



Winter Reference Guide

Version: 1.0.0

Author: [Jeremy Kuhn](#)

1 Introduction

- 1.1 Design principles
- 1.2 Getting help

2 Overview

- 2.1 Winter Core
 - 2.1.1 Creating a Winter module
- 2.2 Winter Modules
 - 2.2.1 Using a module
 - 2.2.2 Available modules
- 2.3 Winter Tools
 - 2.3.1 Winter Maven Plugin

3 Winter Distribution

- 3.1 Requirements
- 3.2 Creating a Winter project
 - 3.2.1 Developing a simple Winter application
 - 3.2.2 Configuring logging
 - 3.2.3 Running the application
 - 3.2.4 Building the application image

4 Winter Core

- 4.1 Motivation
- 4.2 Prerequisites
- 4.3 Overview
 - 4.3.1 Modules and Beans
 - 4.3.2 Java module system
- 4.4 Project Setup
 - 4.4.1 Maven

- 4.4.2 Gradle
- 4.5 Bean
 - 4.5.1 Module Bean
 - 4.5.2 Wrapper Bean
 - 4.5.3 Nested Bean
 - 4.5.4 Overridable
 - 4.5.5 Lifecycle
 - 4.5.6 Visibility
 - 4.5.7 Strategy
- 4.6 Module
 - 4.6.1 The module class
 - 4.6.2 Lifecycle
 - 4.6.3 Module as component
 - 4.6.4 Module as application
- 4.7 Dependency Injection
 - 4.7.1 Bean Socket
 - 4.7.2 Socket Bean
 - 4.7.3 Wiring
- 4.8 Modular application
 - 4.8.1 Composite module
 - 4.8.2 Provided type

5 Winter Modules

- 5.1 Motivation
- 5.2 Prerequisites
- 5.3 Overview
- 5.4 Base
 - 5.4.1 Converter API
 - 5.4.2 Net API
 - 5.4.3 Reflection API
 - 5.4.4 Resource API
- 5.5 Boot
 - 5.5.1 Configuration
 - 5.5.2 Net service
 - 5.5.3 Media type service
 - 5.5.4 Resource service
 - 5.5.5 Converters
 - 5.5.6 Worker pool
 - 5.5.7 Object mapper

- 5.6 Configuration
 - 5.6.1 Configuration source
 - 5.6.2 Configuration loader
- 5.7 HTTP Base
 - 5.7.1 HTTP base API
 - 5.7.2 HTTP header service
- 5.8 HTTP Server
 - 5.8.1 HTTP Server exchange API
 - 5.8.2 HTTP Server
 - 5.8.3 Wrap-up
- 5.9 Web
 - 5.9.1 Routing
 - 5.9.2 Static handler
 - 5.9.3 Web Server
 - 5.9.4 Web Controller

6 Winter Maven Plugin

- 6.1 Usage
 - 6.1.1 Run a module application project
 - 6.1.2 Start and stop the application for integration testing
 - 6.1.3 Build a runtime image
 - 6.1.4 Build an application image
 - 6.1.5 Build a container image tarball
 - 6.1.6 Build and deploy a container image to a Docker daemon
 - 6.1.7 Build and deploy a container image to a remote repository
- 6.2 Goals
 - 6.2.1 Overview
 - 6.2.2 `winter:build-app`
 - 6.2.3 `winter:build-image`
 - 6.2.4 `winter:build-image-docker`
 - 6.2.5 `winter:build-image-tar`
 - 6.2.6 `winter:build-runtime`
 - 6.2.7 `winter:help`
 - 6.2.8 `winter:run`
 - 6.2.9 `winter:start`
 - 6.2.10 `winter:stop`

7 Winter OSS Parent

- 7.1 Dependencies
- 7.2 Maven Plugins

1

Introduction

The **Winter Framework** has been created with the objective of facilitating the creation of Java enterprise applications with maximum modularity, performance, maintainability and customizability.

New technologies are emerging all the time questioning what has been working for years, We strongly believe that we must instead recognize and preserve proven solutions and only provide what is missing or change what is no longer in line with widely accepted evolutions. The Java platform has proven to be resilient to change and offers features that make it an ideal choice to create durable and efficient applications in complex technical and organizational environments which is precisely what is expected in an enterprise world. The Winter Framework is a fully integrated suite of modules built for the Java platform that fully embrace its philosophy by keeping things well organized, strict and explicit with clean APIs and comprehensive documentation.

The Winter framework is open source and licensed under version 2.0 of the [Apache License](#).

Design principles

A Winter application is inherently modular, **modularity** is a key design principle which guarantees a proper separation of concerns providing flexibility, maintainability, stability and ease of development regardless of the lifespan of an application or the number of people involved to develop it. A Winter module is built as a standard Java module extending the [Java module system](#) with [Inversion of Control](#) and [Dependency Injection](#) performed at compile time.

The Winter Framework extends the Java compiler to generate code at compile time when it makes sense to do so which is strictly why annotations were initially created for. When done appropriately, **code generation** can be extremely valuable: issues can be detected ahead of time by analyzing the code during compilation, runtime footprint can be reduced by transferring costly processing like IoC/DI to the compiler improving runtime performance at the same time.

The framework uses a state of the art threading model and it has been designed from the ground up to be fully non-blocking and reactive in order to deliver very **high performance** while simplifying development of highly distributed applications requiring back pressure management.

The inherent modularity of the framework based on the Java module system guarantees a nice and clean project structure which prevents misuse and abuse by clearly separating the concerns and exposing **well designed APIs**.

Special attention has been paid to **configuration** and **customization** which are often overlooked and yet vital to create applications that can adapt to any environment or context.

Getting help

We provide here a reference guide that starts by an overview of the Winter core, modules and tools projects which gives a good idea of what can be done with the framework followed by a more comprehensive documentation that should guide you in the creation of a Winter project using the Winter distribution, the use of the core IoC/DI framework, the various modules including the configuration and the Web server modules and the tools to run, package and distribute Winter components and applications.

The [API documentation](#) provides plenty of details on how to use the various APIs. The [getting started guide](#) is also a good starting point to get into it.

Feel free to report bugs and feature requests or simply ask questions using [GitHub](#)'s issue tracking system if you ran in any issue or wish to see some new functionalities implemented in the framework.

2

Overview

Winter Core

The [Winter framework](#) core project provides an Inversion of Control and Dependency Injection framework for the Java™ platform. It has the particularity of not using reflection for object instantiation and dependency injection, everything being verified and done statically during compilation.

This approach has many advantages over other IoC/DI solutions starting with the static checking of the bean dependency graph at compile time which guarantees that a program is correct and will run properly. Debugging is also made easier since you can actually access the source code where beans are instantiated and wired together. Finally, the startup time of a program is greatly reduced since everything is known in advance, such program can even be further optimized with ahead of time compilation solutions like [GaalVM](#)...

The framework has been designed to build highly modular applications using standard Java modules. A Winter module supports encapsulation, it only exposes the beans that need to be exposed and it clearly specifies the dependencies it requires to operate properly. This greatly improves program stability over time and simplifies the use of a module. Since a Winter module has a very small runtime footprint it can also be easily integrated in any application.

Creating a Winter module

A **Winter Module** is a regular Java module, that requires `io.winterframework.core` and `io.winterframework.core.annotation` modules, and which is annotated with `@Module` annotation. The following *hello* module is a simple Winter module:

```
@io.winterframework.core.annotation.Module
module io.winterframework.example.hello {
    requires io.winterframework.core;
}
```


A **Winter Bean** can be a regular Java class annotated with `@Bean` annotation. A bean represents the basic building block of an application which is typically composed of multiple interconnected beans instances. The following `HelloService` bean can be used to create a basic application:

```
package io.winterframework.example.hello;

import io.winterframework.core.annotation.Bean;

@Bean
public class HelloService {

    public HelloService() {}

    public void sayHello(String name) {
        System.out.println("Hello " + name + "!!!");
    }
}
```

At compile time, the Winter framework will generate a module class named after the module, `io.winterframework.example.hello.Hello` in our example. This class contains all the logic required to instantiate and wire the application beans at runtime. It can be used in a Java program to access and use the `HelloService`. This program can be in the same Java module or in any other Java module which requires module `io.winterframework.example.hello`:

```
package io.winterframework.example.hello;

import io.winterframework.core.v1.Application;

public class Main {

    public static void main(String[] args) {
        Hello hello = Application.with(new Hello.Builder()).run();

        hello.helloService().sayHello(args[0]);
    }
}
```

Building and running with Maven

The development of a Winter module is pretty easy using [Apache Maven](#), you simply need to create a standard Java project that inherits from `io.winterframework.dist:winter-parent` project and declare a dependency to `io.winterframework:winter-core`:

```

<!-- pom.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">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>io.winterframework.dist</groupId>
    <artifactId>winter-parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <groupId>io.winterframework.example</groupId>
  <artifactId>hello</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>io.winterframework</groupId>
      <artifactId>winter-core</artifactId>
    </dependency>
  </dependencies>
</project>

```

Java source files for `io.winterframework.example.hello` module must be placed in `src/main/java` directory, the module can then be built using Maven:

```
$ mvn install
```

You can then run the application:

```
$ mvn winter:run -Dwinter.run.arguments=John
```

```

[INFO] --- winter-maven-plugin:1.0.0:run (default-cli) @ app-hello ---
[INFO] Running project: io.winterframework.example.hello@1.0.0-SNAPSHOT...
Hello John!!!

