You must use the AWS-CLI at this time for any Service Discovery functionality.

First decide if you are creating a public-facing namespace or a private one, as the commands are different:

Creating Namespace

BASH

```
$ aws servicediscovery create-public-dns-namespace --name micronaut.io --create-request-id create-1522767790 --descript

or

$ aws servicediscovery create-private-dns-namespace --name micronaut.internal.io --create-request-id create-1522767790
```

When you run this you will get an operation ID. You can check the status with the `get-operation` CLI command:

Get Operation Results

BASH

```
$ aws servicediscovery get-operation --operation-id asdffasdfsda
```

You can use this command to get the status of any call you make that returns an operation ID.

The result of the command will tell you the ID of the namespace. Write that down, you'll need it for the next steps. If you get an error it will say what the error was.

Creating the Service and DNS Records

The next step is creating the Service and DNS records.

Create Service

BASH

```
$ aws create-service --name yourservicename --create-request-id somenumber --description someservicedescription --dns-
```

The `DnsRecord` type can be `A` (ipv4), `AAAA` (ipv6), `SRV`, or `CNAME`. `RoutingPolicy` can be `WEIGHTED` or `MULTIVALUE`. Keep in mind `CNAME` must use weighted routing type, `SRV` must have a valid port configured.

To add a health check, use the following syntax on the CLI:

Specifying a Health Check

BASH

```
Type=string,ResourcePath=string,FailureThreshold=integer
```

Type can be 'HTTP','HTTPS', or 'TCP'. You can only use a standard health check on a public namespace. See Custom Health Checks for private namespaces. Resource path should be a URL that returns `200 OK` if it is healthy.

For a custom health check, you only need to specify `--health-check-custom-config FailureThreshold=integer` which works on private namespaces as well.

This is also good because Micronaut sends out pulsation commands to let AWS know the instance is still healthy.

For more help run 'aws discoveryservice create-service help'.

You will get a service ID and an ARN back from this command if successful. Write that down, it is going to go into the Micronaut configuration.

Setting up the configuration in Micronaut

Auto Naming Registration

Add the configuration to make your applications register with Route 53 Auto-discovery:

Registration Properties

```yaml
aws:
  route53:
    registration:
      enabled: true
      aws-service-id=<enter the service id you got after creation on aws cli>
    discovery:
      namespace-id=<enter the namespace id you got after creating the namespace>
```

Discovery Client Configuration

Discovery Properties

```yaml
aws:
  route53:
    discovery:
      client
      enabled: true
      aws-service-id: <enter the service id you got after creation on aws cli>
```

You can also call the following methods by getting the bean "Route53AutoNamingClient":

Discovery Methods

```java
// if serviceId is null it will use property "aws.route53.discovery.client.awsServiceId"
Publisher<List<ServiceInstance>> getInstances(String serviceId)
// reads property "aws.route53.discovery.namespaceId"
Publisher<List<String>> getServiceIds()
```

Integration Tests

If you set the environment variable AWS_SUBNET_ID and have credentials configured in your home directory that are valid (in `~/.aws/credentials`) you can run the integration tests. You need a domain hosted on Route53 as well. This test will create a t2.nano instance, a namespace, service, and register that instance to service discovery. When the test completes it will remove/terminate all resources it spun up.

8.2.5 Manual Service Discovery Configuration

If you do not wish to involve a service discovery server like Consul or you interact with a third-party service that cannot register with Consul you can instead manually configure services that are available via Service discovery.

To do this, use the `micronaut.http.services` setting. For example:

Manually configuring services

```yaml
micronaut:
  http:
    services:
      foo:
        urls:
          - http://foo1
          - http://foo2
```

You can then inject a client with `@Client("foo")` and it will use the above configuration to load balance between the two configured servers.

When using `@Client` with service discovery, the service id must be specified in the annotation in kebab-case. The configuration in the example above however can be in camel case.

You can override this configuration in production by specifying an environment variable such as `MICRONAUT_HTTP_SERVICES_FOO_URLS=http://prod1,http://prod2`

Note that by default no health checking will happen to assert that the referenced services are operational. You can alter that by enabling health checking and optionally specifying a health check path (the default is `/health`):

Enabling Health Checking

YAML

```yaml
micronaut:
  http:
    services:
      foo:
        ...
        health-check: true  1
        health-check-interval: 15s  2
        health-check-uri: /health  3
```

1. Whether to health check the service
2. The interval between checks
3. The URI of the health check request

Micronaut starts a background thread to check the health status of the service and if any of the configured services respond with an error code, they are removed from the list of available services.

8.3 Client Side Load Balancing

When **discovering services** from Consul, Eureka, or other Service Discovery servers, the **DiscoveryClient** emits a list of available **ServiceInstance**.

Micronaut by default automatically performs Round Robin client-side load balancing using the servers in this list. This combined with **Retry Advice** adds extra resiliency to your Microservice infrastructure.

The load balancing is handled by the **LoadBalancer** interface, which has a **LoadBalancer.select()** method that returns a `Publisher` which emits a **ServiceInstance**.

The **Publisher** (http://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html) is returned because the process for selecting a **ServiceInstance** may result in a network operation depending on the **Service Discovery** strategy employed.

The default implementation of the **LoadBalancer** interface is **DiscoveryClientRoundRobinLoadBalancer**. You can replace this strategy with another implementation to customize how client side load balancing is handled in Micronaut, since there are many different ways to optimize load balancing.

For example, you may wish to load balance between services in a particular zone, or to load balance between servers that have the best overall response time.

To replace the **LoadBalancer**, define a bean that **replaces** the **DiscoveryClientLoadBalancerFactory**.

In fact that is exactly what the Netflix Ribbon support does, described in the next section.

8.3.1 Netflix Ribbon Support

Using the CLI

If you create your project using the Micronaut CLI, supply the `netflix-ribbon` feature to configure Netflix Ribbon in your project:

```
$ mn create-app my-app --features netflix-ribbon
```

Netflix Ribbon (https://github.com/Netflix/ribbon) is an inter-process communication library used at Netflix with support for customizable load balancing strategies.

If you need more flexibility in how your application performs client-side load balancing, you can use Micronaut's Netflix Ribbon support.

To add Ribbon support to your application, add the `netflix-ribbon` configuration to your build:

Gradle **Maven**

MAVEN

```xml
<dependency>
    <groupId>io.micronaut.netflix</groupId>
    <artifactId>micronaut-netflix-ribbon</artifactId>
</dependency>
```

Copy to Clipboard

The **LoadBalancer** implementations will now be **RibbonLoadBalancer** (https://micronaut-projects.github.io/micronaut-netflix/latest/api/io/micronaut/configuration/ribbon/RibbonLoadBalancer.html) instances.

Ribbon's **Configuration options** (http://netflix.github.io/ribbon/ribbon-core-javadoc/com/netflix/client/config/CommonClientConfigKey.html) can be set using the `ribbon` namespace in configuration. For example in `application.yml`:

Configuring Ribbon

YAML

```
ribbon:
  VipAddress: test
  ServerListRefreshInterval: 2000
```

Each discovered client can also be configured under `ribbon.clients`. For example given a `@Client(id = "hello-world")` you can configure Ribbon settings with:

Per Client Ribbon Settings

YAML

```
ribbon:
  clients:
    hello-world:
      VipAddress: test
      ServerListRefreshInterval: 2000
```

By default Micronaut registers a DiscoveryClientServerList (https://micronaut-projects.github.io/micronaut-netflix/latest/api/io/micronaut/configuration/ribbon/DiscoveryClientServerList.html) for each client that integrates Ribbon with Micronaut's DiscoveryClient.

8.4 Distributed Tracing

When operating Microservices in production it can be challenging to troubleshoot interactions between Microservices in a distributed architecture.

To solve this problem, a way to visualize interactions between Microservices in a distributed manner can be critical. Currently there are various distributed tracing solutions, the most popular of which are Zipkin (https://zipkin.io) and Jaeger (https://www.jaegertracing.io/), both of which provide different levels of support for the Open Tracing (http://opentracing.io) API.

Micronaut features integration with both Zipkin and Jaeger (via the Open Tracing API).

To enable tracing, add the `tracing` module to your build:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut</groupId>
    <artifactId>micronaut-tracing</artifactId>
</dependency>
```

Copy to Clipboard

Tracing Annotations

The io.micronaut.tracing.annotation package contains annotations that can be declared on methods to create new spans or continue existing spans.

The available annotations are:

- The @NewSpan annotation creates a new span, wrapping the method call or reactive type.
- The @ContinueSpan annotation continues an existing span, wrapping the method call or reactive type.
- The @SpanTag annotation can be used on method arguments to include the value of the argument within a Span's tags. When you use `@SpanTag` on an argument, you need either to annotate the method with `@NewSpan` or `@ContinueSpan`.

The following snippet presents an example of using the annotations:

Using Trace Annotations

JAVA

```
@Singleton
class HelloService {

    @NewSpan("hello-world")  1
    public String hello(@SpanTag("person.name") String name) {  2
        return greet("Hello " + name);
    }

    @ContinueSpan  3
    public String greet(@SpanTag("hello.greeting") String greet) {
        return greet;
    }
}
```

1 The @NewSpan annotation starts a new span

2 Use @SpanTag to include method arguments as tags for the span

3 Use the @ContinueSpan annotation to continue an existing span and incorporate additional tags using `@SpanTag`

Tracing Instrumentation

In addition to explicit tracing tags, Micronaut includes a number of instrumentations to ensure that the Span context is propagated between threads and across Microservice boundaries.

These instrumentations are found in the `io.micronaut.tracing.instrument` package and include HTTP **Client Filters** and **Server Filters** to propagate the necessary headers via HTTP.

Tracing Beans

If the Tracing annotations and existing instrumentations are not sufficient, Micronaut's tracing integration registers a `io.opentracing.Tracer` bean which exposes the Open Tracing API and can be dependency-injected as needed.

Depending on the implementation you choose, there are also additional beans. For example for Zipkin `brave.Tracing` and `brave.SpanCustomizer` beans are available too.

8.4.1 Tracing with Zipkin

Zipkin (https://zipkin.io) is a distributed tracing system. It helps gather timing data to troubleshoot latency problems in microservice architectures. It manages both the collection and retrieval of this data.

Running Zipkin

The quickest way to get up and started with Zipkin is with Docker:

Running Zipkin with Docker

BASH

```
$ docker run -d -p 9411:9411 openzipkin/zipkin
```

Navigate to **`http://localhost:9411`** to view traces.

Sending Traces to Zipkin

Using the CLI

If you create your project using the Micronaut CLI, supply the `tracing-zipkin` feature to include Zipkin tracing in your project:

```
$ mn create-app my-app --features tracing-zipkin
```

To send tracing spans to Zipkin, the minimal configuration requires the following dependencies in your build:

Adding Zipkin Dependencies

GROOVY

```
runtime 'io.zipkin.brave:brave-instrumentation-http'
runtime 'io.zipkin.reporter2:zipkin-reporter'
compile 'io.opentracing.brave:brave-opentracing'
```

Then enable ZipKin tracing in your configuration (potentially only your production configuration):

application.yml

YAML

```
tracing:
  zipkin:
    enabled: true
```

Customizing the Zipkin Sender

To send spans you configure a Zipkin sender. You can configure a **HttpClientSender** that sends Spans asynchronously using Micronaut's native HTTP client with the `tracing.zipkin.http.url` setting:

Configuring Multiple Zipkin Servers

YAML

```
tracing:
  zipkin:
    enabled: true
    http:
      url: http://localhost:9411
```

It is unlikely that sending spans to localhost will be suitable for production deployment, so you generally need to configure the location of one or more Zipkin servers for production:

Configuring Multiple Zipkin Servers

```yaml
tracing:
  zipkin:
    enabled: true
    http:
      urls:
        - http://foo:9411
        - http://bar:9411
```

In production, setting `TRACING_ZIPKIN_HTTP_URLS` environment variable with a comma-separated list of URLs also works.

Alternatively to use a different `zipkin2.reporter.Sender` implementation, you can define a bean of type `zipkin2.reporter.Sender` and it will be picked up.

Zipkin Configuration

There are many configuration options available for the Brave client that sends Spans to Zipkin, and they are generally exposed via the **BraveTracerConfiguration** class. Refer to the Javadoc for available options.

Below is an example of customizing Zipkin configuration:

Customizing Zipkin Configuration

```yaml
tracing:
  zipkin:
    enabled: true
    traceId128Bit: true
    sampler:
      probability: 1
```

You can also optionally dependency-inject common configuration classes into **BraveTracerConfiguration** such as `brave.sampler.Sampler` just by defining them as beans. See the API for **BraveTracerConfiguration** for available injection points.

8.4.2 Tracing with Jaeger

Jaeger (https://www.jaegertracing.io) is a distributed tracing system developed at Uber that is more or less the reference implementation for **Open Tracing** (http://opentracing.io).

Running Jaeger

The easiest way to get started with Jaeger is with Docker:

```bash
$ docker run -d \
  -e COLLECTOR_ZIPKIN_HTTP_PORT=9411 \
  -p 5775:5775/udp \
  -p 6831:6831/udp \
  -p 6832:6832/udp \
  -p 5778:5778 \
  -p 16686:16686 \
  -p 14268:14268 \
  -p 9411:9411 \
  jaegertracing/all-in-one:1.6
```

Navigate to **`http://localhost:16686`** to access the Jaeger UI.

See **Getting Started with Jaeger** (https://www.jaegertracing.io/docs/getting-started/) for more information.

Sending Traces to Jaeger

Using the CLI

If you create your project using the Micronaut CLI, supply the `tracing-jaeger` feature to include Jaeger tracing in your project:

```
$ mn create-app my-app --features tracing-jaeger
```

To send tracing spans to Jaeger, the minimal configuration requires the following dependency in your build:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.jaegertracing</groupId>
    <artifactId>jaeger-thrift</artifactId>
    <version>0.31.0</version>
</dependency>
```

Copy to Clipboard

Then enable Jaeger tracing in your configuration (potentially only your production configuration):

application.yml

YAML

```
tracing:
  jaeger:
    enabled: true
```

By default, Jaeger will be configured to send traces to a locally running Jaeger agent.

Jaeger Configuration

There are many configuration options available for the Jaeger client that sends Spans to Jaeger, and they are generally exposed via the **JaegerConfiguration** class. Refer to the Javadoc for available options.

Below is an example of customizing JaegerConfiguration configuration:

Customizing Jaeger Configuration

YAML

```
tracing:
  jaeger:
    enabled: true
    sampler:
      probability: 0.5
    sender:
      agentHost: foo
      agentPort: 5775
    reporter:
      flushInterval: 2000
      maxQueueSize: 200
```

You can also optionally dependency-inject common configuration classes into **JaegerConfiguration** such as `io.jaegertracing.Configuration.SamplerConfiguration` just by defining them as beans. See the API for **JaegerConfiguration** for available injection points.

9 Serverless Functions

Serverless architectures, where you deploy functions that are fully managed by a Cloud environment and are executed in ephemeral processes, require a unique approach.

Traditional frameworks like Grails and Spring are not really suitable since low memory consumption and fast startup time are critical, since the Function as a Service (FaaS) server typically spins up your function for a period using a cold start and then keeps it warm.

Micronaut's compile-time approach, fast startup time, and low memory footprint make it a great candidate for developing functions, and Micronaut includes dedicated support for developing and deploying functions to AWS Lambda, Google Cloud Function, Azure Function, and any FaaS system that supports running functions as containers (such as OpenFaaS, Rift or Fn).

There are generally two approaches to writing functions with Micronaut:

1. Low-level functions written using the native API of the function platform
2. Higher-level functions where you simply define controllers as you normally do in a typical Micronaut application and deploy to the function platform.

The first has marginally less startup time overhead and is typically used for non-HTTP functions such as functions that listen to an event or background functions.

The second is only for HTTP functions and is useful for users who want to take a slice of an existing application and deploy it as a serverless function. If cold start performance is a concern it is recommended that you consider building a native image with GraalVM for this option.

9.1 AWS Lambda

Support for AWS Lambda is implemented in the **Micronaut AWS** (https://micronaut-projects.github.io/micronaut-aws/latest/guide/#whatsNew) subproject.

Simple Functions with AWS Lambda

You can implement AWS Request Handlers with Micronaut that directly implement the AWS Lambda SDK API. See the documentation on **Micronaut Request Handlers** (https://micronaut-projects.github.io/micronaut-aws/latest/guide/#requestHandlers) for more information.

Using the CLI

To create an AWS Lambda Function:

```
$ mn create-function-app my-app --features aws-lambda
```

Or with Micronaut Launch

```
$ curl https://launch.micronaut.io/create/function/example\?features\=aws-lambda -o example.zip
$ unzip example.zip -d example
```

HTTP Functions with AWS Lambda

You can deploy regular Micronaut applications that use **@Controller**, etc. using Micronaut's support for AWS API Gateway. See the documentation on **AWS API Gateway Support** (https://micronaut-projects.github.io/micronaut-aws/snapshot/guide/#apiProxy) for more information.

Using the CLI

To create an AWS API Gateway Proxy application:

```
$ mn create-app my-app --features aws-lambda
```

Or with Micronaut Launch

```
$ curl https://launch.micronaut.io/example.zip\?features\=aws-lambda -o example.zip
$ unzip example.zip -d example
```

9.2 Google Cloud Function

Support for Google Cloud Function is implemented in the **Micronaut GCP** (https://micronaut-projects.github.io/micronaut-gcp/2.0.x/guide/#cloudFunction) subproject.

Simple Functions with Cloud Function

You can implement Cloud Functions with Micronaut that directly implement the **Cloud Function Framework API** (https://github.com/GoogleCloudPlatform/functions-framework-java). See the documentation on **Simple Functions** (https://micronaut-projects.github.io/micronaut-gcp/2.0.x/guide/#simpleFunctions) for more information.

Using the CLI

To create a Google Cloud Function:

```
$ mn create-function-app my-app --features google-cloud-function
```

Or with Micronaut Launch

```
$ curl https://launch.micronaut.io/create/function/example\?features\=google-cloud-function -o example.zip
$ unzip example.zip -d example
```

HTTP Functions with Cloud Function

You can deploy regular Micronaut applications that use **@Controller** etc. using Micronaut's support for HTTP Functions. See the documentation on **Google Cloud HTTP Functions** (https://micronaut-projects.github.io/micronaut-gcp/2.0.x/guide/#httpFunctions) for more information.

Using the CLI

To create a Google Cloud HTTP Function:

```
$ mn create-app my-app --features google-cloud-function
```

Or with Micronaut Launch

```
$ curl https://launch.micronaut.io/example.zip\?features\=google-cloud-function -o example.zip
$ unzip example.zip -d example
```

9.3 Google Cloud Run

To deploy to **Google Cloud Run** (https://cloud.google.com/run) we recommend using **JIB** (https://github.com/GoogleContainerTools/jib) to containerize your application.

Using the CLI

Creating an application with JIB:

```
$ mn create-app my-app --features jib
```

Or with Micronaut Launch

```
$ curl https://launch.micronaut.io/example.zip\?features\=jib -o example.zip
$ unzip example.zip -d example
```

With JIB setup to deploy your application to Google Container Registry, run:

```
$ ./gradlew jib
```

BASH

You are now ready to deploy your application:

```
$ gcloud run deploy --image gcr.io/[PROJECT ID]/example --platform=managed --allow-unauthenticated
```

BASH

Where `[PROJECT ID]` is your project ID. You will be asked to specify a region and will see output like the following:

```
Service name: (example):
Deploying container to Cloud Run service [example] in project [PROJECT_ID] region [us-central1]

✓ Deploying... Done.
  ✓ Creating Revision...
  ✓ Routing traffic...
  ✓ Setting IAM Policy...
Done.
Service [example] revision [example-00004] has been deployed and is serving 100 percent of traffic at https://example-
```

The URL is the URL of your Cloud Run application.