```

Building and running with pure Java

You can also choose to build your Winter module using pure Java commands. Assuming Winter framework modules are located under `lib/` directory and Java source files for `io.winterframework.example.hello` module are placed in `src/io.winterframework.example.hello` directory, you can build the module with the `javac` command:

```
$ javac --processor-module-path lib/ --module-path lib/ --module-source-path src/ -d jmods/ --module
io.winterframework.example.hello
```

The application can then be run as follows:

```

$ java --module-path lib/:jmods/ --module
io.winterframework.example.hello/io.winterframework.example.hello.Main John
Hello John!!!

```

Winter Modules

The [Winter framework](#) modules project provides a collection of components for building highly modular and powerful applications on top of the [Winter IoC/DI framework](#).

While being fully integrated, any of these modules can also be used individually in any application thanks to the high modularity and low footprint offered by the Winter framework.

The objective is to provide a complete consistent set of high end tools and components for the development of fast and maintainable applications.

Using a module

Modules can be used in a Winter module by defining dependencies in the module descriptor. For instance you can create a Web application module using the *boot* and *web* modules:

```
@io.winterframework.core.annotation.Module
module io.winterframework.example.webApp {
    requires io.winterframework.mod.boot;
    requires io.winterframework.mod.web;
}
```

A simple microservice application can then be created in a few lines of code as follows:

```
import io.winterframework.core.annotation.Bean;
import io.winterframework.core.v1.Application;
import io.winterframework.mod.base.resource.MediaTypees;
import io.winterframework.mod.web.annotation.WebController;
import io.winterframework.mod.web.annotation.WebRoute;

@Bean
@WebController
public class MainController {

    @WebRoute( path = "/message", produces = MediaTypees.TEXT_PLAIN)
    public String getMessage() {
        return "Hello, world!";
    }

    public static void main(String[] args) {
        Application.with(new WebApp.Builder()).run();
    }
}
```

Please refer to [Winter distribution](#) for detailed setup and installation instructions.

Comprehensive reference documentations are available for [Winter core](#) and [Winter modules](#).

Several example projects showing various features are also available in the [Winter example project](#). They can also be used as templates to start new Winter application or component projects.

Feel free to report bugs and feature requests in GitHub's issue tracking system if you ran in any issue or wish to see some new functionalities implemented in the framework.

Available modules

The framework currently provides the following modules.

winter-base

The foundational APIs of the Winter framework modules:

- Conversion API used to convert objects from/to other objects
- Net API providing URI manipulation as well as low level network client and server utilities
- Reflect API for manipulating parameterized type at runtime
- Resource API to read/write any kind of resources (eg. file, zip, jar, classpath, module...)

winter-boot

The boot Winter module provides base services to an application:

- a net service used for the implementation of optimized network clients and servers
- a media type service used to determine the media type of a resource
- a resource service used to access resources based on URIs
- a basic set of converters to decode/encode JSON, parameters (string to primitives or common types), media types (text/plain, application/json, application/x-ndjson...)
- a worker thread pool used to execute tasks asynchronously
- a JSON reader/writer

winter-configuration

Application configuration API providing great customization and configuration features to multiple parts of an application (eg. system configuration, multitenant configuration, user preferences...).

This module also introduces the `.cprops` configuration file format which facilitates the definition of complex parameterized configuration.

In addition, it also provides implementations for multiple configuration sources:

- a command line configuration source used to load configuration from command line arguments
- a map configuration source used to load configuration stored in map in memory
- a system environment configuration source used to load configuration from environment variables
- a system properties configuration source used to load configuration from system properties
- a `.properties` file configuration source used to load configuration stored in a `.properties` file
- a `.cprops` file configuration source used to load configuration stored in a `.cprops` file
- a Redis configuration source used to load/store configuration from/to a Redis data store with supports for configuration versioning
- a composite configuration source used to combine multiple sources with support for smart defaulting

- an application configuration source used to load the system configuration of an application from a set of common configuration sources in a specific order, for instance: command line, system properties, system environment, local `configuration.cprops` file and `configuration.cprops` file resource in the application module

Configurations are defined as simple interfaces in a module which are processed by the Winter compiler to generate configuration loaders and beans to make them available in an application with no further effort.

winter-http-base

The Winter HTTP base module provides the foundational API as well as common services for HTTP client and server development, in particular an extensible HTTP header service used to decode and encode HTTP headers.

winter-http-server

The Winter HTTP server module provides a fully reactive HTTP/1.x and HTTP/2 server implementation based on Netty.

It supports the following features:

- SSL
- HTTP compression/decompression
- Server-sent events
- HTTP/2 over cleartext upgrade
- URL encoded form data
- Multipart form data

winter-web

The Winter Web module provides advanced features on top of the HTTP server module, including:

- request routing based on path, path pattern, HTTP method, request and response content negotiation including request and response content type and language of the response.
- path parameters
- transparent payload conversion based on the content type of the request or the response from raw representation (arrays of bytes) to Java objects
- transparent parameter (path, cookie, header, query...) conversion from string to Java objects
- static resource handler to serve static resources from various location based on the resource API
- a complete set of annotations for easy REST controller development

REST controllers can be easily defined using annotations which are processed by the Winter compiler to generate the Web server configuration. The compiler also checks that everything is in order as for example that there are no conflicting routes.

Winter Tools

The Winter framework provides tools for running and building modular Java applications and Winter applications in particular. It allows for instance to create native runtime and application images providing all the dependencies required to run a modular application. It is also possible to build Docker and [OCI](#) images.

Winter Maven Plugin

The [Winter Maven Plugin](#) provides specific goals to:

- run a modular Java application.
- start/stop a modular Java application during the build process to execute integration tests.
- build native a runtime image containing a set of modules and their dependencies creating a light Java runtime.
- build native an application image containing an application and all its dependencies into an easy to install platform dependent package (eg. `.deb`, `.rpm`, `.dmg`, `.exe`, `.msi`...).
- build docker or OCI images of an application into a tarball, a Docker daemon or a container image registry.

The plugin requires [JDK](#) 14 or later and [Apache Maven](#) 3.6.0 or later.

3

Winter Distribution

The Winter distribution provides a parent POM `io.winterframework.dist:winter-parent` and a BOM `io.winterframework.dist:winter-dependencies` for developing Winter components and applications.

The parent POM inherits from the BOM which inherits from the [Winter OSS parent](#) POM. It provides basic build configuration for building Winter components and applications, including dependency management and plugins configuration. It especially includes configuration for the [Winter Maven plugin](#).

The BOM specifies the [Winter core](#) and [Winter modules](#) dependencies as well as OSS dependencies.

The Winter distribution thus defines a consistent sets of dependencies and configuration for developing, building, packaging and distributing Winter components and applications. Upgrading the Winter framework version of a project boils down to upgrade the Winter distribution version which is the version of the Winter parent POM or the Winter BOM.

Requirements

The Winter framework requires [JDK](#) 14 or later and [Apache Maven](#) 3.6.0 or later.

Creating a Winter project

The recommended way to start a new Winter project is to create a Maven project which inherits from the `io.winterframework.dist:winter-parent` project, we might also want to add a dependency to `io.winterframework:winter-core` in order to create a Winter module with IoC/DI:

```

<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">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>io.winterframework.dist</groupId>
    <artifactId>winter-parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <groupId>io.winterframework.example</groupId>
  <artifactId>sample-app</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>io.winterframework</groupId>
      <artifactId>winter-core</artifactId>
    </dependency>
  </dependencies>
</project>

```

That is all we need to develop, run, build, package and distribute a basic Winter component or application. The Winter parent POM provides dependency management and Java compiler configuration to invoke the Winter compiler during the build process as well as Winter tools configuration to be able to run and package the Winter component or application.

If it is not possible to inherit from the Winter parent POM, we can also declare the Winter BOM `io.winterframework.dist:winter-dependencies` in the `<dependencyManagement/>` section to benefit from dependency management but loosing plugins configuration which must then be recovered from the Winter parent POM.

```

<project>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>io.winterframework.dist</groupId>
        <artifactId>winter-dependencies</artifactId>
        <version>1.0.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>

```

Winter modules dependencies can be added in the `<dependencies/>` section of the project POM. For instance the following dependencies can be added to develop a REST microservice application:


```

<project>
  <dependencies>
    <dependency>
      <groupId>io.winterframework.mod</groupId>
      <artifactId>winter-boot</artifactId>
    </dependency>
    <dependency>
      <groupId>io.winterframework.mod</groupId>
      <artifactId>winter-web</artifactId>
    </dependency>
  </dependencies>
</project>

```

Please refer to the [Winter core documentation](#) and [Winter modules documentation](#) to learn how to develop with IoC/DI and how to use Winter modules.

Developing a simple Winter application

We can now start developing a sample REST application. A Winter component or application is a regular Java module annotated with `@io.winterframework.core.annotation.Module`, so the first thing we need to do is to create Java module descriptor `module-info.java` under `src/main/java` which is where Maven finds the sources to compile.

```

@io.winterframework.core.annotation.Module
module io.winterframework.example.sample_app {
  requires io.winterframework.mod.boot;
  requires io.winterframework.mod.web;
}

```

Note that we declared the `io.winterframework.mod.boot` and `io.winterframework.mod.web` module dependencies since we want to create a REST application, please refer to the [Winter modules documentation](#) to learn more.

We then can create the main class of our sample REST application in `src/main/java/io/winterframework/example/sample_app/App.java`:

```

package io.winterframework.example.sample_app;

import io.winterframework.core.annotation.Bean;
import io.winterframework.core.v1.Application;
import io.winterframework.mod.base.resource.MediaTypees;
import io.winterframework.mod.web.annotation.WebController;
import io.winterframework.mod.web.annotation.WebRoute;

@Bean
@WebController
public class App {

  @WebRoute( path = "/message", produces = MediaTypees.TEXT_PLAIN)
  public String getMessage() {
    return "Hello, world!";
  }

  public static void main(String[] args) {
    Application.with(new Sample_app.Builder()).run();
  }
}

```

Configuring logging

Winter framework is using [Log4j 2](#) for logging, Winter application logging can be activated by adding the dependency to `org.apache.logging.log4j:log4j-core`:

```
<project>
  <dependencies>
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-core</artifactId>
    </dependency>
  </dependencies>
</project>
```

If you don't include this dependency at runtime, Log4j falls back to the `SimpleLogger` implementation provided with the API and configured using `org.apache.logging.log4j.simplelog.*` system properties. The log level can then be configured by setting `-Dorg.apache.logging.log4j.simplelog.level=INFO` system property when running the application.

Log4j 2 provides a default configuration with a default root logger level set to `ERROR`, resulting in no info messages being output when starting an application. This can be changed by setting `-Dorg.apache.logging.log4j.level=INFO` system property when running the application.

However the recommended way is to provide a specific `log4j2.xml` logging configuration file in the project resources under `src/main/resources`:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration xmlns="http://logging.apache.org/log4j/2.0/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://logging.apache.org/log4j/2.0/config
https://raw.githubusercontent.com/apache/logging-log4j2/rel/2.14.0/log4j-core/src/main/resources/Log4j-config.xsd"
  status="WARN" shutdownHook="disable">

  <Appenders>
    <Console name="LogToConsole" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{DEFAULT} %highlight{%-5level} [%t] %c{1.} - %msg%n%ex"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="info">
      <AppenderRef ref="LogToConsole"/>
    </Root>
  </Loggers>
</Configuration>
```

Note that the Log4j shutdown hook must be disabled so as not to interfere with the Winter application shutdown hook, if it is not disabled, application shutdown logs might be dropped.

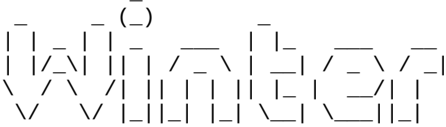
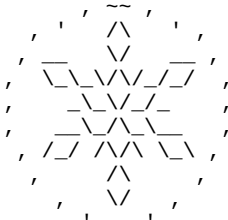
We could have chosen to provide a default logging configuration in the Winter framework itself, but we preferred to stick to standard Log4j 2 configuration rules in order to keep things simple so please refer to the [Log4j 2 configuration documentation](#) to learn how to configure logging.

Running the application

The application is now ready and can be run using the `winter:run` goal:

```
$ mvn winter:run
```

```
...
[INFO] --- winter-maven-plugin:1.0.0:run (default-cli) @ sample-app ---
[INFO] Running project: io.winterframework.example.sample_app@1.0.0-SNAPSHOT...
[=====] 100 %
=====
2021-04-08 23:50:35,261 INFO [main] i.w.c.v.Application - Winter is starting...
```



-- 1.0.2 --

Java runtime : OpenJDK Runtime Environment
Java version : 16+36-2231
Java home : /home/jkuhn/Devel/jdk/jdk-16

Application module : io.winterframework.example.sample_app
Application version : 1.0.0-SNAPSHOT
Application class : io.winterframework.example.sample_app.App

Modules :
* ...

```
2021-04-08 23:50:35,266 INFO [main] i.w.e.s.Sample_app - Starting Module
io.winterframework.example.sample_app...
2021-04-08 23:50:35,266 INFO [main] i.w.m.b.Boot - Starting Module io.winterframework.mod.boot...
2021-04-08 23:50:35,446 INFO [main] i.w.m.b.Boot - Module io.winterframework.mod.boot started in
179ms
2021-04-08 23:50:35,446 INFO [main] i.w.m.w.Web - Starting Module io.winterframework.mod.web...
2021-04-08 23:50:35,446 INFO [main] i.w.m.h.s.Server - Starting Module
io.winterframework.mod.http.server...
2021-04-08 23:50:35,446 INFO [main] i.w.m.h.b.Base - Starting Module
io.winterframework.mod.http.base...
2021-04-08 23:50:35,450 INFO [main] i.w.m.h.b.Base - Module io.winterframework.mod.http.base
started in 3ms
2021-04-08 23:50:35,545 INFO [main] i.w.m.h.s.i.HttpServer - HTTP Server (nio) listening on
http://0.0.0.0:8080
2021-04-08 23:50:35,546 INFO [main] i.w.m.h.s.Server - Module io.winterframework.mod.http.server
started in 99ms
2021-04-08 23:50:35,546 INFO [main] i.w.m.w.Web - Module io.winterframework.mod.web started in 99ms
2021-04-08 23:50:35,546 INFO [main] i.w.e.s.Sample_app - Module
io.winterframework.example.sample_app started in 281ms
```

We can now test the application:

```
$ curl http://127.0.0.1:8080/message
Hello, world!
```

The application can be gracefully shutdown by pressing `Ctrl-c`.

Building the application image

In order to create a native image containing the application and all its dependencies including JDK's dependencies, we can simply invoke the `winter:build-app` goal:

```
$ mvn winter:build-app
```

```
...  
[INFO] Building application image: /home/jkuhn/Devel/git/frmk/io.winterframework.example.sample-  
app/target/maven-winter/application_linux_amd64/sample-app-1.0.0-SNAPSHOT...  
[===== 67 % =====>  
] Creating archive sample-app-1.0.0-SNAPSHOT-application_linux_amd64.zip...
```

This uses `jpackage` tool which is an incubating feature in JDK<16, if you intend to build an application image with an old JDK, you'll need to explicitly add the `jdk.incubator.jpackage` module in `MAVEN_OPTS`:

```
$ export MAVEN_OPTS="--add-modules jdk.incubator.jpackage"
```

This will create a ZIP archive containing a native application distribution `target/sample-app-1.0.0-SNAPSHOT-application_linux_amd64.zip` which will be deployed to the local Maven repository and eventually to a remote Maven repository.

Then in order to install the application on a compatible platform, we just need to download the archive corresponding to the platform, extract it to some location and run the application. Luckily for us this can be done quite easily with Maven dependency plugin:

```
$ mvn dependency:unpack -Dartifact=io.winterframework.example:sample-app:1.0.0-  
SNAPSHOT:zip:application_linux_amd64 -DoutputDirectory=./  
...  
$ ./sample-app-1.0.0-SNAPSHOT/bin/sample-app  
...
```

It is also possible to create platform specific package such as `.deb` or a `.msi` by defining particular formats in the Winter Maven plugin configuration:

```

<project>
  <build>
    <plugins>
      <plugin>
        <groupId>io.winterframework.tool</groupId>
        <artifactId>winter-maven-plugin</artifactId>
        <executions>
          <execution>
            <id>build-app</id>
            <phase>package</phase>
            <goals>
              <goal>build-app</goal>
            </goals>
            <configuration>
              <formats>
                <format>zip</format>
                <format>deb</format>
              </formats>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>

$ mvn package
...

```

Note that there is no cross-platform support and a given platform specific format must be built on the platform it runs on.

Such platform-specific package can then be downloaded and installed using the right package manager:

```

$ mvn dependency:copy -Dartifact=io.winterframework.example:sample-app:1.0.0-SNAPSHOT:deb:application_linux_amd64 -DoutputDirectory=./
...
$ sudo dpkg -i sample-app-1.0.0-SNAPSHOT-application_linux_amd64.deb
...
$ /opt/sample-app/bin/sample-app

```

4

Winter Core

Motivation

[Inversion of Control](#) and [Dependency Injection](#) principles are not new and many Java applications have been developed following these principles over the past two decades using frameworks such as Spring, CDI, Guice... However these recognized solutions might have some issues in practice especially with the way Java has evolved and how applications should be developed nowadays.

Dependency injection errors like a missing dependency or a cycle in the dependency graph are often reported at runtime when the application is started. Most of the time these issues are easy to fix but when considering big applications split into multiple modules developed by different people, it might become more complex. In any case you can't tell for sure if an application will start before you actually start it.

Most IoC/DI frameworks are black boxes, often considered as magical because one gets beans instantiated and wired altogether without understanding what just happened and it is indeed quite hard to figure out how it actually works. This is not a problem as long as everything works as expected but it can become one when you actually need to troubleshoot a failing application.

Beans instantiation and wiring are done at runtime using Java reflection which offers all the advantages of Java dynamic linking at the expense of some performance overhead. Classpath scanning, instantiation and wiring process indeed takes some time and prevents just-in-time compilation optimization making application startup quite slow.

Although IoC frameworks make the development of modular applications easier, they often require a rigorous methodology to make it the right way. For instance, you must know precisely what components are provided and/or required by all the modules composing an application and make sure one doesn't provide a component that might interfere with another.

These points are very high level, please have a look at this [article](#) if you like to learn more about the general ideas behind the Winter framework. The Winter framework proposes a new approach of IoC/DI principles consistent with latest developments of the Java™ platform and perfectly adapted to the development of modern applications in Java.

Prerequisites

In this documentation, we'll assume that you have a working knowledge of [Inversion of Control](#) and [Dependency Injection](#) principles as well as [Object Oriented Programming](#).

Overview

The Winter framework is different in many ways and tries to address previous issues. Its main difference is that it doesn't rely on Java reflection at all to instantiate the beans composing an application (IoC) and wire them together (DI), this is actually done by a class generated by the Winter compiler at compile time.

Since beans and their dependencies are determined at compile time, errors can be raised precisely when they make sense during development or at build time.

There is also no need for complex runtime libraries since the complexity is handled by the compiler which generates a readable class providing only what is required at runtime. This presents two advantages, first applications have a small footprint and start fast since most of the processing is already done and no reflection is involved. Secondly you will be able to actually debug all parts of your application since nothing is hidden behind a complex library, you can actually see when the beans are instantiated with the `new` operator opening rooms to other compile and runtime optimization as well.

The framework also fully embraces the modular system introduced in Java 9 which basically imposes to develop with modularity in mind. A Winter module only exposes the beans that must be exposed to other modules and it clearly indicates the beans it requires to operate. All this makes modular development safer, clearer and more natural.

Modules and Beans

Inversion of control and dependency injection principles have proven to be an elegant and efficient way to create applications in an Object Oriented Programming language. A Java application basically consists in a set of interconnected objects.

A Winter application adds a modular dimension to these principles, the objects or the **beans** composing the application are created and connected in one or more isolated **modules** which are themselves composed in the **application**.

A **module** encapsulates several beans and possibly other modules. It specifies the dependencies it needs to operate and only exposes the beans that need to be exposed from the module perspective. As a result it is isolated from the rest of the application, it is unaware of how and where it is used and it actually doesn't care as long as its requirements are met. It really resembles a class which makes it very familiar to use.

A **bean** is a component of a module and more widely an application. It has required and optional dependencies provided by the module when a bean instance is created.

The **Winter compiler** is an annotation processor which runs inside the Java compiler and generates module classes based on Winter annotations found on the modules and classes being compiled.

Java module system

Before you can create your first Winter module, you must first understand what a Java module is and how it might change your life as a Java developer. If you are already familiar with it, you can skip that section and go directly to the [project setup](#) section.

The Java module system has been introduced in Java 9 mostly to modularize the overgrowing Java runtime library which is now split into multiple interdependent modules loaded when you need them at runtime or compile time. This basically means that the size of the Java runtime you need to compile and/or run your application now depends on your application's needs which is a pretty big improvement.

If you know OSGI or Maven already, you might say that modules have existed in Java for a long time but now they are fully integrated into the language, the compiler and the runtime. You can create a Java module, specify what packages are exposed and what dependencies are required and the good part is that both the compiler and JVM tell you when you do something wrong being as close as possible to the code, there's no more xml or manifest files to care about.

So how do you create a Java module? There is plenty of documentation you can read to have a complete and deep understanding of the Java module system, here we will only explain what you need to know to develop regular Winter modules.

A Java module is specified in a `module-info.java` file located in the default package. Let's assume you want to create module `io.winterframework.example.sample`, you can create the following file structure:

```
src
├── io.winterframework.example.sample
│   ├── io
│   │   ├── winterframework
│   │   │   ├── example
│   │   │   │   ├── sample
│   │   │   │   │   ├── internal
│   │   │   │   │   │   └── ...
│   │   │   └── ...
│   └── module-info.java
```

This is one way to organize the code, the only important thing is to put the `module-info.java` descriptor in the default package.

Now let's have a closer look at the module descriptor:

```
module io.winterframework.example.sample { // 1
    exports io.winterframework.example.sample; // 2
}
```


1. A module is declared using a familiar syntax starting with the `module` keyword followed by the name of the module which must be a valid Java name.
2. The `io.winterframework.example.sample` module exports the `io.winterframework.example.sample` package which means that other modules can only access public types contained in that package. Any type defined in another package within that module is only visible from within the module following usual Java visibility rules (default, public, protected, private). This basically defines a new level of encapsulation at module level. For instance, types in package `io.winterframework.example.sample.internal` are not accessible to other modules regardless of their visibility.

Now let's say you need to use some external types defined and exported in another module `io.winterframework.example.other`:

```
src
├── io.winterframework.example.sample
│   ├── ...
│   └── module-info.java
└── io.winterframework.example.other
    ├── ...
    └── module-info.java
```

If you try to reference any of these types in `io.winterframework.example.sample` module as is the compiler will complain with explicit visibility errors unless you specify that `io.winterframework.example.sample` module requires `io.winterframework.example.other` module:

```
module io.winterframework.example.sample {
    requires io.winterframework.example.other;

    exports io.winterframework.example.sample;
}
```

You should now be able to reference any public types defined in a package exported in `io.winterframework.example.other` module.

The modular system has also changed the way Java applications are built and run. Before we used to specify a classpath listing the locations where the Java compiler and the JVM should look for application's classes whereas now we should specify a module path listing the locations of modules and forget about the classpath.

If we consider previous modules, they are compiled and run as follows:

```
> javac --module-source-path src -d jmods --module io.winterframework.example.sample --module
io.winterframework.example.other
> java --module-path jmods/ --module
io.winterframework.example.sample/io.winterframework.example.sample.Sample
```

There are other subtleties like transitive dependencies, service providers or opened modules and cool features like jmod packaging and the `jlink` tool but for now that's pretty much all you need to know to develop Winter modules which are basically instrumented Java modules.

You should now have a basic understanding of how a Winter application is built and what Java technologies are involved. A Winter application results from the composition of multiple isolated modules which create and wire the beans making up the application. Almost everything is done at compile time where module classes are generated.

Project Setup

Maven

The easiest way to setup a Winter module project is to start by creating a regular Java Maven project which inherits from `io.winterframework.dist:winter-parent` project and depends on `io.winterframework:winter-core`:

```
<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">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>io.winterframework.dist</groupId>
    <artifactId>winter-parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <groupId>io.winterframework.example</groupId>
  <artifactId>sample</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  ...
  <dependencies>
    ...
    <dependency>
      <groupId>io.winterframework</groupId>
      <artifactId>winter-core</artifactId>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

Then you have to add a module descriptor to make it a Java module project. A Winter module requires `io.winterframework.core` and `io.winterframework.core.annotation` modules. If you want your module to be used in other modules it must also export the package where the module class is generated by the Winter compiler which is the module name by default. Remember that a Winter module is materialized in a regular Java class subject to the same rules as any other class in a Java module.

```
module io.winterframework.example.sample {
  requires io.winterframework.core;
  requires io.winterframework.core.annotation;

  exports io.winterframework.example.sample;
}
```

If you do not want your project to inherit from `io.winterframework.dist:winter-parent` project, you'll have to explicitly specify compiler source and target version (≥ 9), dependencies version and configure the Maven compiler plugin to invoke the Winter compiler.

```

<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">
  <modelVersion>4.0.0</modelVersion>
  <groupId>io.winterframework.example</groupId>
  <artifactId>sample</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>9</maven.compiler.source>
    <maven.compiler.target>9</maven.compiler.target>
    <version.winter>1.0.0</version.winter>
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>io.winterframework.dist</groupId>
        <artifactId>winter-dependencies</artifactId>
        <version>${version.winter}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>io.winterframework</groupId>
      <artifactId>winter-core</artifactId>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      ...
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <annotationProcessorPaths>
            <path>
              <groupId>io.winterframework</groupId>
              <artifactId>winter-core-compiler</artifactId>
              <version>${version.winter}</version>
            </path>
          </annotationProcessorPaths>
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>
</project>

```

A Winter module is built just as a regular Maven project using maven commands (compile, package, install...). The module class is generated and compiled during the **compile** phase and included in the resulting JAR file during the **package** phase. If anything related to IoC/DI goes wrong during compilation, the compilation fails with explicit compilation errors reported by the Winter compiler.