9.4 Azure Function

Support for Azure Function is implemented in the **Micronaut Azure** (https://micronaut-projects.github.io/micronaut-azure/1.0.x/guide/index.html#azureFunction) subproject.

Simple Functions with Azure Function

You can implement Azure Functions with Micronaut that directly implement the **Azure Function Java SDK** (https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference-java?tabs=consumption). See the documentation on **Azure Functions** (https://micronaut-projects.github.io/micronaut-azure/1.0.x/guide/index.html#azureFunction) for more information.

Using the CLI

To create an Azure Function:

```
$ mn create-function-app my-app --features azure-function
```

Or with Micronaut Launch

```
$ curl https://launch.micronaut.io/create/function/example\?features\=azure-function -o example.zip
$ unzip example.zip -d example
```

HTTP Functions with Azure Function

You can deploy regular Micronaut applications that use **@Controller** etc. using Micronaut's support for Azure HTTP Functions. See the documentation on **Azure HTTP Functions** (https://micronaut-projects.github.io/micronaut-azure/1.0.x/guide/index.html#azureFunction) for more information.

Using the CLI

To create a Azure HTTP Function:

```
$ mn create-app my-app --features azure-function
```

Or with Micronaut Launch

```
$ curl https://launch.micronaut.io/example.zip\?features\=azure-function -o example.zip
$ unzip example.zip -d example
```

10 Message-Driven Microservices

In the past, with monolithic applications, message listeners that listened to messages from messaging systems would frequently be embedded in the same application unit.

In Microservice architectures it is common to have individual Microservice applications that are driven by a message system such as RabbitMQ or Kafka.

In fact a Message-driven Microservice may not even feature an HTTP endpoint or HTTP server (although this can be valuable from a health check and visibility perspective).

10.1 Kafka Support

Apache Kafka (https://kafka.apache.org) is a distributed stream processing platform that can be used for a range of messaging requirements in addition to stream processing and real-time data handling.

Micronaut features dedicated support for defining both Kafka `Producer` and `Consumer` instances. Micronaut applications built with Kafka can be deployed with or without the presence of an HTTP server.

With Micronaut's efficient compile-time AOP and cloud native features, writing efficient Kafka consumer applications that use very little resources is a breeze.

See the documentation for **Micronaut Kafka** (https://micronaut-projects.github.io/micronaut-kafka/latest/guide) for more information on how to build Kafka applications with Micronaut.

10.2 RabbitMQ Support

RabbitMQ (https://www.rabbitmq.com) is the most widely deployed open source message broker.

Micronaut features dedicated support for defining both RabbitMQ publishers and consumers. Micronaut applications built with RabbitMQ can be deployed with or without an HTTP server.

With Micronaut's efficient compile-time AOP, using RabbitMQ has never been easier. Support has been added for publisher confirms and RPC through reactive streams.

See the documentation for **Micronaut RabbitMQ** (https://micronaut-projects.github.io/micronaut-rabbitmq/latest/guide) for more information on how to build RabbitMQ applications with Micronaut.

10.3 Nats.io Support

Nats.io (https://nats.io/) is a simple, secure, and high-performance open source messaging system for cloud native applications, IoT messaging, and microservices architectures.

Micronaut features dedicated support for defining both Nats.io publishers and consumers. Micronaut applications built with Nats.io can be deployed with or without an HTTP server.

With Micronaut's efficient compile-time AOP, using Nats.io has never been easier. Support has been added for publisher confirms and RPC through reactive streams.

See the documentation for **Micronaut Nats** (https://micronaut-projects.github.io/micronaut-nats/latest/guide/) for more information on how to build Nats.io applications with Micronaut.

11 Standalone Command Line Applications

In certain cases you may wish to create standalone command-line (CLI) applications that interact with your Microservice infrastructure.

Examples of applications like this include scheduled tasks, batch applications and general command line applications.

In this case having a robust way to parse command line options and positional parameters is important.

11.1 Picocli Support

Picocli (https://github.com/remkop/picocli) is a command line parser that supports usage help with ANSI colors, autocomplete, and nested subcommands. It has an annotations API to create command line applications with almost no code, and a programmatic API for dynamic uses like creating Domain Specific Languages.

See the **documentation for the Picocli integration** (https://micronaut-projects.github.io/micronaut-picocli/latest/guide/) for more information.

12 Configurations

Micronaut features several built-in configurations that enable integration with common databases and other servers.

12.1 Configurations for Reactive Programming

Project Reactor (https://projectreactor.io) is used internally by Micronaut. However to use Reactor or other reactive libraries (e.g. RxJava) types in your controller and/or HTTP Client methods you need to include dependencies.

12.1.1 Reactor Support

To add support for Reactor, add the following module:

dependency:io.micronaut.scope="compile" (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/micronaut-reactor/.html)

To use the Reactor HTTP client, add the following dependency:

dependency:io.micronaut.scope="compile" (https://projectreactor.io/docs/core/release/api/reactor/core/publisher/micronaut-reactor-http-client/.html)

For more information see the documentation for Micronaut Reactor (https://micronaut-projects.github.io/micronaut-reactor/latest/guide/).

12.1.2 RxJava 3 Support

To add support for RxJava 3, add the following module:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.rxjava3</groupId>
    <artifactId>micronaut-rxjava3</artifactId>
</dependency>
```

Copy to Clipboard

To use the RxJava 3 HTTP client, add the following dependency:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.rxjava3</groupId>
    <artifactId>micronaut-rxjava3-http-client</artifactId>
</dependency>
```

Copy to Clipboard

For more information see the documentation for Micronaut RxJava 3 (https://micronaut-projects.github.io/micronaut-rxjava3/latest/guide/).

12.1.3 RxJava 2 Support

To add support for RxJava 2, add the following module:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.rxjava2</groupId>
    <artifactId>micronaut-rxjava2</artifactId>
</dependency>
```

Copy to Clipboard

To use the RxJava 2 HTTP client, add the following dependency:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.rxjava2</groupId>
    <artifactId>micronaut-rxjava2-http-client</artifactId>
</dependency>
```

Copy to Clipboard

For more information see the documentation for Micronaut RxJava 2 (https://micronaut-projects.github.io/micronaut-rxjava2/latest/guide/).

12.1.4 RxJava 1 Support

Legacy support for RxJava 1 can be added with the following module:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.rxjava1</groupId>
    <artifactId>micronaut-rxjava1</artifactId>
</dependency>
```

Copy to Clipboard

For more information see the Micronaut RxJava1 (https://micronaut-projects.github.io/micronaut-rxjava1/latest/guide/) documentation.

12.2 Configurations for Data Access

This table summarizes the configuration modules and dependencies to add to your build to enable them:

Table 1. Data Access Configuration Modules

Dependency	Description
`io.micronaut.sql:micronaut-jdbc-dbcp`	Configures SQL DataSource (https://docs.oracle.com/javase/8/docs/api/javax/sql/DataSource.html)s using Commons DBCP (https://commons.apache.org/proper/commons-dbcp/)
`io.micronaut.sql:micronaut-jdbc-hikari`	Configures SQL DataSource (https://docs.oracle.com/javase/8/docs/api/javax/sql/DataSource.html)s using Hikari Connection Pool (https://brettwooldridge.github.io/HikariCP/)
`io.micronaut.sql:micronaut-jdbc-tomcat`	Configures SQL DataSource (https://docs.oracle.com/javase/8/docs/api/javax/sql/DataSource.html)s using Tomcat Connection Pool (https://tomcat.apache.org/tomcat-7.0-doc/jdbc-pool.html)
`io.micronaut.sql:micronaut-hibernate-jpa`	Configures Hibernate/JPA `EntityManagerFactory` beans
`io.micronaut.groovy:micronaut-hibernate-gorm`	Configures GORM for Hibernate (http://gorm.grails.org/latest/hibernate/manual) for Groovy applications
`io.micronaut.mongodb:micronaut-mongo-reactive`	Configures the MongoDB Reactive Driver (http://mongodb.github.io/mongo-java-driver-reactivestreams)
`io.micronaut.groovy:micronaut-mongo-gorm`	Configures GORM for MongoDB (http://gorm.grails.org/latest/mongodb/manual) for Groovy applications
`io.micronaut.neo4j:micronaut-neo4j-bolt`	Configures the Bolt Java Driver (https://github.com/neo4j/neo4j-java-driver) for Neo4j (https://neo4j.com)
`io.micronaut.groovy:micronaut-neo4j-gorm`	Configures GORM for Neo4j (http://gorm.grails.org/latest/neo4j/manual) for Groovy applications
`io.micronaut.sql:micronaut-vertx-mysql-client`	Configures the Reactive MySQL Client (https://github.com/eclipse-vertx/vertx-sql-client/tree/master/vertx-mysql-client)
`io.micronaut.sql:micronaut-vertx-pg-client`	Configures the Reactive Postgres Client (https://github.com/eclipse-vertx/vertx-sql-client/tree/master/vertx-pg-client)
`io.micronaut.redis:micronaut-redis-lettuce`	Configures the Lettuce (https://lettuce.io) driver for Redis (https://redis.io)
`io.micronaut.cassandra:micronaut-cassandra`	Configures the Datastax Java Driver (https://github.com/datastax/java-driver) for Cassandra (http://cassandra.apache.org)

For example, to add support for MongoDB, add the following dependency:

build.gradle

GROOVY

```
compile "io.micronaut.mongodb:micronaut-mongo-reactive"
```

For Groovy users, Micronaut provides special support for GORM (http://gorm.grails.org).

With GORM for Hibernate you cannot have both the `hibernate-jpa` and `hibernate-gorm` dependencies.

The following sections go into more detail about configuration options and the exposed beans for each implementation.

12.2.1 Configuring a SQL Data Source

JDBC DataSources can be configured for one of three currently provided implementations - Apache DBCP2, Hikari, and Tomcat are supported by default.

Configuring a JDBC DataSource

Using the CLI

If you create your project using the Micronaut CLI, supply one of the `jdbc-tomcat`, `jdbc-hikari`, or `jdbc-dbcp` features to preconfigure a simple JDBC connection pool in your project, along with a default H2 database driver:

```
$ mn create-app my-app --features jdbc-tomcat
```

To get started, add a dependency for one of the JDBC configurations that corresponds to the implementation you will use. Choose one of the following:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.sql</groupId>
    <artifactId>micronaut-jdbc-tomcat</artifactId>
    <scope>runtime</scope>
</dependency>
```

Copy to Clipboard

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.sql</groupId>
    <artifactId>micronaut-jdbc-hikari</artifactId>
    <scope>runtime</scope>
</dependency>
```

Copy to Clipboard

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.sql</groupId>
    <artifactId>micronaut-jdbc-dbcp</artifactId>
    <scope>runtime</scope>
</dependency>
```

Copy to Clipboard

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.sql</groupId>
    <artifactId>micronaut-jdbc-ucp</artifactId>
    <scope>runtime</scope>
</dependency>
```

Copy to Clipboard

Also, add a JDBC driver dependency to your build. For example to add the **H2 In-Memory Database** (http://www.h2database.com):

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Copy to Clipboard

For more information see the **Configuring JDBC** (https://micronaut-projects.github.io/micronaut-sql/latest/guide/#jdbc) section of the **Micronaut SQL libraries** (https://github.com/micronaut-projects/micronaut-sql) project.

12.2.2 Configuring Hibernate

Setting up a Hibernate/JPA EntityManager

Using the CLI

If you create your project using the Micronaut CLI, supply the `hibernate-jpa` feature to include a Hibernate JPA configuration in your project:

```
$ mn create-app my-app --features hibernate-jpa
```

Micronaut includes support for configuring a **Hibernate** (http://hibernate.org) / JPA `EntityManager` that builds on the **SQL DataSource support**.

Once you have **configured one or more DataSources** to use Hibernate, add the `hibernate-jpa` dependency to your build:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.sql</groupId>
    <artifactId>micronaut-hibernate-jpa</artifactId>
</dependency>
```

Copy to Clipboard

For more information see the **Configuring Hibernate** (https://micronaut-projects.github.io/micronaut-sql/latest/guide/#hibernate) section of the **Micronaut SQL libraries** (https://github.com/micronaut-projects/micronaut-sql) project.

Using GORM for Hibernate

For Groovy users and users familiar with the Grails framework, special support for **GORM for Hibernate** (http://gorm.grails.org) is available. To use GORM for Hibernate **do not** include Micronaut's built-in **SQL Support** or the `hibernate-jpa` dependency since GORM itself takes responsibility for creating the `DataSource`, `SessionFactory`, etc.

Using the CLI

If you create your project using the Micronaut CLI, supply the `hibernate-gorm` feature to include GORM, a basic connection pool configuration, and a default H2 database driver in your project:

```
$ mn create-app my-app --features hibernate-gorm
```

See the **GORM Modules** (https://micronaut-projects.github.io/micronaut-groovy/latest/guide/#gorm) section of the **Micronaut for Groovy** (https://github.com/micronaut-projects/micronaut-groovy) user guide.

12.2.3 Configuring MongoDB

Setting up the Native MongoDB Driver

Using the CLI

If you create your project using the Micronaut CLI, supply the `mongo-reactive` feature to configure the native MongoDB driver in your project:

```
$ mn create-app my-app --features mongo-reactive
```

Micronaut can automatically configure the native MongoDB Java driver. To use this, add the following dependency to your build:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut.mongodb</groupId>
    <artifactId>micronaut-mongo-reactive</artifactId>
</dependency>
```

Copy to Clipboard

Then configure the URI of the MongoDB server in `application.yml`:

Configuring a MongoDB server

YAML

```
mongodb:
  uri: mongodb://username:password@localhost:27017/databaseName
```

The `mongodb.uri` follows the **MongoDB Connection String** (https://docs.mongodb.com/manual/reference/connection-string) format.

A non-blocking Reactive Streams **MongoClient** (http://mongodb.github.io/mongo-java-driver-reactivestreams/1.8/javadoc/com/mongodb/reactivestreams/client/MongoClient.html) is then available for dependency injection.

To use the blocking driver, add a dependency to your build on the mongo-java-driver:

GROOVY

```
compile "org.mongodb:mongo-java-driver"
```

Then the blocking **MongoClient** (http://mongodb.github.io/mongo-java-driver/3.7/javadoc/com/mongodb/MongoClient.html) will be available for injection.

See the **Micronaut MongoDB** (https://micronaut-projects.github.io/micronaut-mongodb/latest/guide/) documentation for further information on configuring and using MongoDB within Micronaut.

12.2.4 Configuring Neo4j

Micronaut features dedicated support for automatically configuring the **Neo4j Bolt Driver** (https://neo4j.com/docs/developer-manual/current/drivers/) for the popular **Neo4j** (https://neo4j.com/) Graph Database.

Using the CLI

If you create your project using the Micronaut CLI, supply the `neo4j-bolt` feature to configure the Neo4j Bolt driver in your project:

```
$ mn create-app my-app --features neo4j-bolt
```

To configure the Neo4j Bolt driver, first add the `neo4j-bolt` module to your build:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut</groupId>
    <artifactId>micronaut-neo4j-bolt</artifactId>
</dependency>
```

Copy to Clipboard

Then configure the URI of the Neo4j server in `application.yml`:

Configuring `neo4j.uri`

YAML

```
neo4j:
  uri: bolt://localhost
```

The `neo4j.uri` setting must be in the format as described in the **Connection URIs** (https://neo4j.com/docs/developer-manual/current/drivers/client-applications/#driver-connection-uris) section of the Neo4j documentation

Once you have the above configuration in place you can inject an instance of the `org.neo4j.driver.v1.Driver` bean, which features both a synchronous blocking API and a non-blocking API based on `CompletableFuture`.

See the **Micronaut Neo4j** (https://micronaut-projects.github.io/micronaut-neo4j/latest/guide/) documentation for further information on configuring and using Neo4j within Micronaut.

12.2.5 Configuring Postgres

Micronaut supports a reactive and non-blocking client to connect to Postgres using **vertx-pg-client** (https://github.com/eclipse-vertx/vertx-sql-client/tree/master/vertx-pg-client), which can handle many database connections with a single thread.

Configuring the Reactive Postgres Client

Using the CLI

If you create your project using the Micronaut CLI, supply the `vertx-pg-client` feature to configure the Reactive Postgres client in your project:

```
$ mn create-app my-app --features vertx-pg-client
```

To configure the Reactive Postgres client, first add the `vertx-pg-client` module to your build:

build.gradle

GROOVY

```
compile "io.micronaut.sql:micronaut-vertx-pg-client"
```

For more information see the **Configuring Reactive Postgres** (https://micronaut-projects.github.io/micronaut-sql/latest/guide/#pgclient) section of the **Micronaut SQL libraries** (https://github.com/micronaut-projects/micronaut-sql) project.

12.2.6 Configuring Redis

Micronaut features automatic configuration of the **Lettuce** (https://lettuce.io) driver for **Redis** (https://redis.io) via the `redis-lettuce` module.

Configuring Lettuce

Using the CLI

If you create your project using the Micronaut CLI, supply the `redis-lettuce` feature to configure the Lettuce driver in your project:

```
$ mn create-app my-app --features redis-lettuce
```

To configure the Lettuce driver, first add the `redis-lettuce` module to your build:

build.gradle

```groovy
compile "io.micronaut.redis:micronaut-redis-lettuce"
```

Then configure the URI of the Redis server in `application.yml`:

Configuring `redis.uri`

```yaml
redis:
  uri: redis://localhost
```

The `redis.uri` setting must be in the format as described in the **Connection URIs** (https://github.com/lettuce-io/lettuce-core/wiki/Redis-URI-and-connection-details) section of the Lettuce wiki

You can also specify multiple Redis URIs using `redis.uris`, in which case a `RedisClusterClient` is created instead.

For more information and further documentation see the **Micronaut Redis** (https://micronaut-projects.github.io/micronaut-redis/latest/guide) documentation.

12.2.7 Configuring Cassandra

Using the CLI

If you create your project using the Micronaut CLI, supply the `cassandra` feature to include Cassandra configuration in your project:

```
$ mn create-app my-app --features cassandra
```

For more information see the **Micronaut Cassandra Module** (https://micronaut-projects.github.io/micronaut-cassandra/latest/guide/) documentation.

12.2.8 Configuring Liquibase

To configure Micronaut integration with **Liquibase** (http://www.liquibase.org), please follow **these instructions** (https://micronaut-projects.github.io/micronaut-liquibase/latest/guide/index.html).

12.2.9 Configuring Flyway

To configure Micronaut integration with **Flyway** (https://flywaydb.org/), please follow **these instructions** (https://micronaut-projects.github.io/micronaut-flyway/latest/guide/index.html)

13 Logging

Micronaut uses **Slf4j** (http://www.slf4j.org/) to log messages. The default implementation for applications created via Micronaut Launch is **Logback** (http://logback.qos.ch/). Any other Slf4j implementation is supported however.