Gradle

Since version 6.4, it is also possible to use [Gradle](#) to build Winter module projects. Here is a sample `build.gradle` file:

```
plugins {
    id 'application'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'io.winterframework:winter-core:1.0.0'
    annotationProcessor 'io.winterframework:winter-core-compiler:1.0.0'
}

java {
    modularity.inferModulePath = true
    sourceCompatibility = JavaVersion.VERSION_1_9
    targetCompatibility = JavaVersion.VERSION_1_9
}

application {
    mainModule = 'io.winterframework.example.hello'
    mainClassName = 'io.winterframework.example.hello.App'
}
```

Bean

As you already know, a Java application can be reduced to the composition of objects working together. In a Winter application, these objects are instantiated and injected into each other by one or more modules. Inside a module, a bean basically specifies what it needs to create a bean instance (DI) and how to obtain it (IoC).

A bean and a bean instance are two different things that should not be confused. A bean can result in multiple bean instances in the application whereas a bean instance always refers to exactly one bean. A bean is like a plan used to create instances.

A bean is fully identified by its name and the module in which it resides. The following notation is used to represent a bean qualified name: `[MODULE]:[BEAN]`. As a consequence, two beans with the same name cannot exist in the same module but it is safe to have multiple beans with the same name in different modules.

Module Bean

Module bean is the primary type of beans you can create in a Winter module. It is defined by a concrete class annotated with the `@Bean` annotation.

```
import io.winterframework.core.annotation.Bean;
```

```
@Bean
public class SomeBean {
    ...
}
```

In the previous code we created a bean of type `SomeBean`. At compile time, the Winter compiler will include it in the generated module class that you'll eventually use at runtime to obtain `SomeBean` instances.

By default, a bean is named after the simple name of the class starting with a lower case (eg. `someBean` in our previous example). This can be specified in the annotation using the `name` attribute:

```
@Bean(name="customSomeBean")
public class SomeBean {
    ...
}
```

Wrapper Bean

A wrapper bean is a particular form of bean used to define beans whose code cannot be instrumented with Winter's annotations or that require more complex logic to create the instance. This is especially the case for legacy code or third party libraries.

A wrapper bean is defined by a concrete class annotated with both `@Bean` and `@Wrapper` annotations which basically wraps the actual bean instance and include the instantiation, initialization and destruction logic. It must implement the `Supplier<E>` interface which specifies the actual type of the bean as formal parameter.

```
@Bean
@Wrapper
public class SomeWrapperBean implements Supplier<SomeLegacyBean> {

    private SomeLegacyBean instance;

    public SomeWrapperBean() {
        // Creates the wrapped instance
        this.instance = ...
    }

    SomeLegacyBean get() {
        // Returns the wrapped instance
        return this.instance;
    }
    ...
}
```

In the previous code we created a bean of type `SomeLegacyBean`. One instance of the wrapper class is used to create exactly one bean instance and it lives as long as the bean instance is referenced.

Since a wrapper bean is annotated with `@Bean` annotation, it can be configured in the exact same way as a module bean except that it only applies to the wrapper instance which is responsible to configure the actual bean instance. The wrapper instance is never exposed, only the actual bean instance wrapped in it is exposed. As for module beans, `SomeLegacyBean` instances can be obtained using the generated Module class.

Note that since a new wrapper instance is created every time a new bean instance is requested, a wrapper class is not required to return a new or distinct result in the `get()` method. Nonetheless a wrapper instance is used to create, initialize and destroy exactly one instance of the supplied type and as a result it is good practice to have the wrapper instance always return the same bean instance. This is especially true and done naturally when initialization or destruction methods are specified.

When designing a prototype wrapper bean, particular care must be taken to make sure the wrapper does not hold a strong reference to the wrapped instance in order to prevent memory leak when a prototype bean instance is requested by the application. It is strongly advised to rely on `WeakReference<>` in that particular use case.

Nested Bean

A nested bean is, as its name suggests, a bean inside a bean. A nested bean instance is obtained by invoking a particular method on another bean instance. Instances thus obtained participate in dependency injection but unlike other types of bean they do not follow any particular lifecycle or strategy, the implementor of the nested bean method is free to decide whether a new instance should be returned on each invocation.

A nested bean is declared in the class of a bean, by annotating a non-void method with no arguments with `@NestedBean` annotation. The name of a nested bean is given by the name of the bean providing the instance and the name of the annotated method following this notation:

`[MODULE] : [BEAN] . [METHOD_NAME]`.

```
@Bean
public class SomeBean {

    ...

    @NestedBean
    public SomeNestedBean nestedBean() {
        ...
    }
}
```

It is also possible to *cascade* nested beans.

Overridable

A module bean or a wrapper bean can be declared as overridable which allows to override the bean inside the module by a socket bean of the same type.

An overridable bean is defined as a module bean or a wrapper bean whose class has been annotated with `@Overridable`. This basically tells the Winter compiler to create an extra optional socket bean with the particular feature of being able to take over the bean when an instance is provided on module instantiation.

```
@Bean
@Overridable
public class SomeBean {

}
```

Lifecycle

All bean instances follow the subsequent lifecycle in a module:

1. A bean instance is created
2. It is initialized
3. It is active
4. It is "eventually" destroyed

Let's examine each of these steps in details.

A bean instance is always created in a module, when a bean instance is created greatly depends on the context in which it is used, it can be created when a module instance is started or when it is required in the application. In order to create a bean instance the module must provide all the dependencies required by the bean. After that it sets any optional dependencies available on the instance thus obtained. This is actually when and where dependency injection takes place, this aspect will be covered more in details in following sections, for now all you have to know is that when requested the module creates a fully wired bean instance.

After that the module invokes initialization methods on the bean instance to initialize it. An initialization method is declared on the bean class using the `@Init` annotation:

```
@Bean
public class SomeBean {

    @Init
    public void init() {
        ...
    }
}
```

You can specify multiple initialization methods but the order in which they are invoked is undetermined. Inheritance is not considered here, only the methods annotated on the bean class are considered. Bean initialization is useful when you want to execute some code after dependency injection to make the bean instance fully functional (eg. initialize a connection pool, start a server socket...).

After that, the bean instance is active and can be used either directly by accessing it from the module or indirectly through another bean instance where it has been injected.

A bean instance is "eventually" destroyed, typically when its enclosing module instance is stopped. Just as you specified initialization methods, you can specify destruction methods to be invoked when a bean instance is destroyed using the `@Destroy` annotation:

```
@Bean
public class SomeBean {

    @Destroy
    public void destroy() {
        ...
    }
}
```

As for initialization methods, you can specify multiple destruction methods but the order in which they are invoked is undetermined and inheritance is also not considered. Bean destruction is useful when you need to free resources that have been allocated by the bean instance during application operation (eg. shutdown a connection pool, close a server socket...).

In case of wrapper beans, the initialization and destruction of a bean instance is delegated to the initialization and destruction methods specified on the wrapper bean which respectively initialize and destroy the actual bean instance wrapped in the wrapper bean.

```
@Bean
@Wrapper
public class SomeWrapperBean implements Supplier<SomeLegacyBean> {

    private SomeLegacyBean instance;

    public SomeWrapperBean() {
        // Creates the wrapped instance
        this.instance = ...
    }

    @Init
    public void init() {
        // Initialize the wrapped instance
        this.instance.start();
    }

    @Destroy
    public void destroy() {
        // Destroy the wrapped instance
        this.instance.stop();
    }
    ...
}
```

We stated here that all bean instances are eventually destroyed but this is actually not always the case. Depending on the bean strategy and the context in which it is used, it might not be destroyed at all, hopefully workarounds exist to make sure a bean instance is always properly destroyed. We'll cover this more in detail when we'll describe [bean strategies](#).

Visibility

A bean can be assigned a public or private visibility. A public bean is exposed by the module to the rest of the application whereas a private bean is only visible from within the module.

Bean visibility is set in the `@Bean` annotation in the visibility attribute:

```
@Bean(visibility=Visibility.PUBLIC)
public class SomeBean {

}
```

Strategy

A bean is always defined with a particular strategy which controls how a module should create a bean instance when one is requested, either during dependency injection when a module requires a bean instance to inject in another bean instance or during application operation when some application code requests a bean instance to a module instance.

Singleton

The singleton strategy is the default strategy used when no explicit strategy is specified on a bean class. A Winter module only creates one single instance for a singleton bean. That same instance is returned every time an instance of that bean is requested. It is then shared among all dependent beans through dependency injection and also the application if it has requested an instance.

A singleton bean is specified explicitly by setting the `strategy` attribute to `Strategy.SINGLETON` in the `@Bean` annotation:

```
@Bean(strategy = Strategy.SINGLETON)
public class SomeSingletonBean {

}
```

Modules easily support the bean lifecycle for singleton beans since a module instance holds singleton bean instances by design, they can then be properly destroyed when the module instance is stopped.

Particular care must be taken when a singleton bean instance is requested to a module instance by the application as the resulting reference will point to a *managed* instance which will be destroyed when the module instance is stopped leaving the instance referenced in the application in an unpredictable state.

A singleton bean is the basic building block of any application which explains why it is the default strategy. An application is basically made of multiple long living components rather than volatile disposable components. A server is a typical example of singleton bean, it is created when the application is started, initialized to accept requests and destroyed when the application is stopped.

A singleton instance is held by exactly one module instance, if you instantiate a module twice, you'll get two singleton bean instances, one in the first module instance and the other in the second module instance. This basically differs from the standard singleton pattern, you'll see more in detail why this actually matters when we'll describe [composite modules](#).

Prototype

A prototype bean results in the creation of as many instances as requested. All dependent beans in the module get a different bean instance and each time a bean instance is requested to a module instance by the application a new instance is also created.

A prototype bean is specified by setting the `strategy` attribute to `Strategy.PROTOTYPE` in the `@Bean` annotation:

```
@Bean(strategy = Strategy.PROTOTYPE)
public class SomeBean {

}
```

Unlike singleton beans, modules can't always fully support the bean lifecycle for prototype beans. All prototype beans instances are kept in the module instance in order to destroy them when it is stopped. Modules use weak references to prevent memory leaks so that dereferenced instances are automatically removed from the internal registry when the garbage collector reclaims them. This works well for prototype bean instances injected into singleton bean instances since they are actually referenced until the module instance is stopped just like any singleton bean instance. It becomes tricky when a prototype bean instance is requested by the application. In that case, the prototype bean instance is removed from the module instance when it is dereferenced from the application and reclaimed by the garbage collector leaving no chance for the module instance to destroy it properly. The actual behavior is more subtle because a dereferenced prototype bean instance might actually be destroyed when a module is stopped before the instance is reclaimed by the garbage collector.

As a result, it is not recommended to define destruction methods on a prototype bean but if you really need to, you can make your bean implement `AutoCloseable`, specify the `close()` method as the unique destruction method and request prototype bean instances from the application using a try-with-resources block:

```
@Bean(strategy = Strategy.PROTOTYPE)
public class SomeBean implements AutoCloseable {

    @Destroy
    public void close() throws Exception {
        ...
    }
}
```

Then when requesting a prototype bean instance from the application:

```
try(SomeBean bean = module.someBean()) {
    ...
}
```

As soon as the program exits the try-with-resources block the bean instance is properly destroyed, then dereferenced and eventually reclaimed by the garbage collector and finally removed from the module instance. However you should make sure that the `close()` method can be called twice since it actually might.

Prototype beans should be used whenever there is a need to hold a state in a particular context. An HTTP client is a typical example of a stateful instance, different instances should be created and injected in singleton beans so they can deal with concurrency independently to make sure requests are sent only after a response to the previous request has been received.

That might not be the smartest way to use HTTP clients in an application but it gives you the idea.

Prototype beans can also be used to implement the factory pattern, just like a factory, you can request new bean instances on a module. Winter framework makes this actually very powerful since there's no runtime overhead, modules can be created and used anywhere and you never have to worry about the boiler plate code that instantiates the bean since it is generated for you by the framework.

Module

A Winter module can be seen as an isolated collection of beans. The role of a module is to create and wire bean instances in order to expose logic to the application.

In practice, a module is materialized by the class generated by the Winter compiler during compilation and which results from the processing of Winter annotations.

A module is isolated from the rest of the application through its module class which clearly defines the beans exposed by the module and what it needs to operate. As a result, a module doesn't care when and how it is used in an application as long as its requirements are met.

Isolation is actually what makes the Winter framework so special as it greatly simplifies the development of complex modular applications.

A module is defined as a regular Java module annotated with the `@Module` annotation:

```
@Module
Module io.winterframework.sample.sampleModule {
    ...
}
```

The module class

Java modules annotated with `@Module` will be processed by the Winter compiler at compile time. The Winter compiler generates one **module class** per module providing all the code required at runtime to create and wire bean instances.

This class is the entry point of a module and serve several purposes:

- encapsulate beans instances creation and wiring logic
- implement bean instance lifecycle
- specify required or optional module dependencies
- expose public beans
- hide private beans
- guarantee a proper isolation of the module within the application

This regular Java class can be instantiated like any other class. It relies on a minimal runtime library barely visible which makes it self-describing and very easy to use.

Let's see how it looks like for the `io.winterframework.sample.sampleModule` module and `SomeBean` bean, the module class would be used as follows:

```
SampleModule module = new SampleModule.Builder().build(); // 1
module.start(); // 2

SomeBean someBean = module.someBean(); // 3
// Do something useful with someBean
module.stop(); // 4
```

1. The `SampleModule` class is instantiated
2. The module is started
3. The `SomeBean` instance is retrieved
4. Eventually the module is stopped

There are two important things to notice here, first you control when, where and how many times you want to instantiate a module, which brings great flexibility in the way modules are used in your application. For instance integrating a Winter module in an existing code is pretty straightforward as it is plain old Java, it is also possible to create and use a module instance during application operation (eg. when processing a request). Secondly beans are exposed with their actual types through named methods which eventually produces more secure code because static type checking can (finally) be performed by the compiler.

Module classes provide dedicated builders to facilitate the creation of complex modules instances with multiple required and optional dependencies.

By default, the module class is named after the last identifier of the module name and generated in a package named after the module. The full class name can be specified in the annotation using the `className` attribute:

```
@Module(className="io.winterframework.sample.CustomSampleModule")
Module io.winterframework.sample.sampleModule {
    ...
}
```

The module class is like any other class in the module, if you want to use it outside the module you have to explicitly export its package in the module descriptor:

```
@Module
Module io.winterframework.sample.sampleModule {
    exports io.winterframework.sample.sampleModule;
}
```

Most of the time this is something you'll do especially if you want to create [composite modules](#), however if you only use the module class from within the module, typically in a main method or embedded in some other class, you won't have to do it.

Note that the Java compiler fails if you try to export a package which is empty before compilation, since the module class is generated this might actually happen, so you need to make sure the class will be generated in a package containing some code. This is not an ideal situation however a module usually defines and exports a package named after its name so this should solve the issue.

Lifecycle

Just like a bean instance, a module follows a lifecycle:

1. A module instance is created
2. It is started
3. It is active
4. It is stopped

Let's examine each of these steps in details.

A module instance can be created directly in the application or indirectly inside a composite module. A module defines a dedicated **Builder** class that must be used to build the module instance. Relying on a builder is very helpful when considering complex modules with many required and optional dependencies.

The instance must then be started to make it operational. During this phase, all Winter modules composed in the module are instantiated and started and all the beans defined in the module are created and initialized. Dependency injection is performed naturally as beans are created. Since everything has been validated at compile time, we know for sure that everything will work properly.

A module is actually composed by the beans it defines and the beans defined in the modules it composes. This is discussed in details in the [Modular application](#) section.

Once the module instance is active, beans are exposed to the application.

Finally, a module instance is stopped to release resources held by the beans instances. During this phase, beans are destroyed in the reverse order of their creation and composed Winter modules are stopped.

Module as component

Winter modules are very flexible and can be used in many situations. You can for instance develop Winter modules to create reusable software components. Such components would benefit from inversion of control and dependency injection capabilities offered by the framework without interfering with the applications that uses them. A Winter module has also a very low runtime footprint since it creates objects and wires them in a fixed and deterministic way, it can then be created at any time in any situations.

Standalone component

You can imagine a standalone module used to interface with an external system like a coffee maker module for example. From the outside a coffee maker is actually quite simple:

- it requires electricity to operate
- you have to fill it with coffee beans
- you have to supply some water as well
- then you can make some tasty coffee

From the inside on the other hand it can be much more complex than this, it is probably composed of multiple internal components that you actually don't care about as long as the coffee is good.

Let's try to imagine what kind of interface would be exposed by the `io.winterframework.sample.coffeeMakerModule` module without anticipating any implementation.

First of all it would probably export the module's package as it is intended to be used from outside the module:

module-info.java

```
@Module
Module io.winterframework.sample.coffeeMakerModule {
    exports io.winterframework.sample.coffeeMakerModule;
}
```

It might expose three singleton beans:

- `io.winterframework.sample.coffeeMakerModule:coffeeBeansContainer` to be able to fill the coffee maker with beans
- `io.winterframework.sample.coffeeMakerModule:waterReservoir` for water supply
- `io.winterframework.sample.coffeeMakerModule:coffeeMaker` to actually make some coffee

CoffeeBeansContainer

```
public interface CoffeeBeansContainer {
    void fill(CoffeeBean[] beans);
}
```

WaterReservoir

```
public interface WaterReservoir {  
    void fill(int waterQuantity);  
}
```

CoffeeMaker

```
public interface CoffeeMaker {  
    Coffee make();  
}
```

Inside a coffee shop application, you might instantiate several coffee maker modules used in the following way:

```
PowerSupply powerSupply = ... // Get some  
power supply  
  
CoffeeMakerModule coffeeMakerModule = new CoffeeMakerModule.Builder(powerSupply).build();  
coffeeMakerModule.start();  
  
ArabicaCoffeeBeans[] coffeeBeans = ... // Get some  
tasty coffee beans  
coffeeMakerModule.coffeeBeansContainer().fill(coffeeBeans); // fill the  
coffee beans container  
coffeeMakerModule.waterReservoir().fill(1.5); // fill the  
water reservoir with 1.5 Liters  
  
CoffeeMaker coffeeMaker = coffeeMakerModule.coffeeMaker(); // Get the  
coffee maker instance  
  
Coffee coffee_1 = coffeeMaker.make(); // Deliver some tasty coffees  
...  
Coffee coffee_n = coffeeMaker.make();  
  
coffeeMakerModule.stop();
```

The goal of this example was to show the benefits of using Winter modules as standalone components in an application. As you can see:

- implementation details are completely hidden: you don't know and you don't have to know how the beans container, the water reservoir and the coffee maker are working together.
- dependencies are clearly exposed: you must provide some power supply to instantiate the module.
- only significant functionalities are exposed.
- if you look closely, you'll see that no particular technical framework is visible: from a code perspective, the application doesn't see and don't need to know it is using a Winter module, everything is also statically typed and self-describing.