13.1 Logging Messages

To log messages, use the Slf4j `LoggerFactory` to get a logger for your class.

```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class LoggerExample {

    private static Logger logger = LoggerFactory.getLogger(LoggerExample.class);

    public static void main(String[] args) {
        logger.debug("Debug message");
        logger.info("Info message");
        logger.error("Error message");
    }
}
```

13.2 Configuration

Log levels can be configured via properties defined in `application.yml` (and environment variables) with the `logger.levels` prefix:

```yaml
logger:
  levels:
    foo.bar: ERROR
```

The same configuration can be achieved by setting the environment variable `LOGGER_LEVELS_FOO_BAR`. Note that there is currently no way to set log levels for unconventional prefixes such as `foo.barBaz`.

Disabling a Logger with Properties

To disable a logger, you need to set the logger level to `OFF`:

YAML

```yaml
logger:
  levels:
    io.verbose.logger.who.CriedWolf: OFF  1
```

1. This will disable ALL logging for the class `io.verbose.logger.who.CriedWolf`

Note that the ability to control log levels via config is controlled via the **LoggingSystem** interface. Currently, Micronaut includes a single implementation that allows setting log levels for the Logback library. If you use another library, you should provide a bean that implements this interface.

13.3 Logback

To use the logback library, add the following dependency to your build.

Gradle **Maven**

MAVEN

```xml
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
</dependency>
```

Copy to Clipboard

If it does not exist yet, place a **logback.xml** (http://logback.qos.ch/manual/configuration.html) file in the resources folder and modify the content for your needs. For example:

src/main/resources/logback.xml

XML

```xml
<configuration>

    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <withJansi>true</withJansi>

        <encoder>
            <pattern>%cyan(%d{HH:mm:ss.SSS}) %gray([%thread]) %highlight(%-5level) %magenta(%logger{36}) - %msg%n
            </pattern>
        </encoder>
    </appender>

    <root level="info">
        <appender-ref ref="STDOUT"/>
    </root>

</configuration>
```

To change the log level for specific classes or package names, you can add such a logger entry to the configuration section:

XML

```xml
<configuration>
    ...
    <logger name="io.micronaut.context" level="TRACE"/>
    ...
</configuration>
```

13.4 Logging System

The Micronaut Framework has a notion of a logging system. In short, it is a simple API to be able to set log levels in the logging implementation at runtime. Default implementations are provided for Logback and Log4j2. The behavior of the logging system can be overridden by creating your own implementation of **LoggingSystem** and replace the implementation being used with the **@Replaces** annotation.

14 Language Support

Micronaut supports any JVM language that implements the **Java Annotation Processor** (https://docs.oracle.com/javase/8/docs/api/javax/annotation/processing/package-summary.html) API.

Although Groovy does not support this API, special support has been built using AST transformations. The current list of supported languages is: Java, Groovy, and Kotlin (via the `kapt` tool).

Theoretically any language that supports a way to analyze the AST at compile time could be supported. The **io.micronaut.inject.writer** package includes language-neutral classes that build **BeanDefinition** classes at compile time using the ASM tool.

The following sections cover language-specific features and considerations for using Micronaut.

14.1 Micronaut for Java

For Java, Micronaut uses a Java **BeanDefinitionInjectProcessor** annotation processor to process classes at compile time and produce **BeanDefinition** classes.

The major advantage here is that you pay a slight cost at compile time, but at runtime Micronaut is largely reflection-free, fast, and consumes very little memory.

14.1.1 Using Micronaut with Java 9+

Micronaut is built with Java 8 but works fine with Java 9 and above. The classes that Micronaut generates sit alongside existing classes in the same package, hence do not violate anything regarding the Java module system.

There are some considerations when using Java 9+ with Micronaut.

The javax.annotation package

Using the CLI

If you create your project using the Micronaut CLI, the `javax.annotation` dependency is added to your project automatically if you use Java 9+.

The `javax.annotation`, which includes `@PostConstruct`, `@PreDestroy`, etc. has been moved from the core JDK to a module. In general annotations in this package should be avoided and instead the `jakarta.annotation` equivalents used.

14.1.2 Incremental Annotation Processing with Gradle

Micronaut supports **Gradle incremental annotation processing** (https://docs.gradle.org/current/userguide/java_plugin.html#sec:incremental_annotation_processing) which speeds up builds by compiling only classes that have changed, avoiding a full recompilation.

However, the support is disabled by default since Micronaut allows the definition of custom meta-annotations (to for example define **custom AOP advice**) that need to be configured for processing.

The following example demonstrates how to enable and configure incremental annotation processing for annotations you have defined under the `com.example` package:

Enabling Incremental Annotation Processing

GROOVY

```
tasks.withType(JavaCompile) {
    options.compilerArgs = [
        '-Amicronaut.processing.incremental=true',
        '-Amicronaut.processing.annotations=com.example.*',
    ]
}
```

If you do not enable processing for your custom annotations, they will be ignored by Micronaut, which may break your application.

14.1.3 Using Project Lombok

Project Lombok (https://projectlombok.org) is a popular java library that adds a number of useful AST transformations to the Java language via annotation processors.

Since both Micronaut and Lombok use annotation processors, special care must be taken when configuring Lombok to ensure that the Lombok processor runs **before** Micronaut's processor.

If you use Gradle, add the following dependencies:

Configuring Lombok in Gradle

GROOVY

```
compileOnly 'org.projectlombok:lombok:1.18.12'
annotationProcessor "org.projectlombok:lombok:1.18.12"
...
// Micronaut processor defined after Lombok
annotationProcessor "io.micronaut:micronaut-inject-java"
```

Or if using Maven:

Configuring Lombok in Maven

XML

```
<dependencies>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.12</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
...
<annotationProcessorPaths combine.self="override">
  <path>
    <!-- must precede micronaut-inject-java -->
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.12</version>
  </path>
  <path>
    <groupId>io.micronaut</groupId>
    <artifactId>micronaut-inject-java</artifactId>
    <version>${micronaut.version}</version>
  </path>
    <path>
    <groupId>io.micronaut</groupId>
    <artifactId>micronaut-validation</artifactId>
    <version>${micronaut.version}</version>
  </path>
</annotationProcessorPaths>
```

In both cases (Gradle and Maven) the Micronaut processor must be configured after the Lombok processor. Reversing the order of the declared dependencies will not work.

14.1.4 Configuring an IDE

You can use any IDE to develop Micronaut applications, if you depend on your configured build tool (Gradle or Maven) to build the application.

However, running tests within the IDE is currently possible with **IntelliJ IDEA** (http://jetbrains.com/idea) or Eclipse 4.9 or higher.

See the section on **IDE Setup** in the Quick start for more information on how to configure IntelliJ and Eclipse.

14.1.5 Retaining Parameter Names

By default with Java, the parameter name data for method parameters is not retained at compile time. This can be a problem for Micronaut if you do not define parameter names explicitly and depend on an external JAR that is already compiled.

Consider this interface:

Client Interface

JAVA

```
interface HelloOperations {
    @Get("/hello/{name}")
    String hello(String name);
}
```

At compile time the parameter name `name` is lost and becomes `arg0` when compiled against or read via reflection later. To avoid this problem you have two options. You can either declare the parameter name explicitly:

Client Interface

JAVA

```
interface HelloOperations {
    @Get("/hello/{name}")
    String hello(@QueryValue("name") String name);
}
```

Or alternatively it is recommended that you compile all bytecode with `-parameters` flag to `javac`. See **Obtaining Names of Method Parameters** (https://docs.oracle.com/javase/tutorial/reflect/member/methodparameterreflection.html). For example in `build.gradle`:

build.gradle

GROOVY

```
compileJava.options.compilerArgs += '-parameters'
```

14.2 Micronaut for Groovy

Groovy (http://groovy-lang.org) has first-class support in Micronaut.

Groovy-Specific Modules

Additional modules exist specific to Groovy that improve the overall experience. These are detailed in the table below:

Table 1. Groovy-Specific Modules

Dependency	Description
`io.micronaut:micronaut-inject-groovy`	Includes AST transformations to generate bean definitions. Should be `compileOnly` on your classpath.
`io.micronaut:micronaut-runtime-groovy`	Adds the ability to specify configuration under `src/main/resources` in Groovy format (i.e. `application.groovy`)
`io.micronaut:micronaut-function-groovy`	Includes AST transforms that make it easier to write **Functions** for AWS Lambda

The most common module you need is `micronaut-inject-groovy`, which enables DI and AOP for Groovy classes.

Groovy Support in the CLI

The Micronaut **Command Line Interface** includes special support for Groovy. To create a Groovy application, use the `groovy` lang option. For example:

Create a Micronaut Groovy application

BASH

```
$ mn create-app hello-world --lang groovy
```

The above generates a Groovy project, built with Gradle. Use the `-build maven` flag to generate a project built with Maven instead.

Once you have created an application with the `groovy` feature, commands like `create-controller`, `create-client` etc. generate Groovy files instead of Java. The following example demonstrates this when using interactive mode of the CLI:

Create a bean

BASH

```
$ mn
| Starting interactive mode...
| Enter a command name to run. Use TAB for completion:
mn>

create-bean          create-client        create-controller
create-job           help

mn> create-bean helloBean
| Rendered template Bean.groovy to destination src/main/groovy/hello/world/HelloBean.groovy
```

The above example demonstrates creating a Groovy bean that looks like the following:

Micronaut Bean

GROOVY

```
package hello.world

import javax.inject.Singleton

@Singleton
class HelloBean {

}
```

Groovy automatically imports `groovy.lang.Singleton` which can be confusing as it conflicts with `javax.inject.Singleton`. Make sure you use `javax.inject.Singleton` when declaring a Micronaut singleton bean to avoid surprising behavior.

We can also create a client - don't forget Micronaut can act as a client or a server!

Create a client

BASH

```
mn> create-client helloClient
| Rendered template Client.groovy to destination src/main/groovy/hello/world/HelloClient.groovy
```

Micronaut Client

GROOVY

```
package hello.world

import io.micronaut.http.client.annotation.Client
import io.micronaut.http.annotation.Get
import io.micronaut.http.HttpStatus

@Client("hello")
interface HelloClient {

    @Get
    HttpStatus index()
}
```

Now let's create a controller:

Create a controller

BASH

```
mn> create-controller helloController
| Rendered template Controller.groovy to destination src/main/groovy/hello/world/HelloController.groovy
| Rendered template ControllerSpec.groovy to destination src/test/groovy/hello/world/HelloControllerSpec.groovy
mn>
```

Micronaut Controller

GROOVY

```
package hello.world

import io.micronaut.http.annotation.Controller
import io.micronaut.http.annotation.Get
import io.micronaut.http.HttpStatus

@Controller("/hello")
class HelloController {

    @Get
    HttpStatus index() {
        return HttpStatus.OK
    }
}
```

As you can see from the output from the CLI, a **Spock** (http://spockframework.org) test was also generated for you which demonstrates how to test the controller:

HelloControllerSpec.groovy

GROOVY

```
...
    void "test index"() {
        given:
        HttpResponse response = client.toBlocking().exchange("/hello")

        expect:
        response.status == HttpStatus.OK
    }
...
```

Notice how you use Micronaut both as client and as a server to test itself.

Programmatic Routes with GroovyRouterBuilder

If you prefer to build your routes programmatically (similar to Grails `UrlMappings`), a special `io.micronaut.web.router.GroovyRouteBuilder` exists that has some enhancements to make the DSL better.

The following example shows `GroovyRouteBuilder` in action:

Using GroovyRouteBuilder

GROOVY

```
@Singleton
static class MyRoutes extends GroovyRouteBuilder {

    MyRoutes(ApplicationContext beanContext) {
        super(beanContext)
    }

    @Inject
    void bookResources(BookController bookController, AuthorController authorController) {
        GET(bookController) {
            POST("/hello{/message}", bookController.&hello)  1
        }
        GET(bookController, ID) {  2
            GET(authorController)
        }
    }
}
```

1 You can use injected controllers to create routes by convention and Groovy method references to create routes to methods

2 The `ID` property can be used to reference include an `{id}` URI variable

The above example results in the following routes:

- `/book` - Maps to `BookController.index()`
- `/book/hello/{message}` - Maps to `BookController.hello(String)`
- `/book/{id}` - Maps to `BookController.show(String id)`
- `/book/{id}/author` - Maps to `AuthorController.index`

Using GORM in a Groovy application

GORM (http://gorm.grails.org) is a data access toolkit originally created as part of Grails. It supports multiple database types. The following table summarizes the modules needed to use GORM, and links to documentation.

Table 2. GORM Modules

Dependency	Description
`io.micronaut.groovy:micronaut-hibernate-gorm`	Configures **GORM for Hibernate** (http://gorm.grails.org/latest/hibernate/manual) for Groovy applications. See the **Hibernate Support** docs
`io.micronaut.groovy:micronaut-mongo-gorm`	Configures **GORM for MongoDB** (http://gorm.grails.org/latest/mongodb/manual) for Groovy applications. See the **Mongo Support** docs.
`io.micronaut.groovy:micronaut-neo4j-gorm`	Configures **GORM for Neo4j** (http://gorm.grails.org/latest/neo4j/manual) for Groovy applications. See the **Neo4j Support** docs.

Once you have configured a GORM implementation per the instructions linked in the table above you can use all features of GORM.

GORM Data Services (http://gorm.grails.org/latest/hibernate/manual/index.html#dataServices) can also participate in dependency injection and life cycle methods:

GORM Data Service VehicleService.groovy

GROOVY

```
@Service(Vehicle)
abstract class VehicleService {
    @PostConstruct
    void init() {
       // do something on initialization
    }

    abstract Vehicle findVehicle(@NotBlank String name)

    abstract Vehicle saveVehicle(@NotBlank String name)
}
```

You can also define the service as an interface instead of an abstract class to have GORM implement the methods for you.

Serverless Functions with Groovy

A microservice application is just one way to use Micronaut. You can also use it for serverless functions like on AWS Lambda.

With the `function-groovy` module, Micronaut features enhanced support for functions written in Groovy.

See the section on **Serverless Functions** for more information.

14.3 Micronaut for Kotlin

The **Command Line Interface** for Micronaut includes special support for Kotlin. To create a Kotlin application use the `kotlin` lang option. For example:

Create a Micronaut Kotlin application

```
$ mn create-app hello-world --lang kotlin
```

BASH

Support for Kotlin in Micronaut is built upon the **Kapt** (https://kotlinlang.org/docs/reference/kapt.html) compiler plugin, which includes support for Java annotation processors. To use Kotlin in your Micronaut application, add the proper dependencies to configure and run kapt on your `kt` source files. Kapt creates Java "stub" classes for your Kotlin classes, which can then be processed by Micronaut's Java annotation processor. The stubs are not included in the final compiled application.

Learn more about kapt and its features from the **official documentation.** (https://kotlinlang.org/docs/reference/kapt.html)

The Micronaut annotation processors are declared in the `kapt` scope when using Gradle. For example:

Example build.gradle

```
dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib-jdk8:$kotlinVersion"  1
    compile "org.jetbrains.kotlin:kotlin-reflect:$kotlinVersion"
    kapt "io.micronaut:micronaut-inject-java"  2

    kaptTest "io.micronaut:micronaut-inject-java"  3
    ...
}
```

GROOVY

1. Add the Kotlin standard libraries
2. Add the `micronaut-inject-java` dependency under the `kapt` scope, so classes in `src/main` are processed
3. Add the `micronaut-inject-java` dependency under the `kaptTest` scope, so classes in `src/test` are processed.

With a `build.gradle` file similar to the above, you can now run your Micronaut application using the `run` task (provided by the Application plugin):

```
$ ./gradlew run
```

BASH

An example controller written in Kotlin can be seen below:

src/main/kotlin/example/HelloController.kt

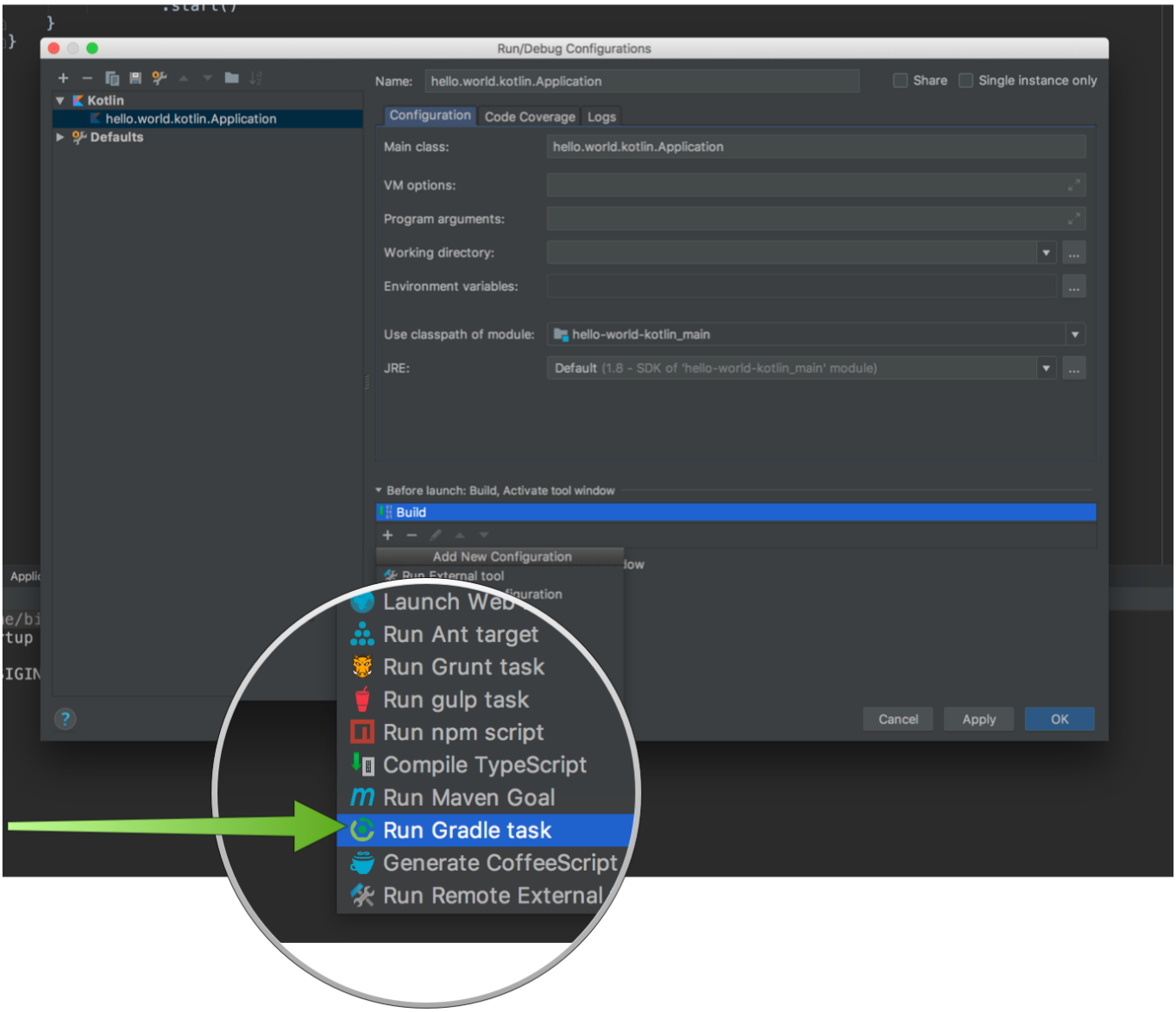
```
package example

import io.micronaut.http.annotation.*

@Controller("/")
class HelloController {

    @Get("/hello/{name}")
    fun hello(name: String): String {
        return "Hello $name"
    }
}
```

KOTLIN

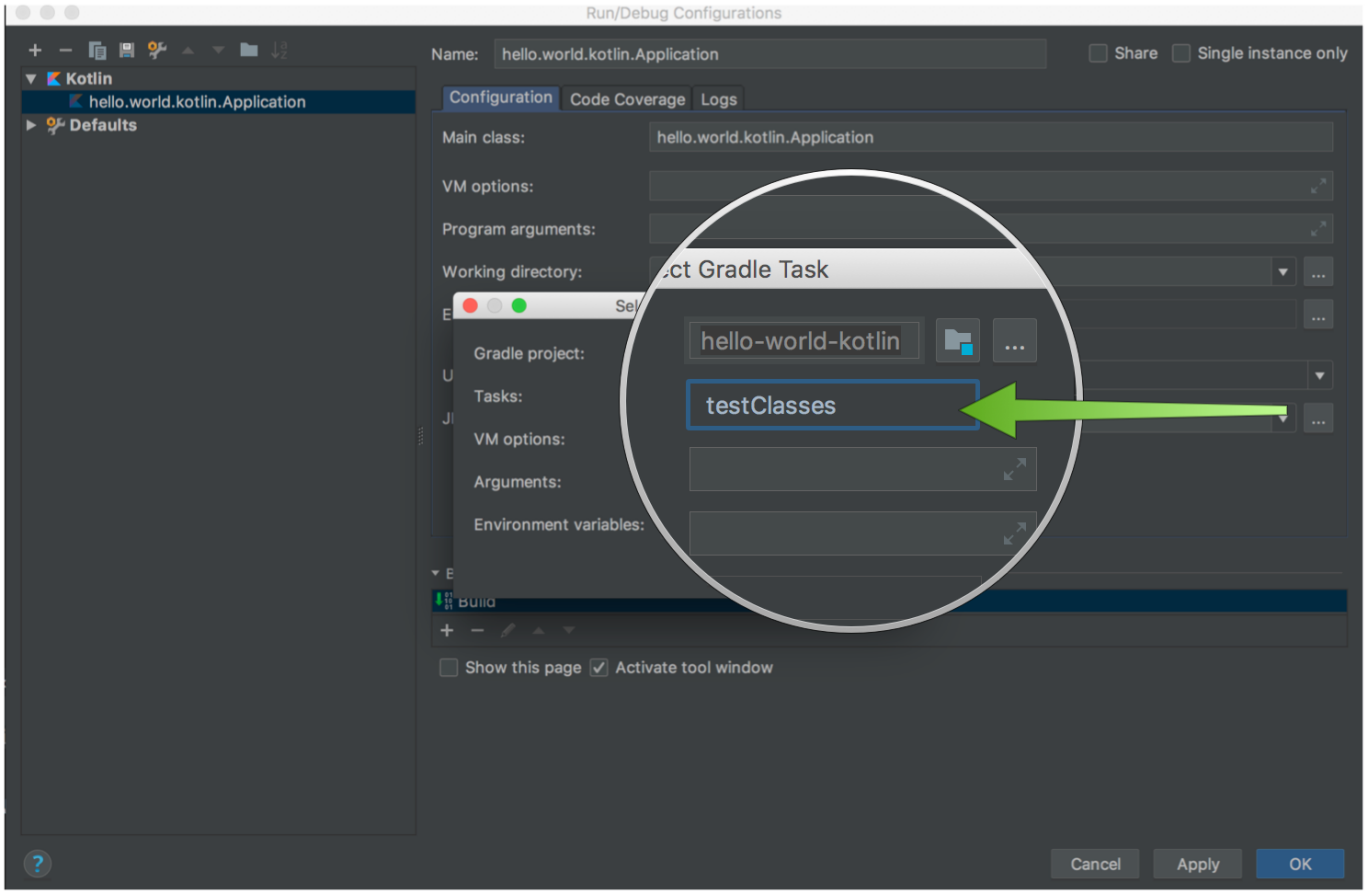
14.3.1 Kotlin, Kapt and IntelliJ

As of this writing, IntelliJ's built-in compiler does not directly support Kapt and annotation processing. You must instead configure Intellij to run Gradle (or Maven) compilation as a build step before running your tests or application class.

First, edit the run configuration for tests or for the application and select "Run Gradle task" as a build step:

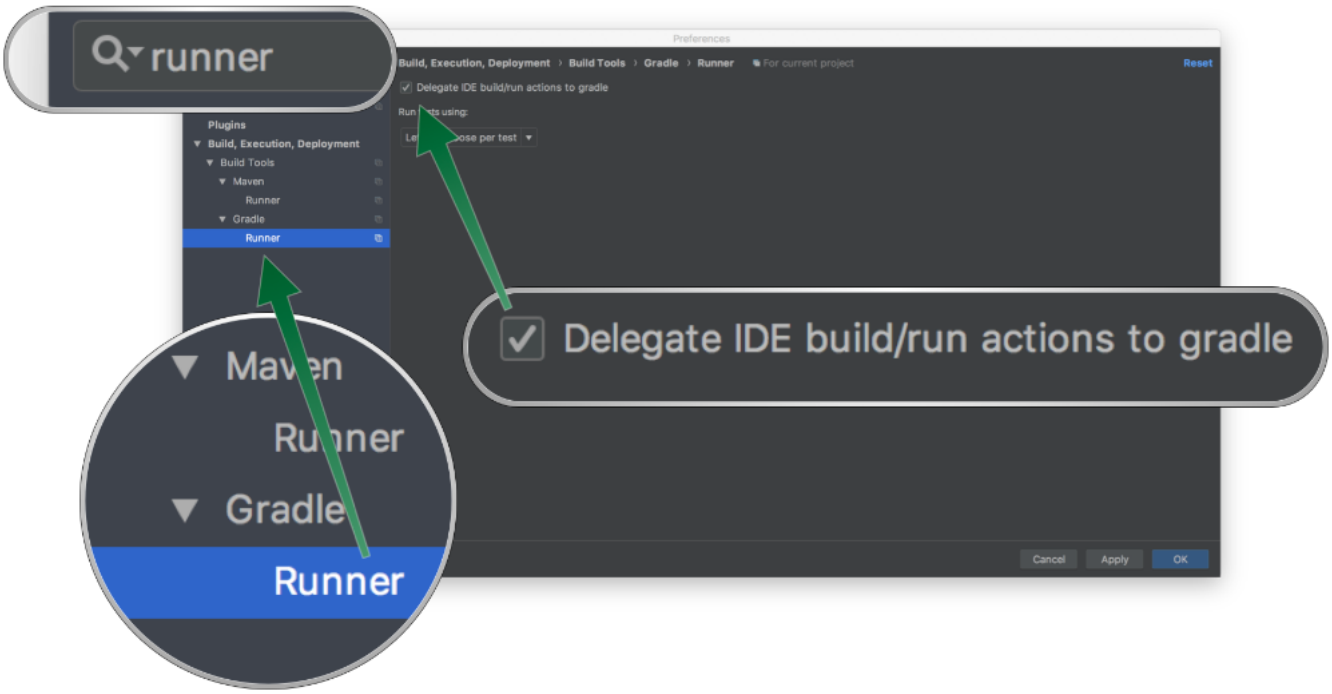


Then add the `classes` task as task to execute for the application or for tests the `testClasses` task:



Now when you run tests or start the application, Micronaut will generate classes at compile time.

Alternatively, you can [delegate IntelliJ build/run actions to Gradle](https://www.jetbrains.com/help/idea/gradle.html#delegate_build_gradle) (https://www.jetbrains.com/help/idea/gradle.html#delegate_build_gradle) completely:



14.3.2 Incremental Annotation Processing with Gradle and Kapt

To enable Gradle incremental annotation processing with Kapt, the arguments as specified in **Incremental Annotation Processing with Gradle** must be sent to Kapt.

The following example demonstrates how to enable and configure incremental annotation processing for annotations you have defined under the `com.example` and `io.example` packages:

Enabling Incremental Annotation Processing in Kapt

KOTLIN

```
kapt {
    arguments {
        arg("micronaut.processing.incremental", true)
        arg("micronaut.processing.annotations", "com.example.*,io.example.*")
    }
}
```

If you do not enable processing for your custom annotations, they will be ignored by Micronaut, which may break your application.

14.3.3 Kotlin and AOP Advice

Micronaut provides a compile-time AOP API that does not use reflection. When you use any Micronaut **AOP Advice**, it creates a subclass at compile-time to provide the AOP behaviour. This can be a problem because Kotlin classes are final by default. If the application was created with the Micronaut CLI, the Kotlin **all-open** (https://kotlinlang.org/docs/reference/compiler-plugins.html#all-open-compiler-plugin) plugin is configured for you to automatically change your classes to `open` when an AOP annotation is used. To configure it yourself, add the **Around** class to the list of supported annotations.

If you prefer not to or cannot use the `all-open` plugin, you must declare the classes that are annotated with an AOP annotation to be open:

JAVA

```
import io.micronaut.http.annotation.Controller
import io.micronaut.http.annotation.Get
import io.micronaut.http.HttpStatus
import io.micronaut.validation.Validated
import javax.validation.constraints.NotBlank

@Validated
@Controller("/email")
open class EmailController {  1

    @Get("/send")
    fun index(@NotBlank recipient: String,  1
              @NotBlank subject: String): HttpStatus {
        return HttpStatus.OK
    }
}
```

1 if you use `@Validated` AOP Advice, you need to use `open` at class and method level.

The `all-open` plugin does not handle methods. If you declare an AOP annotation on a method, you must manually declare it as open.

14.3.4 Kotlin and Retaining Parameter Names

Like with Java, the parameter name data for method parameters is not retained at compile time when using Kotlin. This can be a problem for Micronaut if you do not define parameter names explicitly and depend on an external JAR that is already compiled.

To enable retention of parameter name data with Kotlin, set the `javaParameters` option to `true` in your `build.gradle`:

configuration in Gradle

GROOVY

```
compileTestKotlin {
    kotlinOptions {
        jvmTarget = '1.8'
        javaParameters = true
    }
}
```

Or if using Maven configure the Micronaut Maven Plugin accordingly:

configuration in Maven

XML

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <!-- ... -->
  <build>
    <plugins>
      <!-- ... -->
      <plugin>
        <artifactId>kotlin-maven-plugin</artifactId>
        <groupId>org.jetbrains.kotlin</groupId>
        <configuration>
            <javaParameters>true</javaParameters>
            <!-- ... -->
        </configuration>
        <!-- ... -->
      </plugin>
      <!-- ... -->
    </plugins>
  </build>
</project>
```

14.3.5 Coroutines Support

Kotlin coroutines allow you to create asynchronous applications with imperative style code. A Micronaut controller action can be a `suspend` function:

Controller suspend function example

KOTLIN

```
@Get("/simple", produces = [MediaType.TEXT_PLAIN])
suspend fun simple(): String {  1
    return "Hello"
}
```

Copy to Clipboard

1 The function is marked as `suspend`, though in reality it won't be suspended.

Controller suspend function example

KOTLIN

```
@Get("/delayed", produces = [MediaType.TEXT_PLAIN])
suspend fun delayed(): String {  1
    delay(1)  2
    return "Delayed"
}
```

Copy to Clipboard

1 The function is marked as `suspend`.

2 The `delay` is called to make sure that a function is suspended and the response is returned from a different thread.

Controller suspend function example

KOTLIN

```
@Status(HttpStatus.CREATED)  1
@Get("/status")
suspend fun status() {
}
```

Copy to Clipboard

1 `suspend` function also works when all we want is to return a status.

Controller suspend function example

KOTLIN

```
@Status(HttpStatus.CREATED)
@Get("/statusDelayed")
suspend fun statusDelayed() {
    delay(1)
}
```

Copy to Clipboard

You can also use `Flow` type for streaming server and client. A streaming controller can return `Flow`, for example:

Streaming JSON on the Server with Flow

KOTLIN

```
    @Get(value = "/headlinesWithFlow", processes = [MediaType.APPLICATION_JSON_STREAM])
    internal fun streamHeadlinesWithFlow(): Flow<Headline> =  1
        flow {  2
            repeat(100) {  3
                with (Headline()) {
                    text = "Latest Headline at ${ZonedDateTime.now()}"
                    emit(this)  4
                    delay(1_000)  5
                }
            }
        }
```

Copy to Clipboard

1 A method `streamHeadlinesWithFlow` is defined that produces `application/x-json-stream`

2 A `Flow` is created using `flow`

3 This `Flow` emits 100 messages

4 Emitting happens with `emit suspend` function

5 There is a one second *delay* between messages

A streaming client can simply return a `Flow`, for example:

Streaming client with Flow

KOTLIN

```
import io.micronaut.http.MediaType
import io.micronaut.http.annotation.Get
import io.micronaut.http.client.annotation.Client
import kotlinx.coroutines.flow.Flow

@Client("/streaming")
interface HeadlineFlowClient {

    @Get(value = "/headlinesWithFlow", processes = [MediaType.APPLICATION_JSON_STREAM])  1
    fun streamFlow(): Flow<Headline>  2
}
```

Copy to Clipboard

1 The `@Get` method is defined as processing responses of type `APPLICATION_JSON_STREAM`

2 The return type is `Flow`

14.3.6 Reactive Context Propagation

Micronaut supports context propagation from Reactor's context to coroutine context. To enable this propagation you need to include following dependency:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>org.jetbrains.kotlinx</groupId>
    <artifactId>kotlinx-coroutines-reactor</artifactId>
</dependency>
```

Copy to Clipboard

For more detailed information on how to use the library you can find at the official **documentation (https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-reactor/kotlinx.coroutines.reactor/-reactor-context/index.html)**.

Following example shows how to propagate Reactor context from the HTTP filter to the controller's coroutine:

Simple filter which writes into Reactor's context

KOTLIN

```
@Filter(Filter.MATCH_ALL_PATTERN)
class ReactorHttpServerFilter : HttpServerFilter {

    override fun doFilter(request: HttpRequest<*>, chain: ServerFilterChain): Publisher<MutableHttpResponse<*>> {
        val trackingId = request.headers["X-TrackingId"] as String
        return Mono.from(chain.proceed(request)).contextWrite {
            it.put("reactorTrackingId", trackingId)
        }
    }

}
```

Copy to Clipboard

Access Reactor context by retrieving `ReactorContext` from the coroutine context:

Reading Reactor context in the coroutine

KOTLIN

```
    @Get("/data")
    suspend fun getTracingId(request: HttpRequest<*>): String {
        val reactorContextView = currentCoroutineContext()[ReactorContext.Key]!!.context
        return reactorContextView.get("reactorTrackingId") as String
    }
```

Copy to Clipboard

It's possible to use coroutines Reactor integration to create a filter using a suspended function:

Suspended function filter which writes into Reactor's context

KOTLIN

```
@Filter(Filter.MATCH_ALL_PATTERN)
class SuspendHttpServerFilter : CoroutineHttpServerFilter {

    override suspend fun filter(request: HttpRequest<*>, chain: ServerFilterChain): MutableHttpResponse<*> {
        val trackingId = request.headers["X-TrackingId"] as String
        return withContext(Context.of("suspendTrackingId", trackingId).asCoroutineContext()) {
            chain.next(request)
        }
    }

}

interface CoroutineHttpServerFilter : HttpServerFilter {

    suspend fun filter(request: HttpRequest<*>, chain: ServerFilterChain): MutableHttpResponse<*>

    override fun doFilter(request: HttpRequest<*>, chain: ServerFilterChain): Publisher<MutableHttpResponse<*>> {
        return mono {
            filter(request, chain)
        }
    }

}

suspend fun ServerFilterChain.next(request: HttpRequest<*>): MutableHttpResponse<*> {
    return this.proceed(request).asFlow().single()
}
```

Copy to Clipboard

14.4 Micronaut for GraalVM

GraalVM (https://www.graalvm.org) is a new universal virtual machine from Oracle that supports a polyglot runtime environment and the ability to compile Java applications to native machine code.

Any Micronaut application can be run using the GraalVM JVM, however special support has been added to Micronaut to support running Micronaut applications using GraalVM's `native-image` tool (https://www.graalvm.org/reference-manual/native-image/).

Micronaut currently supports GraalVM version {graalVersion} and the team is improving the support in every new release. Don't hesitate to report issues (https://github.com/micronaut-projects/micronaut-core/issues) however if you find any problem.

Many of Micronaut's modules and third-party libraries have been verified to work with GraalVM: HTTP server, HTTP client, Function support, Micronaut Data JDBC and JPA, Service Discovery, RabbitMQ, Views, Security, Zipkin, etc. Support for other modules is evolving and will improve over time.

Getting Started

Use of GraalVM's `native-image` tool is only supported in Java or Kotlin projects. Groovy relies heavily on reflection which is only partially supported by GraalVM.

To start using GraalVM, first install the GraalVM SDK via the Getting Started (https://www.graalvm.org/docs/getting-started/) instructions or using Sdkman! (https://sdkman.io/).

14.4.1 Microservices as GraalVM native images

Getting Started with Micronaut and GraalVM

Since Micronaut 2.2, any Micronaut application is ready to be built into a native image using the Micronaut Gradle or Maven plugins. To get started, create a new application:

Creating a GraalVM Native Microservice

BASH

```
$ mn create-app hello-world
```

You can use `--build maven` for a Maven build.

Building a Native Image Using Docker

To build your native image using Gradle and Docker, run:

Building a Native Image with Docker and Gradle

BASH

```
$ ./gradlew dockerBuildNative
```

To build your native image using Maven and Docker, run:

Building a Native Image with Docker and Maven

BASH

```
$ ./mvnw package -Dpackaging=docker-native
```

Building a Native Image Without Using Docker

To build your native image without using Docker, install the GraalVM SDK via the **Getting Started** (https://www.graalvm.org/docs/getting-started/) instructions or using **Sdkman!** (https://sdkman.io/):

Installing GraalVM {graalVersion} with SDKman

BASH

```
$ sdk install java {graalVersion}.r11-grl
$ sdk use java {graalVersion}.r11-grl
```

The `native-image` tool was extracted from the base GraalVM distribution and is available as a plugin. To install it, run:

Installing `native-image` tool

BASH

```
$ gu install native-image
```

Now you can build a native image with Gradle by running the `nativeImage` task:

Creating native image with Gradle

BASH

```
$ ./gradlew nativeImage
```

The native image will be built in the `build/native-image` directory.

To create a native image with Maven and the Micronaut Maven plugin, use the `native-image` packaging format:

Creating native image with Maven

BASH

```
$ ./mvnw package -Dpackaging=native-image
```

which builds the native image in the `target` directory.

You can then run the native image from the directory where you built it.

Run native image

BASH

```
$ ./hello-world
```

Understanding Micronaut and GraalVM

Micronaut itself does not rely on reflection or dynamic classloading, so it works automatically with GraalVM native, however certain third-party libraries used by Micronaut may require additional input about uses of reflection.

Micronaut includes an annotation processor that helps to generate the `reflect-config.json` metadata files that are automatically picked up by the `native-image` tool:

Gradle **Maven**

MAVEN

```
<annotationProcessorPaths>
    <path>
        <groupId>io.micronaut</groupId>
        <artifactId>micronaut-graal</artifactId>
    </path>
</annotationProcessorPaths>
```

Copy to Clipboard

This processor generates:

- A `reflect-config.json` file in the `META-INF/native-image` directory in your build classes directory (`target/classes` with Maven and typically `build/classes/java/main` with Gradle).