Factory component

You can also create a module as a generic factory or builder to ease the creation of complex objects. If we consider previous example from a different perspective, we can imagine a factory module that could be used to build coffee makers from raw materials.

It would also probably export the module's package so it can be used from outside the module:

module-info.java

```
@Module
Module io.winterframework.sample.coffeeMakerFactoryModule {
    exports io.winterframework.sample.coffeeMakerFactoryModule;
}
```

Then it would expose the `io.winterframework.sample.coffeeMakerFactoryModule:coffeeMaker` prototype bean:

```
public interface CoffeeMaker {

    void fillWithCoffeeBeans(CoffeeBeans[] beans);

    void filleWithWater();

    Coffee makeCoffee();
}
```

Inside a cooking appliances factory application, you might instantiate one or more coffee maker factory module to produce coffee makers:

```
CoffeeMakerFactoryModule coffeeMakerFactoryModule = new
CoffeeMakerFactoryModule.Builder(rawMaterials...).build();
coffeeMakerFactoryModule.start();

CoffeeMaker coffeeMaker_1 = coffeeMakerFactoryModule.coffeeMaker(); // We can massively produce
coffee makers
...
CoffeeMaker coffeeMaker_n = coffeeMakerFactoryModule.coffeeMaker();

coffeeMakerFactoryModule.stop();
```

The context and the approach are clearly different here, the purpose of a factory component module is to enable developers to use IoC/DI to easily create complex objects.

Processing component

Dependency Injection is mostly about interconnecting objects to form an application, but this is more a consequence of how IoC/DI frameworks are designed than an absolute fact. A Winter module is cheap, it can also be created and used during the operation of an application to process requests. This makes it possible to have data objects or contextual objects injected and used in bean instances.

Let's say we have created a highly customizable coffee maker, capable of producing a coffee based on many parameters: steam pressure, temperature, grinding size... These parameters have to be used in various components of the coffee maker. These data have to be provided each time a customer orders a coffee

Propagating the right data to the right coffee maker component can be a tedious task. A Winter module can be created to inject data where they are needed based on the dependencies of each beans composing the coffee maker module and eventually process the request.


```

public Coffee orderCoffee(Param_1 p1, Param_2 p2, ... Param_n pn) {
    // Receive a large amount of parameters to make a coffee

    try {
        CoffeeMakerModule coffeeMakerFactoryModule = new CoffeeMakerFactoryModule.Builder(p1, p2,
        ... pn).build(); // Parameters are injected only where they are needed
        coffeeMakerModule.start();

        return coffeeMakerModule.coffeeMaker().makeCoffee();
    }
    finally {
        coffeeMakerFactoryModule.stop();
    }
}

```

You can then benefit from dependency injection inside the business logic, performance shouldn't be impacted by bean instantiation or dependency injection logic because the creation of a module instance is no different than creating some objects with the **new** operator and invoking some setter methods. This is especially interesting when you have to process very complex requests with a lot of input data.

Module as application

A Winter module can also be used to bootstrap a whole application. In such situation one single Winter module is started as an application in the main method of a class. This class can be defined in the same module but this is not mandatory as long as it has access to the application module. The role of an application module is to create and start all the components of the application.

```

public static void main(String[] args) {
    CoffeeMakerModule coffeeMakerModule = Application.with(new
    CoffeeMakerModule.Builder(...)).run();
    ...
}

```

An application module is basically a regular module whose lifecycle is managed by the **Application** class. A module instance is created and started when the **run()** method is invoked and eventually stopped when the JVM shuts down.

Note that this involves a shutdown hook, as a consequence there is actually no guarantee that the module will be stopped especially if the JVM is not gracefully shut down.

Furthermore, a Winter application outputs a customizable **Banner** on startup providing useful environment information in the application log.

[illegible]

The `StandardBanner` is displayed by default but you can specify custom implementations as well:

Dependency Injection

In order to understand how this works, you could imagine that each bean exposes multiple sockets and that multiple wires leave the bean, as many as necessary. After creating and initializing bean instances, the module has to plug these wires into compatible sockets. The type of the wire, which is the type of the bean, must match the type of the socket, which is the type of the dependency defined in the bean.

Dependency injection is validated and fully determined at compile time, the module class just instantiates and injects beans in a predetermined order without having to worry about missing dependencies or dependency cycles amongst others.

Bean Socket

A **bean socket** designates a bean dependency. A bean can have two kinds of dependencies and then define two kinds of sockets: required and optional. Required dependencies must be resolved to create an operational bean instance whereas optional dependencies add extra capabilities to the bean instance. As a consequence, a module has to wire every required sockets, the Winter compiler actually raises compilation errors on beans with unresolved required sockets.

The Winter framework tries to be as less intrusive as possible, a bean specifies its sockets using standard Java as constructor arguments for required sockets and setter methods for optional sockets. Creating a bean is then very natural.

A bean socket is fully identified by its name, the name of the bean which defines it and the module in which the bean resides. The following notation is used to represent a bean socket qualified name: `[MODULE]:[BEAN]:[SOCKET_NAME]`. On a given bean in a given module, it is not possible to specify two sockets with the same name.

Let's go back to our coffee maker example and define the dependencies of the `CoffeeMaker` bean.

CoffeeMakerImpl

```
@Bean
public class CoffeeMakerImpl implements CoffeeMaker {

    private PowerSupply powerSupply;

    private WaterReservoir waterReservoir;

    private CoffeeBeansContainer coffeeBeansContainer;

    public CoffeeMakerImpl(PowerSupply powerSupply, WaterReservoir waterReservoir,
CoffeeBeansContainer coffeeBeansContainer) {
        this.powerSupply = powerSupply;
        this.waterReservoir = waterReservoir;
        this.coffeeBeansContainer = coffeeBeansContainer;
    }

    public Coffee make() {
        ...
    }
}
```

The `io.winterframework.sample.coffeeMakerModule:coffeeMakerImpl` bean then specifies three required sockets:

- `io.winterframework.sample.coffeeMakerModule:coffeeMakerImpl:powerSupply`
- `io.winterframework.sample.coffeeMakerModule:coffeeMakerImpl:waterReservoir`
- `io.winterframework.sample.coffeeMakerModule:coffeeMakerImpl:coffeeBeansContainer`

There should be only one public constructor defined in a bean class, this is actually proper bean design. Defining multiple constructors means that there are probably some dependencies not really required by the bean to work properly. Only required dependencies should be specified in a single bean constructor and optional dependencies in multiple setter methods. However if for some reasons multiple public constructors are defined on a bean class, you can explicitly specify which constructor to consider using the `@BeanSocket` annotation.

```
@Bean
public class CoffeeMakerImpl implements CoffeeMaker {

    @BeanSocket
    public CoffeeMakerImpl(PowerSupply powerSupply, WaterReservoir waterReservoir,
        CoffeeBeansContainer coffeeBeansContainer) {
        ...
    }

    public CoffeeMakerImpl(PowerSupply powerSupply, WaterReservoir waterReservoir,
        CoffeeBeansContainer coffeeBeansContainer, SomeOptionalDependency dependency) {
        ...
    }
}
```

The coffee maker should now have everything it needs to make coffee but let's say we want the coffee maker to be able to make cappuccinos, it will then need a `MilkFrother`. The coffee maker can use a `MilkFrother` when available but it doesn't require a `MilkFrother` to make coffee, only to make cappuccinos, as a result it should be declared in an optional socket.

```
@Bean
public class CoffeeMakerImpl implements CoffeeMaker {

    ...
    private MilkFrother milkFrother

    ...
    public void setMilkFrother(MilkFrother milkFrother) {
        this.milkFrother = milkFrother;
    }

    public Coffee make() {
        ...
        if(this.milkFrother != null) {
            // Do something useful with the milk frother
            ...
        }
    }
}
```

The `io.winterframework.sample.coffeeMakerModule:coffeeMakerImpl` bean now specifies one optional socket: `io.winterframework.sample.coffeeMakerModule:coffeeMakerImpl:milkFrother`.