For example the following class:

```
package example;

import io.micronaut.core.annotation.ReflectiveAccess;

@ReflectiveAccess
class Test {
    ...
}
```

JAVA

The above example results in the public methods, declared fields and declared constructors of `example.Test` being included in `reflect-config.json`.

If you have more advanced requirements and only wish to include certain fields or methods, use the annotation on any constructor, field or method to include only the specific field, constructor or method.

To provide your own `reflect.json`, add one to `src/main/graal/reflect.json` and it will be automatically picked up.

Adding Additional Classes for Reflective Access

To inform Micronaut of additional classes to be included in the generated `reflect.json` file at compile time, either annotate a class with **@ReflectiveAccess** or **@TypeHint**.

Both allows for reflective access, and the latter is typically used on a module or `Application` class to include classes that are needed reflectively. For example, the following is from Micronaut's Jackson module:

```
@TypeHint(
    value = {  1
        PropertyNamingStrategy.UpperCamelCaseStrategy.class,
        ArrayList.class,
        LinkedHashMap.class,
        HashSet.class
    },
    accessType = TypeHint.AccessType.ALL_DECLARED_CONSTRUCTORS  2
)
```

JAVA

1 The `value` member specifies which classes require reflection.

2 The `accessType` member specifies if only classloading access is needed or whether full reflection on all public members is needed.

Generating Native Images

GraalVM's `native-image` command generates native images. You can use this command manually to generate your native image. For example:

The `native-image` command

```
native-image --class-path build/libs/hello-world-0.1-all.jar  1
```

BASH

1 The `class-path` argument refers to the Micronaut shaded JAR

Once the image is built, run the application using the native image name:

Running the Native Application

```
$ ./hello-world
15:15:15.153 [main] INFO  io.micronaut.runtime.Micronaut - Startup completed in 14ms. Server Running: http://localhost:
```

BASH

As you can see, the native image startup completes in milliseconds, and memory consumption does not include the overhead of the JVM (a native Micronaut application runs with just 20mb of memory).

Resource file generation

Starting in Micronaut 3.0 the automatic generation of the `resource-config.json` file is now part of the **Gradle** (https://github.com/micronaut-projects/micronaut-gradle-plugin) and **Maven** (https://github.com/micronaut-projects/micronaut-maven-plugin) plugins.

14.4.2 GraalVM and Micronaut FAQ

How does Micronaut manage to run on GraalVM?

Micronaut features a Dependency Injection and Aspect-Oriented Programming runtime that uses no reflection. This makes it easier for Micronaut applications to run on GraalVM since there are **limitations** (https://github.com/oracle/graal/blob/master/substratevm/LIMITATIONS.md) particularly around **reflection** (https://github.com/oracle/graal/blob/master/substratevm/REFLECTION.md) on SubstrateVM.

How can I make a Micronaut application that uses picocli run on GraalVM?

Picocli provides a `picocli-codegen` module with a tool for generating a GraalVM reflection configuration file. The tool can be run **manually** (https://picocli.info/picocli-on-graalvm.html) or automatically as part of the build. The module's **README** (https://github.com/remkop/picocli/tree/master/picocli-codegen) has usage instructions with code snippets for configuring Gradle and Maven to generate a `cli-reflect.json` file automatically as part of the build. Add the generated file to the `-H:ReflectionConfigurationFiles` option when running the `native-image` tool.

What about other Third-Party Libraries?

Micronaut cannot guarantee that third-party libraries work on GraalVM SubstrateVM, that is down to each individual library to implement support.

I Get a "Class XXX is instantiated reflectively…" Exception. What do I do?

If you get an error such as:

```
Class myclass.Foo[] is instantiated reflectively but was never registered. Register the class by using org.graalvm.nativeimage.RuntimeReflection
```

You may need to manually tweak the generated `reflect.json` file. For regular classes you need to add an entry into the array:

JSON

```
[
    {
        "name" : "myclass.Foo",
        "allDeclaredConstructors" : true
    },
    ...
]
```

For arrays this must use the Java JVM internal array representation. For example:

JSON

```
[
    {
        "name" : "[Lmyclass.Foo;",
        "allDeclaredConstructors" : true
    },
    ...
]
```

What if I want to set the heap's maximum size with `-Xmx`, but I get an `OutOfMemoryError`?

If you set the maximum heap size in the Dockerfile that you use to build your native image, you will probably get a runtime error like this:

```
java.lang.OutOfMemoryError: Direct buffer memory
```

The problem is that Netty tries to allocate 16MB of memory per chunk with its default settings for `io.netty.allocator.pageSize` and `io.netty.allocator.maxOrder`:

JAVA

```
int defaultChunkSize = DEFAULT_PAGE_SIZE << DEFAULT_MAX_ORDER; // 8192 << 11 = 16MB
```

The simplest solution is to specify `io.netty.allocator.maxOrder` explicitly in your Dockerfile's entrypoint. A working example with `-Xmx64m`:

DOCKERFILE

```
ENTRYPOINT ["/app/application", "-Xmx64m", "-Dio.netty.allocator.maxOrder=8"]
```

To go further, you can also experiment with `io.netty.allocator.numHeapArenas` or `io.netty.allocator.numDirectArenas`. You can find more information about Netty's `PooledByteBufAllocator` in the **official documentation** (https://netty.io/4.1/api/io/netty/buffer/PooledByteBufAllocator.html).

15 Management & Monitoring

Using the CLI

If you create your project using the Micronaut CLI, supply the `management` feature to configure the management endpoints in your project:

```
$ mn create-app my-app --features management
```

Inspired by Spring Boot and Grails, the Micronaut `management` dependency adds support for monitoring of your application via **endpoints**: special URIs that return details about the health and state of your application. The `management` endpoints are also integrated with Micronaut's `security` dependency, allowing for sensitive data to be restricted to authenticated users in your security system (see **Built-in Endpoints Access** (https://micronaut-projects.github.io/micronaut-security/{micronautSecurityVersion}/guide/#builtInEndpointsAccess) in the Security section).

To use the `management` features described in this section, add this dependency to your build:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut</groupId>
    <artifactId>micronaut-management</artifactId>
</dependency>
```

Copy to Clipboard

15.1 Creating Endpoints

In addition to the **Built-In Endpoints**, the `management` dependency also provides support for creating custom endpoints. These can be enabled and configured like the built-in endpoints, and can be used to retrieve and return any metrics or other application data.

15.1.1 The Endpoint Annotation

An Endpoint can be created by annotating a class with the **Endpoint** annotation, and supplying it with (at minimum) an endpoint id.

FooEndpoint.java

JAVA

```
@Endpoint("foo")
class FooEndpoint {
    ...
}
```

If a single `String` argument is supplied to the annotation, it is used as the endpoint id.

It is possible to supply additional (named) arguments to the annotation. Other possible arguments to `@Endpoint` are described in the table below:

Table 1. Endpoint Arguments

Argument	Description	Endpoint Example
`String id`	The endpoint id (or name)	`@Endpoint(id = "foo")`
`String prefix`	Prefix used for configuring the endpoint (see **Endpoint Configuration**)	`@Endpoint(prefix = "foo")`
`boolean defaultEnabled`	Sets whether the endpoint is enabled when no configuration is set (see **Endpoint Configuration**)	`@Endpoint(defaultEnabled = false)`
`boolean defaultSensitive`	Sets whether the endpoint is sensitive if no configuration is set (see **Endpoint Configuration**)	`@Endpoint(defaultSensitive = false)`

Example of custom Endpoint

The following example `Endpoint` class creates an endpoint accessible at `/date`:

CurrentDateEndpoint

Java Groovy Kotlin

JAVA

```
import io.micronaut.management.endpoint.annotation.Endpoint;

@Endpoint(id = "date",
          prefix = "custom",
          defaultEnabled = true,
          defaultSensitive = false)
public class CurrentDateEndpoint {

    //.. endpoint methods

}
```

Copy to Clipboard

15.1.2 Endpoint Methods

Endpoints respond to `GET` ("read"), `POST` ("write") and `DELETE` ("delete") requests. To return a response from an endpoint, annotate its public method(s) with one of the following annotations:

Table 1. Endpoint Method Annotations

Annotation	Description
Read	Responds to `GET` requests
Write	Responds to `POST` requests

Annotation	Description
Delete	Responds to `DELETE` requests

Read Methods

Annotating a method with the **Read** annotation causes it to respond to `GET` requests.

CurrentDateEndpoint

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.management.endpoint.annotation.Endpoint;

import io.micronaut.management.endpoint.annotation.Read;

@Endpoint(id = "date",
          prefix = "custom",
          defaultEnabled = true,
          defaultSensitive = false)
public class CurrentDateEndpoint {

    private Date currentDate;

    @Read
    public Date currentDate() {
        return currentDate;
    }

}
```

Copy to Clipboard

The above method responds to the following request:

BASH

```
$ curl -X GET localhost:55838/date

1526085903689
```

The **Read** annotation accepts an optional `produces` argument, which sets the media type returned from the method (default is `application/json`):

CurrentDateEndpoint

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.management.endpoint.annotation.Endpoint;

import io.micronaut.management.endpoint.annotation.Read;

import io.micronaut.http.MediaType;
import io.micronaut.management.endpoint.annotation.Selector;

@Endpoint(id = "date",
          prefix = "custom",
          defaultEnabled = true,
          defaultSensitive = false)
public class CurrentDateEndpoint {

    private Date currentDate;

    @Read(produces = MediaType.TEXT_PLAIN)  1
    public String currentDatePrefix(@Selector String prefix) {
        return prefix + ": " + currentDate;
    }

}
```

Copy to Clipboard

1 Supported media types are represented by **MediaType**

The above method responds to the following request:

BASH

```
$ curl -X GET localhost:8080/date/the_date_is

the_date_is: Fri May 11 19:24:21 CDT
```

Write Methods

Annotating a method with the **Write** annotation causes it to respond to `POST` requests.

CurrentDateEndpoint

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.management.endpoint.annotation.Endpoint;

import io.micronaut.management.endpoint.annotation.Write;

import io.micronaut.http.MediaType;
import io.micronaut.management.endpoint.annotation.Selector;

@Endpoint(id = "date",
          prefix = "custom",
          defaultEnabled = true,
          defaultSensitive = false)
public class CurrentDateEndpoint {

    private Date currentDate;

    @Write
    public String reset() {
        currentDate = new Date();

        return "Current date reset";
    }

}
```

Copy to Clipboard

The above method responds to the following request:

BASH

```
$ curl -X POST http://localhost:39357/date

Current date reset
```

The **Write** annotation accepts an optional `consumes` argument, which sets the media type accepted by the method (default is `application/json`):

MessageEndpoint

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.Requires;
import io.micronaut.management.endpoint.annotation.Endpoint;

import io.micronaut.management.endpoint.annotation.Write;

import io.micronaut.http.MediaType;

@Endpoint(id = "message", defaultSensitive = false)
public class MessageEndpoint {

    String message;

    @Write(consumes = MediaType.APPLICATION_FORM_URLENCODED, produces = MediaType.TEXT_PLAIN)
    public String updateMessage(String newMessage) {
        this.message = newMessage;

        return "Message updated";
    }

}
```

Copy to Clipboard

The above method responds to the following request:

BASH

```
$ curl -X POST http://localhost:65013/message -H 'Content-Type: application/x-www-form-urlencoded' -d $'newMessage=A ne

Message updated
```

Delete Methods

Annotating a method with the **Delete** annotation causes it to respond to `DELETE` requests.

MessageEndpoint

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.context.annotation.Requires;
import io.micronaut.management.endpoint.annotation.Endpoint;

import io.micronaut.management.endpoint.annotation.Delete;

@Endpoint(id = "message", defaultSensitive = false)
public class MessageEndpoint {

    String message;

    @Delete
    public String deleteMessage() {
        this.message = null;

        return "Message deleted";
    }

}
```

Copy to Clipboard

The above method responds to the following request:

BASH

```
$ curl -X DELETE http://localhost:65013/message

Message deleted
```

15.1.3 Endpoint Sensitivity

Endpoint sensitivity can be controlled for the entire endpoint through the endpoint annotation and configuration. Individual methods can be configured independently from the endpoint as a whole, however. The **@Sensitive** annotation can be applied to methods to control their sensitivity.

AlertsEndpoint

Java | Groovy | Kotlin

JAVA

```
import io.micronaut.http.MediaType;
import io.micronaut.management.endpoint.annotation.Delete;
import io.micronaut.management.endpoint.annotation.Endpoint;
import io.micronaut.management.endpoint.annotation.Read;
import io.micronaut.management.endpoint.annotation.Sensitive;
import io.micronaut.management.endpoint.annotation.Write;

import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

@Endpoint(id = "alerts", defaultSensitive = false)  1
public class AlertsEndpoint {

    private final List<String> alerts = new CopyOnWriteArrayList<>();

    @Read
    List<String> getAlerts() {
        return alerts;
    }

    @Delete
    @Sensitive(true)  2
    void clearAlerts() {
        alerts.clear();
    }

    @Write(consumes = MediaType.TEXT_PLAIN)
    @Sensitive(property = "add.sensitive", defaultValue = true)  3
    void addAlert(String alert) {
        alerts.add(alert);
    }
}
```

```
1 The endpoint is not sensitive by default, and the default prefix of endpoints is used.
2 This method is always sensitive, regardless of any other factors
3 The property value is appended to the prefix and id to lookup a configuration value
```

Copy to Clipboard

If the configuration key `endpoints.alerts.add.sensitive` is set, that value determines the sensitivity of the `addAlert` method.

1. `endpoint` is the first token because that is the default value for `prefix` in the endpoint annotation and is not set explicitly in this example.
2. `alerts` is the next token because that is the endpoint id
3. `add.sensitive` is the next token because that is the value set to the `property` member of the **@Sensitive** annotation.

If the configuration key is not set, the `defaultValue` is used (defaults to `true`).

15.1.4 Endpoint Configuration

Endpoints with the `endpoints` prefix can be configured through their default endpoint id. If an endpoint exists with the id of `foo`, it can be configured through `endpoints.foo`. In addition, default values can be provided through the `all` prefix.

For example, consider the following endpoint.

FooEndpoint.java

JAVA

```java
@Endpoint("foo")
class FooEndpoint {
    ...
}
```

By default the endpoint is enabled. To disable it, set `endpoints.foo.enabled` to false. If `endpoints.foo.enabled` is not set and `endpoints.all.enabled` is `false`, the endpoint will be disabled.

The configuration values for the endpoint override those for `all`. If `endpoints.foo.enabled` is `true` and `endpoints.all.enabled` is `false`, the endpoint will be enabled.

For all endpoints, the following configuration values can be set.

YAML

```yaml
endpoints:
  <any endpoint id>:
    enabled: Boolean
    sensitive: Boolean
```

The base path for all endpoints is `/` by default. If you prefer the endpoints to be available under a different base path, configure `endpoints.all.path`. For example, if the value is set to `/endpoints`, the foo endpoint will be accessible at `/endpoints/foo`.

15.2 Built-In Endpoints

When the `management` dependency is added to your project, the following built-in endpoints are enabled by default:

Table 1. Default Endpoints

Endpoint	URI	Description
BeansEndpoint	`/beans`	Returns information about the loaded bean definitions in the application (see **BeansEndpoint**)
HealthEndpoint	`/health`	Returns information about the "health" of the application (see **HealthEndpoint**)
InfoEndpoint	`/info`	Returns static information from the state of the application (see **InfoEndpoint**)
LoggersEndpoint	`/loggers`	Returns information about available loggers and permits changing the configured log level (see **LoggersEndpoint**)
MetricsEndpoint	`/metrics`	Return the **application metrics**. Requires the `micrometer-core` configuration on the classpath.
RefreshEndpoint	`/refresh`	Refreshes the application state (see **RefreshEndpoint**)
RoutesEndpoint	`/routes`	Returns information about URIs available to be called for your application (see **RoutesEndpoint**)

Endpoint	URI	Description
EnvironmentEndpoint	/env	Returns information about the environment and its property sources (see EnvironmentEndpoint)
ThreadDumpEndpoint	/threaddump	Returns information about the current threads in the application.

In addition, the following built-in endpoint(s) are provided by the `management` dependency but are not enabled by default:

Table 2. Disabled Endpoints

Endpoint	URI	Description
CachesEndpoint	/caches	Returns information about the caches and permits invalidating them (see CachesEndpoint)
ServerStopEndpoint	/stop	Shuts down the application server (see ServerStopEndpoint)

It is possible to open all endpoints for unauthenticated access defining `endpoints.all.sensitive: false` but this should be used with care because private and sensitive information will be exposed.

Management Port

By default, all management endpoints are exposed over the same port as the application. You can alter this behaviour by specifying the `endpoints.all.port` setting:

YAML

```
endpoints:
  all:
    port: 8085
```

In the above example the management endpoints are exposed only over port 8085.

JMX

Micronaut provides functionality to register endpoints with JMX. See the section on **JMX** to get started.

15.2.1 The Beans Endpoint

The beans endpoint returns information about the loaded bean definitions in the application. The bean data returned by default is an object where the key is the bean definition class name and the value is an object of properties about the bean.

To execute the beans endpoint, send a GET request to /beans.

Configuration

To configure the beans endpoint, supply configuration through `endpoints.beans`.

Beans Endpoint Configuration Example

YAML

```
endpoints:
  beans:
    enabled: Boolean
    sensitive: Boolean
```

Customization

The beans endpoint is composed of a bean definition data collector and a bean data implementation. The bean definition data collector (BeanDefinitionDataCollector) is responsible for returning a publisher that returns the data used in the response. The bean definition data (BeanDefinitionData) is responsible for returning data about an individual bean definition.

To override the default behavior for either of the helper classes, either extend the default implementations (DefaultBeanDefinitionDataCollector, DefaultBeanDefinitionData), or implement the relevant interface directly. To ensure your implementation is used instead of the default, add the @Replaces annotation to your class with the value being the default implementation.

15.2.2 The Info Endpoint

The info endpoint returns static information from the state of the application. The info exposed can be provided by any number of "info sources".

To execute the info endpoint, send a GET request to /info.

Configuration

To configure the info endpoint, supply configuration through `endpoints.info`.

Info Endpoint Configuration Example

YAML

```
endpoints:
  info:
    enabled: Boolean
    sensitive: Boolean
```

Customization

The info endpoint consists of an info aggregator and any number of info sources. To add an info source, create a bean class that implements **InfoSource**. If your info source needs to retrieve data from Java properties files, extend the **PropertiesInfoSource** interface which provides a helper method for this purpose.

All info source beans are collected together with the info aggregator. To provide your own implementation of the info aggregator, create a class that implements **InfoAggregator** and register it as a bean. To ensure your implementation is used instead of the default, add the **@Replaces** annotation to your class with the value being the **default implementation**.

The default info aggregator returns a map containing the combined properties returned by all the info sources. This map is returned as JSON from the /info endpoint.

Provided Info Sources

Configuration Info Source

The **ConfigurationInfoSource** returns configuration properties under the `info` key. In addition to string, integer and boolean values, more complex properties can be exposed as maps in the JSON output (if the configuration format supports it).

Info Source Example (`application.groovy`)

GROOVY

```
info.demo.string = "demo string"
info.demo.number = 123
info.demo.map = [key: 'value', other_key: 123]
```

The above config results in the following JSON response from the info endpoint:

JSON

```
{
  "demo": {
    "string": "demo string",
    "number": 123,
    "map": {
      "key": "value",
      "other_key": 123
    }
  }
}
```

Configuration

The configuration info source can be disabled using the `endpoints.info.config.enabled` property.

Git Info Source

If a `git.properties` file is available on the classpath, the **GitInfoSource** exposes the values in that file under the `git` key. Generating of a `git.properties` file must be configured as part of your build. One easy option for Gradle users is the **Gradle Git Properties Plugin** (https://plugins.gradle.org/plugin/com.gorylenko.gradle-git-properties). Maven users can use the **Maven Git Commit ID Plugin** (https://github.com/git-commit-id/maven-git-commit-id-plugin).

Configuration

To specify an alternate path or name of the properties file, supply a custom value in the `endpoints.info.git.location` property.

The git info source can be disabled using the `endpoints.info.git.enabled` property.

Build Info Source

If a `META-INF/build-info.properties` file is available on the classpath, the **BuildInfoSource** exposes the values in that file under the `build` key. Generating a `build-info.properties` file must be configured as part of your build. One easy option for Gradle users is the **Gradle Build Info Plugin** (https://plugins.gradle.org/plugin/com.pasam.gradle.buildinfo). An option for Maven users is the **Spring Boot Maven Plugin** (https://docs.spring.io/spring-boot/docs/current/maven-plugin/examples/build-info.html)

Configuration

To specify an alternate path/name of the properties file, supply a custom value in the `endpoints.info.build.location` property.

The build info source can be disabled using the `endpoints.info.build.enabled` property.

15.2.3 The Health Endpoint

The health endpoint returns information about the "health" of the application, which is determined by any number of "health indicators".

To execute the health endpoint, send a GET request to /health. Additionally the health endpoint exposes /health/liveness and /health/readiness health indicators.

Configuration

To configure the health endpoint, supply configuration through `endpoints.health`.

Health Endpoint Configuration Example

YAML

```yaml
endpoints:
  health:
    enabled: Boolean
    sensitive: Boolean
    details-visible: String  1
    status:
      http-mapping: Map<String, HttpStatus>
```

1 One of **DetailsVisibility**

The `details-visible` setting controls whether health detail will be exposed to users who are not authenticated.

For example, setting:

Using `details-visible`

YAML

```yaml
endpoints:
  health:
    details-visible: ANONYMOUS
```

exposes detailed information from the various health indicators about the health status of the application to anonymous unauthenticated users.

The `endpoints.health.status.http-mapping` setting controls which status codes to return for each health status. The defaults are described in the table below:

Status	HTTP Code
UP	**OK** (200)
UNKNOWN	**OK** (200)
DOWN	**SERVICE_UNAVAILABLE** (503)

You can provide custom mappings in `application.yml`:

Custom Health Status Codes

YAML

```yaml
endpoints:
  health:
    status:
      http-mapping:
        DOWN: 200
```

The above returns **OK** (200) even when the **HealthStatus** is **DOWN**.

Customization

The health endpoint consists of a health aggregator and any number of health indicators. To add a health indicator, create a bean class that implements **HealthIndicator**. It is recommended to also use either **@Liveness** or **@Readiness** qualifier. If no qualifier is used, the health indicator will be part of /health and /health/readiness endpoints. A base class **AbstractHealthIndicator** is available to subclass to make the process easier.

All health indicator beans are collected together with the health aggregator. To provide your own implementation of the health aggregator, create a class that implements **HealthAggregator** and register it as a bean. To ensure your implementation is used instead of the default, add the **@Replaces** annotation to your class with the value being the default implementation **DefaultHealthAggregator**.

The default health aggregator returns an overall status calculated based on the health statuses of the indicators. A **health status** consists of several pieces of information.

Name	The name of the status
Description	The description of the status

Operational	Whether the functionality the indicator represents is functional
Severity	How severe the status is. A higher number is more severe

The "worst" status is returned as the overall status. A non-operational status is selected over an operational status. A higher severity is selected over a lower severity.

Provided Indicators

All Micronaut provided health indicators are exposed on /health and /health/readiness endpoints.

Disk Space

A health indicator is provided that determines the health of the application based on the amount of free disk space. Configuration for the disk space health indicator can be provided under the `endpoints.health.disk-space` key.

Disk Space Indicator Configuration Example

YAML

```
endpoints:
  health:
    disk-space:
      enabled: Boolean
      path: String #The file path used to determine the disk space
      threshold: String | Long #The minimum amount of free space
```

The threshold can be provided as a string like "10MB" or "200KB", or the number of bytes.

JDBC

The JDBC health indicator determines the health of your application based on the ability to successfully create connections to datasources in the application context. The only configuration option supported is to enable or disable the indicator by the `endpoints.health.jdbc.enabled` key.

Discovery Client

If your application uses service discovery, a health indicator is included to monitor the health of the discovery client. The data returned can include a list of the services available.

15.2.4 The Metrics Endpoint

Micronaut can expose application metrics via integration with **Micrometer** (http://micrometer.io).

Using the CLI

If you create your project using the Micronaut CLI, supply one of the micrometer features to enable metrics and preconfigure the selected registry in your project. For example:

```
$ mn create-app my-app --features micrometer-atlas
```

The metrics endpoint returns information about the "metrics" of the application. To execute the metrics endpoint, send a GET request to `/metrics`. This returns a list of available metric names.

You can get specific metrics by using `/metrics/[name]` such as `/metrics/jvm.memory.used`.

See the documentation for **Micronaut Micrometer** (https://micronaut-projects.github.io/micronaut-micrometer/latest/guide/) for a list of registries and information on how to configure, expose and customize metrics output.

15.2.5 The Refresh Endpoint

The refresh endpoint refreshes the application state, causing all **Refreshable** beans in the context to be destroyed and reinstantiated upon further requests. This is accomplished by publishing a **RefreshEvent** in the Application Context.

To execute the refresh endpoint, send a POST request to /refresh.

BASH

```
$ curl -X POST http://localhost:8080/refresh
```

When executed without a body, the endpoint first refreshes the **Environment** and performs a diff to detect any changes, and then only performs the refresh if changes are detected. To skip this check and refresh all `@Refreshable` beans regardless of environment changes (e.g., to force refresh of cached responses from third-party services), add a `force` parameter in the POST request body.

BASH

```
$ curl -X POST http://localhost:8080/refresh -H 'Content-Type: application/json' -d '{"force": true}'
```

Configuration

To configure the refresh endpoint, supply configuration through `endpoints.refresh`.

Beans Endpoint Configuration Example

YAML

```yaml
endpoints:
  refresh:
    enabled: Boolean
    sensitive: Boolean
```

15.2.6 The Routes Endpoint

The routes endpoint returns information about URIs available to be called for your application. By default the data returned includes the URI, allowed method, content types produced, and information about the method that would be executed.

To execute the routes endpoint, send a GET request to /routes.

Configuration

To configure the routes endpoint, supply configuration through `endpoints.routes`.

Routes Endpoint Configuration Example

YAML

```yaml
endpoints:
  routes:
    enabled: Boolean
    sensitive: Boolean
```

Customization

The routes endpoint is composed of a route data collector and a route data implementation. The route data collector (**RouteDataCollector**) is responsible for returning a publisher that returns the data used in the response. The route data (**RouteData**) is responsible for returning data about an individual route.

To override the default behavior for either of the helper classes, either extend the default implementations (**DefaultRouteDataCollector**, **DefaultRouteData**), or implement the relevant interface directly. To ensure your implementation is used instead of the default, add the **@Replaces** annotation to your class with the value being the default implementation.

15.2.7 The Loggers Endpoint

The loggers endpoint returns information about the available loggers in the application and permits configuring their log level.

The loggers endpoint is disabled by default and must be explicitly enabled with the setting `endpoints.loggers.enabled=true`.

To get a collection of all loggers by name with their configured and effective log levels, send a GET request to /loggers. This also provides a list of the available log levels.

BASH

```
$ curl http://localhost:8080/loggers

{
    "levels": [
        "ALL", "TRACE", "DEBUG", "INFO", "WARN", "ERROR", "OFF", "NOT_SPECIFIED"
    ],
    "loggers": {
        "ROOT": {
            "configuredLevel": "INFO",
            "effectiveLevel": "INFO"
        },
        "io": {
            "configuredLevel": "NOT_SPECIFIED",
            "effectiveLevel": "INFO"
        },
        "io.micronaut": {
            "configuredLevel": "NOT_SPECIFIED",
            "effectiveLevel": "INFO"
        },
        // etc...
    }
}
```

To get the log levels of a particular logger, include the logger name in your GET request. For example, to access the log levels of the logger 'io.micronaut.http':

```
$ curl http://localhost:8080/loggers/io.micronaut.http

{
    "configuredLevel": "NOT_SPECIFIED",
    "effectiveLevel": "INFO"
}
```

BASH

If the named logger does not exist, it is created with an unspecified (i.e. `NOT_SPECIFIED`) configured log level (its effective log level is usually that of the root logger).

To update the log level of a single logger, send a POST request to the named logger URL and include a body providing the log level to configure.

```
$ curl -i -X POST \
       -H "Content-Type: application/json" \
       -d '{ "configuredLevel": "ERROR" }' \
       http://localhost:8080/loggers/ROOT

HTTP/1.1 200 OK

$ curl http://localhost:8080/loggers/ROOT

{
    "configuredLevel": "ERROR",
    "effectiveLevel": "ERROR"
}
```

BASH

Configuration

To configure the loggers endpoint, supply configuration through `endpoints.loggers`.

Loggers Endpoint Configuration Example

```
endpoints:
  loggers:
    enabled: Boolean
    sensitive: Boolean
```

YAML

By default, the endpoint doesn't allow changing the log level by unauthorized users (even if `sensitive` is set to `false`). To allow this you must set `endpoints.loggers.write-sensitive` to `false`.

Customization

The loggers endpoint is composed of two customizable parts: a **LoggersManager** and a **LoggingSystem**. See the **logging section of the documentation** for information on customizing the logging system.

The **LoggersManager** is responsible for retrieving and setting log levels. If the default implementation is not sufficient for your use case, simply provide your own implementation and replace the **default implementation** with the **@Replaces** annotation.

15.2.8 The Caches Endpoint

The caches endpoint documentation is available at the **micronaut-cache project** **(https://micronaut-projects.github.io/micronaut-cache/latest/guide/index.html#endpoint)**.

15.2.9 The Server Stop Endpoint

The stop endpoint shuts down the application server.

To execute the stop endpoint, send a POST request to /stop.

Configuration

To configure the stop endpoint, supply configuration through `endpoints.stop`.

Stop Endpoint Configuration Example

```
endpoints:
  stop:
    enabled: Boolean
    sensitive: Boolean
```

YAML

By default, the stop endpoint is disabled and must be explicitly enabled to be used.

15.2.10 The Environment Endpoint

The environment endpoint returns information about the **Environment** and its **PropertySource**s.

Properties that may contain sensitive data are masked.

Configuration

To configure the environment endpoint, supply configuration through `endpoints.env`.

Environment Endpoint Configuration Example

YAML

```yaml
endpoints:
  env:
    enabled: Boolean    # default: true
    sensitive: Boolean  # default: true
```

Getting information about the environment

To execute the endpoint, send a `GET` request to `/env`.

Getting information about a particular `PropertySource`

To execute the endpoint, send a `GET` request to `/env/{propertySourceName}`.

15.2.11 The ThreadDump Endpoint

The threaddump endpoint returns information about the threads running in your application.

To execute the threaddump endpoint, send a GET request to /threaddump.

Configuration

To configure the threaddump endpoint, supply configuration through `endpoints.threaddump`.

Threaddump Endpoint Configuration Example

YAML

```yaml
endpoints:
  threaddump:
    enabled: Boolean
    sensitive: Boolean
```

Customization

The thread dump endpoint delegates to a **ThreadInfoMapper**) that is responsible for transforming the `java.lang.management.ThreadInfo` objects into any other to be sent for serialization.

16 Security

Micronaut has a full-featured security solution for all of the common security patterns.

See the documentation for **Micronaut Security** (https://micronaut-projects.github.io/micronaut-security/{micronautSecurityVersion}/guide) for more information on how to secure your applications.

17 Multi-Tenancy

See the **Micronaut Multitenancy documentation** (https://micronaut-projects.github.io/micronaut-multitenancy/latest/guide) to learn about Micronaut's support for common tasks such as tenant resolution for multi-tenancy-aware Micronaut applications.

18 Micronaut CLI

The Micronaut CLI is the recommended way to create new Micronaut projects. The CLI includes commands for generating specific categories of projects, allowing you to choose between build tools, test frameworks, and even pick the language to use in your application. The CLI also provides commands for generating artifacts such as controllers, client interfaces, and serverless functions.

We have a website that can be used to generate projects instead of the CLI. Check out **Micronaut Launch** (https://micronaut.io/launch/) to get started!

When **Micronaut is installed on your computer**, you can call the CLI with the `mn` command.

```
$ mn create-app my-app
```
BASH

A Micronaut CLI project can be identified by the `micronaut-cli.yml` file, which is included at the project root if it was generated via the CLI. This file will include the project's profile, default package, and other variables. The project's default package is evaluated based on the project name.

```
$ mn create-app my-demo-app
```
BASH

results in the default package being `my.demo.app`.

You can supply your own default package when creating the application by prefixing the application name with the package:

```
$ mn create-app example.my-demo-app
```
BASH

results in the default package being `example`.

Interactive Mode

If you run `mn` without any arguments, the Micronaut CLI launches in interactive mode. This is a shell-like mode which lets you run multiple CLI commands without re-initializing the CLI runtime, and is especially suitable when you use code-generation commands (such as `create-controller`), create multiple projects, or are just exploring CLI features. Tab-completion is enabled, enabling you to hit the `TAB` key to see possible options for a given command or flag.

```
$ mn
| Starting interactive mode...
| Enter a command name to run. Use TAB for completion:
mn>
```
BASH

Help and Info

General usage information can be viewed using the `help` flag on a command.

```
mn> create-app -h
Usage: mn create-app [-hivVx] [--list-features] [-b=BUILD-TOOL] [--jdk=<javaVersion>] [-l=LANG]
                     [-t=TEST] [-f=FEATURE[,FEATURE...]]... [NAME]
Creates an application
      [NAME]                The name of the application to create.
  -b, --build=BUILD-TOOL    Which build tool to configure. Possible values: gradle, gradle_kotlin,
                              maven.
  -f, --features=FEATURE[,FEATURE...]
  -h, --help                Show this help message and exit.
  -i, --inplace             Create a service using the current directory
      --jdk, --java-version=<javaVersion>
                            The JDK version the project should target
  -l, --lang=LANG           Which language to use. Possible values: java, groovy, kotlin.
      --list-features       Output the available features and their descriptions
  -t, --test=TEST           Which test framework to use. Possible values: junit, spock, kotest.
```
BASH

A list of available features can be viewed using the `--list-features` flag on any of the create commands.

```
mn> create-app --list-features
Available Features
(+) denotes the feature is included by default
  Name                           Description
  -----------------------------  ---------------
  Cache
  cache-caffeine                 Adds support for cache using Caffeine (https://github.com/ben-manes/caffeine)
  cache-ehcache                  Adds support for cache using EHCache (https://www.ehcache.org/)
  cache-hazelcast                Adds support for cache using Hazelcast (https://hazelcast.org/)
  cache-infinispan               Adds support for cache using Infinispan (https://infinispan.org/)
```
BASH

18.1 Creating a Project

Creating a project is the primary usage of the CLI. The primary command for creating a new project is `create-app`, which creates a standard server application that communicates over HTTP. For other types of application, see the documentation below.

Table 1. Micronaut CLI Project Creation Commands

Command	Description	Options	Example
`create-app`	Creates a basic Micronaut application.	• `-l, --lang` • `-t, --test` • `-b, --build` • `-f, --features` • `-i, --inplace`	BASH `mn create-app my-project --features mongo-reactive,security-jwt --build maven`
`create-cli-app`	Creates a command-line Micronaut application.	• `-l, --lang` • `-t, --test` • `-b, --build` • `-f, --features` • `-i, --inplace`	BASH `mn create-cli-app my-project --features http-client,jdbc-hikari --build maven --lang kotlin --test kotest`
`create-function-app`	Creates a Micronaut serverless function, using AWS by default.	• `-l, --lang` • `-t, --test` • `-b, --build` • `-f, --features` • `-i, --inplace`	BASH `mn create-function-app my-lambda-function --lang groovy --test spock`
`create-messaging-app`	Creates a Micronaut application that only communicates via a messaging protocol. Uses Kafka by default but can be switched to RabbitMQ with `--features rabbitmq`.	• `-l, --lang` • `-t, --test` • `-b, --build` • `-f, --features` • `-i, --inplace`	BASH `mn create-messaging-app my-broker --lang groovy --test spock`
`create-grpc-app`	Creates a Micronaut application that uses gRPC.	• `-l, --lang` • `-t, --test` • `-b, --build` • `-f, --features` • `-i, --inplace`	BASH `mn create-grpc-app my-grpc-app --lang groovy --test spock`

Create Command Flags

The create-* commands generate a basic Micronaut project, with optional flags to specify features, language, test framework, and build tool. All projects except functions include a default `Application` class for starting the application.

Table 2. Flags

Flag	Description	Example
`-l, --lang`	Language to use for the project (one of `java`, `groovy`, `kotlin` - default is `java`)	`--lang groovy`
`-t, --test`	Test framework to use for the project (one of `junit`, `spock` - default is `junit`)	`--test spock`
`-b, --build`	Build tool (one of `gradle`, `gradle_kotlin`, `maven` - default is `gradle` for the languages `java` and `groovy`; default is `gradle_kotlin` for language `kotlin`)	`--build maven`

Flag	Description	Example
`-f, --features`	Features to use for the project, comma-separated	`--features security-jwt,mongo-gorm` BASH or `-f security-jwt -f mongo-gorm` BASH
`-i, --inplace`	If present, generates the project in the current directory (project name is optional if this flag is set)	`--inplace`

Once created, the application can be started using the `Application` class, or the appropriate build tool task.

Starting a Gradle project

```
$ ./gradlew run
```
BASH

Starting a Maven project

```
$ ./mvnw mn:run
```
BASH

Language/Test Features

By default, the create commands generate a Java application, with JUnit configured as the test framework. All the options chosen and features applied are stored as properties in the `micronaut-cli.yml` file, as shown below:

micronaut-cli.yml

```
applicationType: default
defaultPackage: com.example
testFramework: junit
sourceLanguage: java
buildTool: gradle
features: [annotation-api, app-name, application, gradle, http-client, java, junit, logback, netty-server, shade, yaml]
```
YAML

Some commands rely on the data in this file to determine if they should be executable. For example, the `create-kafka-listener` command requires `kafka` to be one of the features in the list.

The values in `micronaut-cli.yml` are used by the CLI for code generation. After a project is generated, you can edit these values to change the project defaults, however you must supply the required dependencies and/or configuration to use your chosen language/framework. For example, you could change the `testFramework` property to `spock` to cause the CLI to generate Spock tests when running commands (such as `create-controller`), but you need to add the Spock dependency to your build.

Groovy

To create an app with Groovy support (which uses Spock by default), supply the appropriate language via the `lang` flag:

```
$ mn create-app my-groovy-app --lang groovy
```
BASH

This includes the Groovy and Spock dependencies in your project, and writes the appropriates values in `micronaut-cli.yml`.

Kotlin

To create an app with Kotlin support (which uses Kotest by default), supply the appropriate language via the `lang` flag:

```
$ mn create-app my-kotlin-app --lang kotlin
```
BASH

This includes the Kotlin and Kotest dependencies in your project, and writes the appropriates values in `micronaut-cli.yml`.

Build Tool

By default, `create-app` creates a Gradle project, with a `build.gradle` file in the project root directory. To create an app using the Maven build tool, supply the appropriate option via the `build` flag:

```
$ mn create-app my-maven-app --build maven
```

BASH

Create-Cli-App

The `create-cli-app` command generates a **Micronaut command line application** project, with optional flags to specify language, test framework, features, profile, and build tool. By default, the project includes the `picocli` feature to support command line option parsing. The project will include a `*Command` class (based on the project name, e.g. `hello-world` generates `HelloWorldCommand`), and an associated test which instantiates the command and verifies that it can parse command line options.

Once created, the application can be started using the `*Command` class, or the appropriate build tool task.

Starting a Gradle project

```
$ ./gradlew run
```

BASH

Starting a Maven project

```
$ ./mvnw mn:run
```

BASH

Create Function App

The `create-function-app` command generates a **Micronaut function** project, optimized for serverless environments, with optional flags to specify language, test framework, features, and build tool. The project will include a `*Function` class (based on the project name, e.g. `hello-world` generates `HelloWorldFunction`), and an associated test which instantiates the function and verifies that it can receive requests.

Currently, AWS Lambda, Micronaut Azure, and Google Cloud are the supported cloud providers for Micronaut functions. To use other providers, add one in the features: `--features azure-function` or `--features google-cloud-function`.

Contribute

The CLI source code is at **https://github.com/micronaut-projects/micronaut-starter**. Information about how to contribute and other resources are there.

18.1.1 Comparing Versions

The easiest way to see version dependency updates and other changes for a new version of Micronaut is to produce one clean application using the older version and another using the newer version of the `mn` CLI, and then comparing those directories.

18.2 Features

Features consist of additional dependencies and configuration to enable specific functionality in your application. Micronaut profiles define a large number of features, including features for many of the configurations provided by Micronaut, such as the **Data Access Configurations**

```
$ mn create-app my-demo-app --features mongo-reactive
```

BASH

This adds the necessary dependencies and configuration for the **MongoDB Reactive Driver** **(http://mongodb.github.io/mongo-java-driver-reactivestreams)** in your application. You can view the available features using the `--list-features` flag for whichever create command you use.

```
$ mn create-app --list-features # Output will be supported features for the create-app command
$ mn create-function-app --list-features # Output will be supported features for the create-function-app command, diffe
```

BASH

18.3 Commands

You can view a full list of available commands using the help flag, for example:

```
$ mn -h
Usage: mn [-hvVx] [COMMAND]
Micronaut CLI command line interface for generating projects and services.
Application generation commands are: 1

*  create-app NAME
*  create-cli-app NAME
*  create-function-app NAME
*  create-grpc-app NAME
*  create-messaging-app NAME

Options:
  -h, --help         Show this help message and exit.
  -v, --verbose      Create verbose output.
  -V, --version      Print version information and exit.
  -x, --stacktrace   Show full stack trace when exceptions occur.

Commands: 2
  create-app               Creates an application
  create-cli-app           Creates a CLI application
  create-function-app      Creates a Cloud Function
  create-grpc-app          Creates a gRPC application
  create-messaging-app     Creates a messaging application
  create-job               Creates a job with scheduled method
  create-bean              Creates a singleton bean
  create-websocket-client  Creates a Websocket client
  create-client            Creates a client interface
  create-controller        Creates a controller and associated test
  feature-diff             Produces the diff of an original project with an original project with
                             additional features.
  create-websocket-server  Creates a Websocket server
  create-test              Creates a simple test for the project's testing framework
```

1 Here you can see the project generation commands lists

2 All commands available in the current directory are listed here

3 **Note:** the things listed after the project creation commands (always available) depend on the current directory context

All the code-generation commands honor the values written in `micronaut-cli.yml`. For example, assume the following `micronaut-cli.yml` file.

micronaut-cli.yml

```
defaultPackage: example
---
testFramework: spock
sourceLanguage: java
```

With the above settings, the `create-bean` command (by default) generates a Java class with an associated Spock test, in the `example` package. Commands accept arguments and these defaults can be overridden on a per-command basis.

Base Commands

These commands are always available within the context of a micronaut project.

Create-Bean

Table 1. Create-Bean Flags

Flag	Description	Example
`-l, --lang`	The language used for the bean class	`--lang groovy`
`-f, --force`	Whether to overwrite existing files	`--force`

The `create-bean` command generates a simple **Singleton** (https://docs.oracle.com/javaee/6/api/javax/inject/Singleton.html) class. It does not create an associated test.

```
$ mn create-bean EmailService
| Rendered template Bean.java to destination src/main/java/example/EmailService.java
```

Create-Job

Table 2. Create-Job Flags

Flag	Description	Example

Flag	Description	Example
`-l, --lang`	The language used for the job class	`--lang groovy`
`-f, --force`	Whether to overwrite existing files	`--force`

The `create-job` command generates a simple **Scheduled** class. It follows a `*Job` convention for generating the class name. It does not create an associated test.

```
$ mn create-job UpdateFeeds --lang groovy
| Rendered template Job.groovy to destination src/main/groovy/example/UpdateFeedsJob.groovy
```

BASH

HTTP-Related Commands

Create-Controller

Table 3. Create-Controller Flags

Flag	Description	Example
`-l, --lang`	The language used for the controller	`--lang groovy`
`-f, --force`	Whether to overwrite existing files	`--force`

The `create-controller` command generates a **Controller** class. It follows a `*Controller` convention for generating the class name. It creates an associated test that runs the application and instantiates an HTTP client, which can make requests against the controller.

```
$ mn create-controller Book
| Rendered template Controller.java to destination src/main/java/example/BookController.java
| Rendered template ControllerTest.java to destination src/test/java/example/BookControllerTest.java
```

BASH

Create-Client

Table 4. Create-Client Flags

Flag	Description	Example
`-l, --lang`	The language used for the client	`--lang groovy`
`-f, --force`	Whether to overwrite existing files	`--force`

The `create-client` command generates a simple **Client** interface. It follows a `*Client` convention for generating the class name. It does not create an associated test.

```
$ mn create-client Book
| Rendered template Client.java to destination src/main/java/example/BookClient.java
```

BASH

Create-Websocket-Server

Table 5. Create-Websocket-Server Flags

Flag	Description	Example
`-l, --lang`	The language used for the server	`--lang groovy`
`-f, --force`	Whether to overwrite existing files	`--force`

The `create-websocket-server` command generates a simple **ServerWebSocket** class. It follows a `*Server` convention for generating the class name. It does not create an associated test.

```
$ mn create-websocket-server MyChat
| Rendered template WebsocketServer.java to destination src/main/java/example/MyChatServer.java
```

BASH

Create-Websocket-Client

Table 6. Create-Websocket-Client Flags

Flag	Description	Example
`-l, --lang`	The language used for the client	`--lang groovy`
`-f, --force`	Whether to overwrite existing files	`--force`

The `create-websocket-client` command generates a simple **WebSocketClient** abstract class. It follows a `*Client` convention for generating the class name. It does not create an associated test.

```
$ mn create-websocket-client MyChat
| Rendered template WebsocketClient.java to destination src/main/java/example/MyChatClient.java
```

CLI Project Commands

Create-Command

Table 7. Create-Command Flags

Flag	Description	Example
`-l, --lang`	The language used for the command	`--lang groovy`
`-f, --force`	Whether to overwrite existing files	`--force`

The `create-command` command generates a standalone application that can be executed as a **picocli** (http://picocli.info) **Command** (http://picocli.info/apidocs/picocli/CommandLine.Command.html). It follows a `*Command` convention for generating the class name. It creates an associated test that runs the application and verifies that a command line option was set.

```
$ mn create-command print
| Rendered template Command.java to destination src/main/java/example/PrintCommand.java
| Rendered template CommandTest.java to destination src/test/java/example/PrintCommandTest.java
```

This list is just a small subset of the code generation commands in the Micronaut CLI. To see all context-sensitive commands the CLI has available (and under what circumstances they apply), check out the **micronaut-starter** (https://github.com/micronaut-projects/micronaut-starter) project and find the classes that extend `CodeGenCommand`. The `applies` method dictates whether a command is available or not.

18.4 Reloading

Reloading (or "hot-loading") refers to the framework reinitializing classes (and parts of the application) when changes to the source files are detected.

Since Micronaut prioritizes startup time and most Micronaut apps can start up within seconds, a productive workflow can often be had by restarting the application as changes are made; for example, by running a test class within an IDE.

However, to have your changes automatically reloaded, Micronaut supports automatic restart and the use of third-party reloading agents.

18.4.1 Automatic Restart

There are various ways to achieve reloading of classes on the JVM, and all have their advantages and disadvantages. The following are possible ways to achieve reloading without restarting the JVM:

- **JVM Agents** - A JVM agent like JRebel can be used, however these can produce unusual errors, may not support all JDK versions, and can result in cached or stale classes.
- **ClassLoader Reloading** - ClassLoader-based reloading is a popular solution used by most JVM frameworks; however it once again can lead to cached or stale classes, memory leaks, and weird errors if the incorrect classloader is used.
- **Debugger HotSwap** - The Java debugger supports hotswapping of changes at runtime, but only supports a few use cases.

Given the problems with existing solutions and a lack of a way built into the JVM to reload changes, the safest and best solution to reloading, and the one recommended by the Micronaut team, is to use automatic application restart via a third-party tool.

Micronaut's startup time is fast and automatic restart leads to a clean slate without potential hard to debug problems or memory leaks cropping up.

Maven Restart

To have automatic application restarts with Maven, use the Micronaut Maven plugin (included by default when creating new Maven projects) and run the following command:

Using the Micronaut Maven Plugin

```
$ ./mvnw mn:run
```

Every time you change a class, the plugin automatically restarts the server.

Gradle Restart

Gradle automatic restart can be activated when using the Micronaut Gradle plugin by activating Gradle's support for continuous builds via the `-t` flag:

Using Gradle for Automatic Restart

```
./gradlew run -t
```

Every time you make a change to class or resources, Gradle recompiles and restarts the application.

18.4.2 JRebel

JRebel (https://www.jrebel.com/products/jrebel) is a proprietary reloading solution that involves an agent library, as well as sophisticated IDE support. The JRebel documentation includes detailed steps for IDE integration and usage. In this section, we show how to install and configure the agent for Maven and Gradle projects.

Using the CLI

If you create your project using the Micronaut CLI, supply the `jrebel` feature to preconfigure JRebel reloading in your project. Note that you need to install JRebel and supply the correct path to the agent in the `gradle.properties` file (for Gradle) or `pom.xml` (for Maven). The necessary steps are described below.

```
$ mn create-app my-app --features jrebel
```

Install/configure JRebel Agent

The simplest way to install JRebel is to download the "standalone" installation package from the JRebel download page (https://www.jrebel.com/products/jrebel/download). Unzip the downloaded file to a convenient location, for example `~/bin/jrebel`

The installation directory contains a `lib` directory with the agent files. For the appropriate agent based on your operating system, see the table below:

Table 1. JRebel Agent

OS	Agent
Windows 64-bit JDK	`[jrebel directory]\lib\jrebel64.dll`
Windows 32-bit JDK	`[jrebel directory]\lib\jrebel32.dll`
Mac OS X 64-bit JDK	`[jrebel directory]/lib/libjrebel64.dylib`
Mac OS X 32-bit JDK	`[jrebel directory]/lib/libjrebel32.dylib`
Linux 64-bit JDK	`[jrebel directory]/lib/libjrebel64.so`
Linux 32-bit JDK	`[jrebel directory]/lib/libjrebel32.so`

Note the path to the appropriate agent, and add the value to your project build.

Gradle

Add the path to `gradle.properties` (create the file if necessary), as the `rebelAgent` property.

gradle.properties

```
#Assuming installation path of ~/bin/jrebel/
rebelAgent= -agentpath:~/bin/jrebel/lib/libjrebel64.dylib
```

PROPERTIES

Add the appropriate JVM arg to `build.gradle` (not necessary if using the CLI feature)

```
run.dependsOn(generateRebel)
if (project.hasProperty('rebelAgent')) {
    run.jvmArgs += rebelAgent
}
```

GROOVY

You can start the application with `./gradlew run`, and it will include the agent. See the section on **Gradle Reloading** or **IDE Reloading** to set up the recompilation.

Maven

Configure the Micronaut Maven Plugin accordingly:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <!-- ... -->
  <build>
    <plugins>
      <!-- ... -->
      <plugin>
        <groupId>io.micronaut.build</groupId>
        <artifactId>micronaut-maven-plugin</artifactId>
          <configuration>
            <jvmArguments>
              <jvmArgument>-agentpath:~/bin/jrebel/lib/jrebel6/lib/libjrebel64.dylib</jvmArgument>
            </jvmArguments>
          </configuration>
      </plugin>
      <plugin>
        <groupId>org.zeroturnaround</groupId>
        <artifactId>jrebel-maven-plugin</artifactId>
        <version>1.1.10</version>
        <executions>
          <execution>
            <id>generate-rebel-xml</id>
            <phase>process-resources</phase>
            <goals>
              <goal>generate</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <!-- ... -->
    </plugins>
  </build>
</project>
```

18.4.3 Recompiling with Gradle

Gradle supports **continuous builds** (https://docs.gradle.org/current/userguide/command_line_interface.html#sec:continuous_build), letting you run a task that will be rerun whenever source files change. To use this with a reloading agent (configured as described above), run the application normally (with the agent), and then run a recompilation task in a separate terminal with continuous mode enabled.

Run the app

```bash
$ ./gradlew run
```

Run the recompilation

```bash
$ ./gradlew -t classes
```

The `classes` task will be rerun every time a source file is modified, allowing the reloading agent to pick up the change.

18.4.4 Recompiling with an IDE

If you use a build tool such as Maven which does not support automatic recompilation on file changes, you may use your IDE to recompile classes in combination with a reloading agent (as configured in the above sections).

IntelliJ

IntelliJ unfortunately does not have an automatic rebuild option that works for a running application. However, you can trigger a "rebuild" of the project with `CMD-F9` (Mac) or `CTRL-F9` (Windows/Linux).

Eclipse

Under the `Project` menu, check the `Build Automatically` option. This will trigger a recompilation of the project whenever file changes are saved to disk.

18.5 Proxy Configuration

To configure the CLI to use an HTTP proxy there are two steps. Configuration options can be passed to the cli through the `MN_OPTS` environment variable.

For example on *nix systems:

```bash
export MN_OPTS="-Dhttps.proxyHost=127.0.0.1 -Dhttps.proxyPort=3128 -Dhttp.proxyUser=test -Dhttp.proxyPassword=test"
```

The profile dependencies are resolved over HTTPS so the proxy port and host are configured with `https.`, however the user and password are specified with `http.`.

For Windows systems the environment variable can be configured under `My Computer/Advanced/Environment Variables`.

19 Internationalization

19.1 Resource Bundles

A resource bundle is a Java .properties file that contains locale-specific data.

Given this Resource Bundle:

src/main/resources/io/micronaut/docs/i18n/messages_en.properties

```
hello=Hello
```
PROPERTIES

src/main/resources/io/micronaut/docs/i18n/messages_es.properties

```
hello=Hola
```
PROPERTIES

You can use **ResourceBundleMessageSource**, an implementation of **MessageSource** which eases accessing **Resource Bundles** (https://docs.oracle.com/javase/8/docs/api//java/util/ResourceBundle.html) and provides cache functionality, to access the previous messages:

ResourceBundleMessageSource Example

Java | Groovy | Kotlin

```java
ResourceBundleMessageSource ms = new ResourceBundleMessageSource("io.micronaut.docs.i18n.messages");
assertEquals("Hola", ms.getMessage("hello", MessageContext.of(new Locale("es"))).get());
assertEquals("Hello", ms.getMessage("hello", MessageContext.of(Locale.ENGLISH)).get());
```
JAVA

Copy to Clipboard

Do not instantiate a new `ResourceBundleMessageSource` each time you retrieve a message. Instantiate it once, for example in a `javax.inject.Singleton`, and reuse it.

20 Appendices

20.1 Frequently Asked Questions (FAQ)

The following section covers frequently asked questions that you may find yourself asking while considering to use or using Micronaut.

Does Micronaut modify my bytecode?

No. Your classes are your classes. Micronaut does not transform classes or modify the bytecode generated from the code you write. Micronaut produces additional classes at compile time in the same package as your original unmodified classes.

Why Doesn't Micronaut use Spring?

When asking why Micronaut doesn't use Spring, it is typically in reference to the Spring Dependency Injection container.

The Spring ecosystem is very broad and there are many Spring libraries you can use directly in Micronaut without requiring the Spring container.

The reason Micronaut features its own native **JSR-330** (https://www.jcp.org/en/jsr/detail?id=330) compliant dependency injection is that the cost of these features in Spring (and any reflection-based DI/AOP container) is too great in terms of memory consumption and the impact on startup time. To support dependency injection at runtime, Spring:

- **Reads the bytecode** (https://github.com/spring-projects/spring-framework/tree/master/spring-core/src/main/java/org/springframework/core/type/classreading) of every bean it finds at runtime.
- **Synthesizes new annotations** (https://github.com/spring-projects/spring-framework/blob/a691065d05741a4f1ca17925c8a5deec0f378c8b/spring-core/src/main/java/org/springframework/core/annotation/AnnotationUtils.java#L1465) for each annotation on each bean method, constructor, field etc. to support Annotation metadata.
- **Builds Reflective Metadata** (https://github.com/spring-projects/spring-framework/blob/master/spring-beans/src/main/java/org/springframework/beans/CachedIntrospectionResults.java) for each bean for every method, constructor, field, etc.

The result is a progressive degradation of startup time and memory consumption as your application incorporates more features.

For Microservices and Serverless functions where it is critical that startup time and memory consumption remain low, the above behaviour is an undesirable reality of using the Spring container, hence the designers of Micronaut chose not to use Spring.

Does Micronaut support Scala?

Micronaut supports any JVM language that supports the Annotation Processor API. Scala currently does not support this API. However, Groovy also doesn't support this API and special support has been built that processes the Groovy AST. It may be technically possible to support Scala in the future if a module similar to `inject-groovy` is built, but as of this writing Scala is not supported.

Can Micronaut be used for purposes other than Microservices?

Yes. Micronaut is very modular and you can choose to use just the Dependency Injection and AOP implementation by including the `micronaut-inject-java` (or `micronaut-inject-groovy` for Groovy) dependency in your application.

In fact Micronaut's support for **Serverless Computing** uses this exact approach.

What are the advantages of Micronaut's Dependency Injection and AOP implementation?

Micronaut processes your classes and produces all metadata at compile time. This eliminates the need for reflection, cached reflective metadata, and the requirement to analyze your classes at runtime, all of which lead to slower startup performance and greater memory consumption.

In addition, Micronaut builds reflection-free AOP proxies at compile time, which improves performance, reduces stack trace sizes, and reduces memory consumption.

Why does Micronaut have its own Consul and Eureka client implementations?

The majority of Consul and Eureka clients that exist are blocking and include many external dependencies that inflate your JAR files.

Micronaut's **DiscoveryClient** uses Micronaut's native HTTP client, greatly reducing the need for external dependencies and providing a reactive API onto both discovery servers.

Why am I encountering a NoSuchMethodError occurs loading my beans (Groovy)?

Groovy by default imports classes in the `groovy.lang` package, including one named `@Singleton`, an AST transformation class that makes your class a singleton by adding a private constructor and static retrieval method. This annotation is easily confused with the `javax.inject.Singleton` annotation used to define singleton beans in Micronaut. Make sure you use the correct annotation in your Groovy classes.

Why is it taking much longer than it should to start the application

Micronaut's startup time is typically very fast. At the application level however, it is possible to affect startup time. If you are seeing slow startup, review any application startup listeners or `@Context` scope beans that are slowing startup.

Some network issues can also cause slow startup. On the Mac for example, misconfiguration of your `/etc/hosts` file can cause issues. See the following **stackoverflow answer (https://stackoverflow.com/a/39698914/1264846)**.

20.2 Using Snapshots

Micronaut milestone and stable releases are distributed to Maven Central.

The following snippet shows how to use Micronaut `SNAPSHOT` versions with Gradle. The latest snapshot will always be the next patch version plus 1 with `-SNAPSHOT` appended. For example if the latest release is "1.0.1", the current snapshot would be "1.0.2-SNAPSHOT".

GROOVY

```
ext {
    micronautVersion = '2.4.0-SNAPSHOT'
}
repositories {
    mavenCentral() 1
    maven { url "https://s01.oss.sonatype.org/content/repositories/snapshots/" } 2
}
```

1 Micronaut releases are available on Maven Central

2 Micronaut snapshots are available on Sonatype OSS Snapshots

In the case of Maven, edit `pom.xml`:

```xml
<?xml version="1.0" encoding="UTF-8"?>
  ...

  <parent>
    <groupId>io.micronaut</groupId>
    <artifactId>micronaut-parent</artifactId>
    <version>2.4.0-SNAPSHOT</version>
  </parent>

  <properties>
    <micronaut.version>2.4.0-SNAPSHOT</micronaut.version> 1
    ...
  </properties>

  <repositories>
    <repository>
      <id>sonatype-snapshots</id>
      <url>https://s01.oss.sonatype.org/content/repositories/snapshots/</url> 2
    </repository>
  </repositories>

  ...
</project>
```

1 Set the snapshot version.

2 Micronaut snapshots are available on Sonatype OSS Snapshots

Previously snapshots were published to Bintray, however due to JFrog shutting the service down, snapshots are now being published to Sonatype.

20.3 Common Problems

The following section covers common problems developers encounter when using Micronaut.

Dependency injection is not working

The most common causes of Dependency Injection failing to work are not having the appropriate annotation processor configured, or an incorrectly configured IDE. See the section on **Language Support** for how to get setup in your language.

A NoSuchMethodError occurs loading beans (Groovy)

By default, Groovy imports classes in the `groovy.lang` package which includes a class called `Singleton`. This is an AST transformation annotation that makes your class a singleton by adding a private constructor and static retrieval method. This annotation is easily confused with the `javax.inject.Singleton` annotation used to define singleton beans in Micronaut. Make sure you use the correct annotation in your Groovy classes.

It is taking much longer to start my application than it should (*nix OS)

This is likely due to a bug related to `java.net.InetAddress.getLocalHost()` calls causing a long delay. The solution is to edit your `/etc/hosts` file to add an entry containing your host name. To find your host name, run `hostname` in a terminal. Then edit your `/etc/hosts` file to add or change entries like the example below, replacing `<hostname>` with your host name.

```
127.0.0.1       localhost <hostname>
::1             localhost <hostname>
```

To learn more about this issue, see this **stackoverflow answer (https://stackoverflow.com/a/39698914/1264846)**

20.4 Breaking Changes

This section documents breaking changes between Micronaut versions

3.1.0

Retrieving the port from the Netty embedded server is no longer supported if the server is configured to bind to a random port and the server has not been started.

3.0.0

Core Changes

Annotation Inheritance

Possibly the most important change in Micronaut 3.0 is how annotations are inherited from parent classes, methods and interfaces.

Micronaut 2.x did not respect the rules defined in the **AnnotatedElement** (https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/AnnotatedElement.html), and inherited all annotations from parent interfaces and types regardless of the presence of the **Inherited** (https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Inherited.html) annotation.

With Micronaut 3.x and above only annotations that are explicitly meta-annotated with **Inherited** (https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Inherited.html) are now inherited from parent classes and interfaces. This applies to types in the case where one extends another, and methods in the case where one overrides another.

Many of Micronaut's core annotations have been annotated with `@Inherited`, so no change will be required, but some annotations that are either outside Micronaut or defined by user code will need changes to code or the annotation.

In general, behaviour which you wish to override is not inherited by default in Micronaut 3.x and above including **Bean Scopes**, **Bean Qualifiers**, **Bean Conditions**, **Validation Rules** and so on.

The following table summarizes the core Micronaut annotations and which are inherited and which are not:

Table 1. Annotation Inheritance in Micronaut 3.x and above

Annotation	Inherited
@Adapter	✅
@Around	❌
@AroundConstruct	❌
@InterceptorBean	❌
@InterceptorBinding	❌
@Introduction	❌
@Blocking	✅
@Creator	❌
@EntryPoint	✅
@Experimental (source level)	❌
@Indexes & @Indexed	✅
@Internal	✅
@Introspected	✅
@NonBlocking	✅
@Nullable	❌
@NonNull	❌
@Order	❌
@ReflectiveAccess	❌
@TypeHint	❌
@SingleResult	✅
@Bindable	✅
@Format	✅
@MapFormat	✅
@ReadableBytes	✅
@Version	❌
@AliasFor	❌
@Any	❌
@Bean	❌
@BootstrapContextCompatible	✅
@ConfigurationBuilder	❌
@ConfigurationInject	❌
@ConfigurationProperties	❌

Annotation	Inherited
@ConfigurationReader	❌
@Context	❌
@DefaultImplementation	✅
@DefaultScope	❌
@EachBean	❌
@Executable	✅
@Factory	❌
@NonBinding	❌
@Parallel	❌
@Parameter	❌
@Primary	❌
@Property	❌
@PropertySource	❌
@Prototype	❌
@Replaces	❌
@Requirements	❌
@Requires	❌
@Secondary	❌
@Type	❌
@Value	❌
@Controller	❌
@Body	✅
@Consumes	✅
@CookieValue	✅
@CustomHttpMethod	✅
@Delete	✅
@Error	✅
@Filter	❌
@FilterMatcher	❌
@Get	✅
@Head	✅
@Header	✅
@Headers	✅
@HttpMethodMapping	✅
@Options	✅
@Part	✅
@Patch	✅
@PathVariable	✅
@Post	✅
@Produces	✅
@Put	✅

Annotation	Inherited
@QueryValue	✅
@RequestAttribute	✅
@RequestAttributes	✅
@RequestBean	✅
@Status	✅
@Trace	✅
@UriMapping	✅
@Client	❌
@JacksonFeatures	❌
@Delete	✅
@Endpoint	❌
@Read	✅
@Sensitive	✅
@Selector	✅
@Write	✅
@Liveness	❌
@Readiness	❌
@MessageBody	✅
@MessageHeader	✅
@MessageHeaders	✅
@MessageListener	❌
@MessageMapping	✅
@MessageProducer	❌
@SendTo	✅
@CircuitBreaker	❌
@Fallback	❌
@Recoverable	❌
@Retryable	❌
@Refreshable	❌
@ScopedProxy	❌
@ThreadLocal	❌
@EventListener	✅
@RequestScope	❌
@Async	❌
@ExecuteOn	❌
@Scheduled	❌
@SessionValue	✅
@ContinueSpan	✅
@NewSpan	✅
@SpanTag	✅
@Validated	✅

Annotation	Inherited
@ClientWebSocket	❌
@OnClose	✅
@OnError	✅
@OnMessage	✅
@OnOpen	✅
@ServerWebSocket	❌
@WebSocketComponent	❌
@WebSocketMapping	✅

When upgrading an application you may need to take action if you implement an interface or subclass a superclass and override a method.

For example the annotations defined in `javax.validation` are not inherited by default, so they must be defined again in any overridden or implemented methods.

This behaviour grants more flexibility if you need to redefine the validation rules. Note that it is still possible to inherit validation rules through meta-annotations. See the section on **Annotation Inheritance** for more information.

Error Response Format

The default value of `jackson.always-serialize-errors-as-list` is now true. That means by default the Hateoas JSON errors will always be a list. For example:

Example error response

```
{
  ...
  "_embedded": {
    "errors": [
      {
        "message": "Person.name: must not be blank"
      }
    ]
  }
}
```

To revert to the previous behavior where a singular error was populated in the message field instead of including `_embedded.errors`, set the configuration setting to false.

Runtime Classpath Scanning Removed

It is no longer possible to scan the classpath at runtime using the `scan` method of the **Environment** interface.

This functionality has not been needed for some time as scanning is implemented at build time through **Bean Introspections**.

Inject Annotations

Micronaut now provides the `jakarta.inject` annotations as a transitive dependency instead of the `javax.inject` annotations. To continue using the old annotations, add the following dependency.

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

Copy to Clipboard

Nullable Annotations

Micronaut no longer exports any third party dependency for nullability annotations. Micronaut now provides its own annotations for this purpose (**Nullable** and **NonNull**) that are used for our APIs. To continue using other nullability annotations, simply add the relevant dependency.

Internally, Micronaut makes use of a third party annotation that may manifest as a warning in your project:

```
warning: unknown enum constant When.MAYBE
  reason: class file for javax.annotation.meta.When not found
```

This warning is harmless and can be ignored. To eliminate this warning, add the following dependency to your project's compile only classpath:

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>com.google.code.findbugs</groupId>
    <artifactId>jsr305</artifactId>
</dependency>
```

Copy to Clipboard

Server Filter Behavior

In Micronaut 2 server filters could have been called multiple times in the case of an exception being thrown, or sometimes not at all if the error resulted before route execution. This also allowed for filters to handle exceptions thrown from routes. Filters have changed in Micronaut 3 to always be called exactly once for each request, under all conditions. Exceptions are no longer propagated to filters and instead the resulting error response is passed through the reactive stream.

In the case of a response being created as a result of an exception, the original cause is now stored as a response attribute (**EXCEPTION**). That attribute can be read by filters to have context for the error HTTP response.

The **OncePerRequestHttpServerFilter** class is now deprecated and will be removed in the next major release. The **OncePerRequestHttpServerFilter** stores a request attribute when the filter is executed and some functionality may rely on that attribute existing. The class will still create the attribute but it is recommended to instead create a custom attribute in your filter class and use that instead of the one created by **OncePerRequestHttpServerFilter**.

There is also a minor behavior change in when the response gets written. Any modifications to the response after the underlying `onNext` call is made will not have any effect as the response has already been written.

HTTP Compile Time Validation

Compile time validation of HTTP related classes has been moved to its own module. To continue validating controllers, websocket server classes add `http-validation` to the annotation processor classpath.

Gradle **Maven**

MAVEN

```
<dependency>
    <groupId>io.micronaut</groupId>
    <artifactId>micronaut-http-validation</artifactId>
</dependency>
```

Copy to Clipboard

Decapitalization Strategy

For many cases, one common one being introspections, getter names like `getXForwarded()` would result in the bean property being `XForwarded`. The name will now be `xForwarded`. This can affect many areas of the framework where names like `XForwarded` are used.

@Order default

Previously the default order value for the `@Order` annotation was the lowest precedence. It is now 0.

Classes Renaming

- `RxJavaRouteDataCollector` has been renamed to `DefaultRouteDataCollector`.
- `RxJavaBeanDefinitionDataCollector.html` has been renamed to `DefaultBeanDefinitionDataCollector`.
- `RxJavaHealthAggregator` has been renamed to `DefaultHealthAggregator`

Deprecation Removal

Classes, constructors, etc. that have been deprecated in previous versions of Micronaut have been removed.

Reflective Bean Map

In several places in Micronaut, it is required to get a map representation of your object. In previous versions, a reflection based strategy was used to retrieve that information if the class was not annotated with `@Introspected`. That functionality has been removed and it is now required to annotate classes with `@Introspected` that are being used in this way. Any class may be affected if it is passed as an argument or returned from any controller or client, among other use cases.

Cookie Secure Configuration

Previously the `secure` configuration for cookies was only respected if the request was determined to be sent over https. Due to a number of factors including proxies, HTTPS requests can be presented to the server as if they are HTTP. In those cases the setting was not having any effect. The setting is now respected regardless of the status of the request. If the setting is not set, cookies will be secure if the request is determined to be HTTPS.

Server Error Route Priority

Previously if a route could not be satisfied, or an `HttpStatusException` was thrown, routes for the relevant HTTP status was searched before routes that handled the specific exception. In Micronaut 3 routes that handle the exception will be searched first, then routes that handle the HTTP status.

Status Route Default Response Status

Status error routes will now default to produce responses with the same HTTP status as specified in the `@Error` annotation. In previous versions a 200 OK response was created. For example:

```
@Error(status = HttpStatus.UNSUPPORTED_MEDIA_TYPE)
String unsupportedMediaTypeHandler() {
    return "not supported";
}
```

The above method will result in a response of HTTP status 415 with a body of "not supported". Previously it would have been a response of HTTP status 200 with a body of "not supported". To specify the desired response status, either annotate the method with `@Status` or return an `HttpResponse`.

No Longer Possible to Inject a `List` of `Provider`

In Micronaut 2.x it was possible to inject a `List<javax.inject.Provider>`, although this was undocumented behaviour. In Micronaut 3.x injecting a list of `Provider` instances is no longer possible and you should instead use the **BeanProvider** API which provides `stream()` and `iterator()` methods to provide the same functionality.

Injecting ExecutorService

In previous versions of Micronaut it was possible to inject an **ExecutorService** (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html) without any qualifiers and the default Netty event loop group would be injected. Because the event loop should not be used for general purpose use cases, the injection will now fail by default with a non unique bean exception. The injection point should be qualified for which executor service is desired.

Subclasses Returned From Factories Not Injectable

It is no longer possible to inject the internal implementation type from beans produced via factories. The type returned from the factory or any of its super types are able to be injected.

For example:

JAVA

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ExecutorService;
import javax.inject.Singleton;

public class ExecutorFactory {
    @Singleton
    public ExecutorService executorService() {
        return ForkJoinPool.commonPool();
    }
}
```

In the above case, if the `ExecutorService` had been already been retrieved from the context in previous logic, a call to `context.getBean(ForkJoinPool.class)` would locate the already created bean. This behaviour was inconsistent because if the bean had not yet been created then this lookup would not work. In Micronaut 3 for consistency this is no longer possible.

You can however restore the behaviour by changing the factory to return the implementation type:

JAVA

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ExecutorService;
import javax.inject.Singleton;
public class ExecutorFactory {

    @Singleton
    public ForkJoinPool executorService() {
        return ForkJoinPool.commonPool();
    }
}
```

No Longer Possible to Define AOP Advice on a Bean Produced from a Factory with Constructor arguments

In previous versions of Micronaut it was possible to define AOP advice to a factory method that returned a class that featured constructor arguments. This could lead to undefined behaviour since the argument of the generated proxy which would be dependency injected by the framework may be different from manually constructed proxy target.

The following definition is now invalid in Micronaut 3 and above and will lead to a compilation error:

JAVA

```
import io.micronaut.context.annotation.*;
import io.micronaut.runtime.context.scope.*;

@Factory
class ExampleFactory {

    @ThreadLocal
    Test test() {
        return new Test("foo");
    }
}

class Test {
    // illegally defines constructor arguments
    Test(String name) {}
}
```

Implementations of `javax.inject.Provider` No Longer Generate Factories

In Micronaut 2.x if you defined a bean that implemented the `javax.inject.Provider` interface then the return type of the `get` method also automatically became a bean.

For example:

JAVA

```
import javax.inject.Provider;
import javax.inject.Singleton;

@Singleton
public class AProvider implements Provider<A> {
    @Override
    public A get() {
        return new AImpl();
    }
}
```

In the above example a bean of type `A` would automatically be exposed by Micronaut. This behaviour is no longer supported and instead the **@Factory** annotation should be used to express the same behaviour. For example:

JAVA

```
import io.micronaut.context.annotation.Factory;
import javax.inject.Provider;
import javax.inject.Singleton;

@Factory
public class AProvider implements Provider<A> {
    @Override
    @Singleton
    public A get() {
        return new AImpl();
    }
}
```

Fewer Executable Methods Generated for Controllers and Message Listeners

Previous versions of Micronaut specified the **@Executable** annotation as a meta-annotation on the **@Controller**, **@Filter** and **@MessageListener** annotations. This resulted in generating executable method all non-private methods of classes annotated with these annotations.

In Micronaut 3.x and above the **@Executable** has been moved to a meta-annotation of **@HttpMethodMapping** and **@MessageMapping** instead to reduce memory consumption and improve efficiency.

If you were relying on the presence of these executable methods you must explicitly annotate methods in your classes with **@Executable** to restore this behaviour.

GraalVM changes

In previous versions of Micronaut annotating a class with `@Introspected` automatically added it to the GraalVM `reflect-config.json` file. The original intended usage of the annotation is to generate **Bean Introspection Metadata** so Micronaut can instantiate the class and call getters and setters without using reflection.

Starting in Micronaut 3.x the `@Introspected` annotation doesn't add the class to the GraalVM `reflect-config.json` file anymore, because in most of the cases is not really necessary. If you need to declare a class to be accessed by reflection, use the `@ReflectiveAccess` annotation instead.

Another change is regarding the GraalVM resources created at compile-time. In previous versions of Micronaut adding a dependency on `io.micronaut:micronaut-graal` triggered the generation of the GraalVM `resource-config.json` that included all the resources in `src/main/resources` so they were included in the native image. Starting in Micronaut 3.x that is done in either the Gradle or Maven plugins.

Exception Handler Moves

Two exception handlers that were in `micronaut-server-netty` have now been moved to `micronaut-server` since they were not specific to Netty. Their package has also changed as a result.

Table 2. Package changes

Old	New
http-server-netty/src/main/java/io/micronaut/http/server/netty/converters/DuplicateRouteHandler.java	http-server/src/main/java/io/micronaut/http/server/exceptions/DuplicateRouteHandler.java
http-server-netty/src/main/java/io/micronaut/http/server/netty/converters/UnsatisfiedRouteHandler.java	http-server/src/main/java/io/micronaut/http/server/exceptions/UnsatisfiedRouteHandler.jav

Module Changes

New package for Micronaut Cassandra

The classes in Micronaut Cassandra have been moved from `io.micronaut.configuration.cassandra` to `io.micronaut.cassandra` package.

Micronaut Security

Many of the APIs in the Micronaut Security module have undergone changes. Please see the **Micronaut Security (https://micronaut-projects.github.io/micronaut-security/{micronautSecurityVersion}/guide)** documentation for the details.

Groovy changes

In previous version the missing property wouldn't set the field value to `null` as it would do for the Java code, in the version 3 it should behave in the same way.

Please refactor to use the default value in the `@Value` annotation:

GROOVY

```groovy
@Nullable
@Value('${greeting}')
protected String before = "Default greeting"

@Nullable
@Value('${greeting:Default greeting}')
protected String after
```