By convention, every setter method on a bean is considered an optional socket, this enforces proper bean design. However in some situations you might need to explicitly specify which setter methods are sockets. In order to do that, you need to annotate every socket setter method of the bean with the `@BeanSocket` annotation.

```

@Bean
public class CoffeMakerImpl implements CoffeMaker {

    ...
    @BeanSocket
    public void setMilkFrother(MilkFrother milkFrother) {
        ...
    }

    ...
    public void setSomethingElse() {
        ...
    }
}

```

Note that Winter annotation are not inherited from ancestor class, the Winter compiler only considers the bean class annotated with `@Bean` so you must explicitly override setter methods to specify optional sockets defined in a class ancestor. This might not be obvious but it is actually the safer way that gives a perfect control on the sockets you want to expose in your beans.

Single and multiple

We can differentiate two kinds of bean socket: single socket and multiple socket. A single socket can be of any type except arrays, `java.util.List`, `java.util.Set` and `java.util.Collection` whereas the type of a multiple socket is necessarily an array, a `java.util.List`, a `java.util.Set` or a `java.util.Collection`. Multiple beans can be wired to a multiple socket whereas only one bean is wired to a single socket.

Lazy

A socket can be annotated with the `@Lazy` to indicate that a bean instance supplier should be provided instead of an actual bean instance. A lazy socket must then necessarily be of type `Supplier<E>` which specifies the actual type of the socket as formal parameter.

A lazy socket allows a dependent bean to lazily retrieve a bean instance. This presents several advantages when prototype beans are wired into a lazy socket, it is then possible to create fully wired bean instances on demand during the operation of a module and use them when processing a request for instance.

Socket Bean

Bean sockets designates the dependencies of a single bean. All beans in a module must be operational for a module to work properly as a consequence all beans required sockets must be resolved but what if one or more *plugs* are missing inside the module to match all these sockets? The dependency can then be declared at module level using a particular kind of bean: the **socket bean**.

From inside a module, a socket bean is considered as any regular beans as it takes part in the dependency injection process. From outside the module, it designates a module dependency that is provided when a module is instantiated.

Unlike other type of beans, a socket bean is not a concrete class, it must be an interface annotated with `@Bean` extending the `Supplier<E>` interface. The supplier's formal parameter designates the type of the dependency to provide.

Let's say the coffee maker module does not provide any `PowerSupply` bean internally, this makes sense since a power supply might be required to make coffee but it is clearly unrelated. We must then find a way to provide a `PowerSupply` inside the module to make it work. We can then create a `PowerSupplySocket` socket bean inside the coffee maker module.

```
@Bean
public interface PowerSupplySocket implements Supplier<PowerSupply> {}
```

This creates socket bean `io.winterframework.sample.coffeeMakerModule:powerSupplySocket` in the module `io.winterframework.sample.coffeeMakerModule`. As you can imagine, this bean can be injected in other module's beans just like any regular beans.

The module class generated by the Winter compiler now defines an argument of type `PowerSupply` in the module's builder constructor, we must then provide a `PowerSupply` instance in order to instantiate the module.

```
PowerSupply powerSupply = ...
CoffeeMakerModule coffeeMakerModule = new CoffeeMakerModule.Builder(powerSupply).build();
...
```

A socket bean appears in the builder constructor when it is wired to a required bean socket inside the module. On the other hand, a socket bean wired to an optional bean socket appears in an extra method of the module's builder class.

We might want to be able to stick a brand sticker on the coffee maker, this is obviously completely optional and external to the coffee maker module. We can then define a `BrandStickerSocket` in the module.

```
BrandSticker brandSticker = ...
CoffeeMakerModule coffeeMakerModule = new
CoffeeMakerModule.Builder(powerSupply).brandSticker(brandSticker).build();
...
```

It is interesting to notice here that a Winter module explicitly specifies its dependencies which is extremely valuable to create complex modular applications involving multiple people working together, one can easily understand how to use another one's module without mentioning the fact that the compiler can actually check that everything fits together since beans, modules and modules builder arguments are all statically typed.

Wiring

The Winter compiler wires beans together based on the sockets defined in the module. A viable module is a module that has:

- all its required sockets resolved, either internally with another bean in the module or externally through a socket bean
- no cycles in the resulting graph of beans

Autowiring

By default, the Winter compiler tries to automatically wire the beans in a module based on their respective types and the types of the sockets they expose.

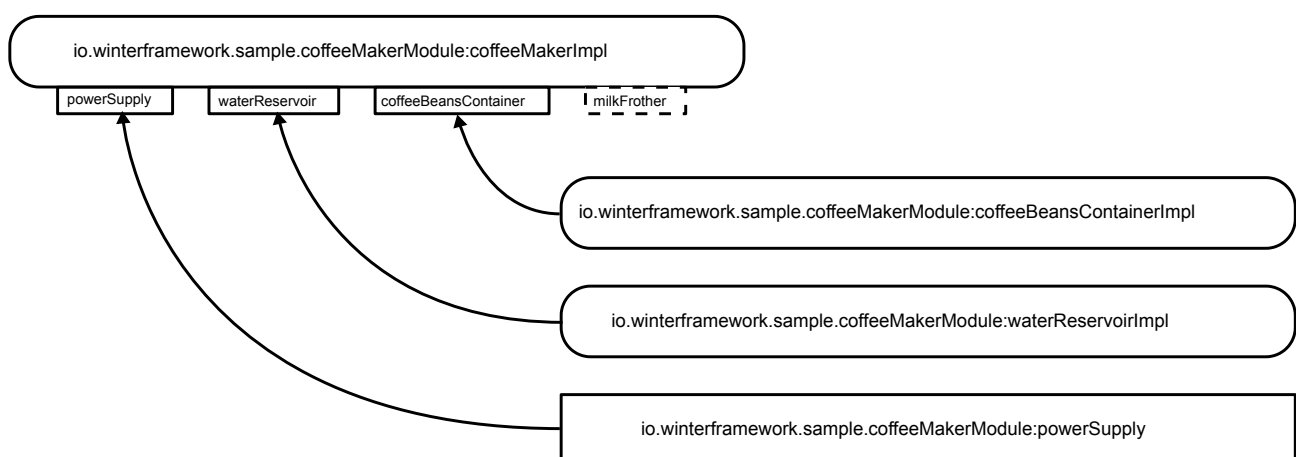
In the coffee maker module we have the following beans:

- `io.winterframework.sample.coffeeMakerModule:coffeeMakerImpl`
- `io.winterframework.sample.coffeeMakerModule:waterReservoirImpl`
- `io.winterframework.sample.coffeeMakerModule:coffeeBeansContainerImpl`
- `io.winterframework.sample.coffeeMakerModule:powerSupply` (socket bean)

The `io.winterframework.sample.coffeeMakerModule:coffeeMakerImpl` bean defines the following sockets:

- `io.winterframework.sample.coffeeMakerModule:coffeeMakerImpl:powerSupply`
- `io.winterframework.sample.coffeeMakerModule:coffeeMakerImpl:waterReservoir`
- `io.winterframework.sample.coffeeMakerModule:coffeeMakerImpl:coffeeBeansContainer`
- `io.winterframework.sample.coffeeMakerModule:coffeeMakerImpl:milkFrother` (optional)

This configuration results in the following graph of beans:



The module is viable since all required beans sockets are resolved and the graph of beans is a directed acyclic graph. The Winter compiler can then generate a module class containing the logic to instantiate the beans in the right order and the dependency injection logic. When an instance of the `io.winterframework.sample.coffeeMakerModule` module is started, the `waterReservoirImpl` bean and the `coffeeBeansContainerImpl` are instantiated first then the `coffeeMakerImpl` bean is instantiated next using previously created instances and the `powerSupply` instance injected when the module was created.

In case one or more bean sockets cannot be resolved, the Winter compiler outputs specific compilation errors for each one of them. When this happens, you must either define module beans or socket beans inside the module matching the unresolved sockets in order for the module to compile.

You'll learn in the [Modular application](#) section that there is another way to provide beans in a module by *composing* another Winter module inside your module.

Explicit wiring

It is not possible for the Winter compiler to automatically wire module beans when more than one bean matching a socket exists in the module. In that case, the Winter compiler outputs specific compilation errors on bean sockets presenting such conflicts. In order for the module to compile, these conflicts must be explicitly resolved.

Let's assume we actually have two beans of type `WaterReservoir` in the coffee maker module: `smallWaterReservoir` and `bigWaterReservoir`, the `coffeeMakerImpl` requires only one `WaterReservoir` since the `waterReservoir` socket is a single socket, we clearly have a conflict that the Winter compiler cannot resolve on its own because it cannot decide for you which water reservoir bean is best suited. So you have to explicitly tell the Winter compiler what to do using a `@Wire` annotation on the module definition:

```
@Module
@Wire(beans="smallWaterReservoir", into="coffeeMakerImpl:waterReservoir")
Module io.winterframework.sample.coffeeMakerModule {
    ...
}
```

In the `@Wire` annotation the `beans` attribute is used to specify which beans must be wired into the socket specified in the `into` attribute.

The `beans` attribute is an array of bean qualified names of the form `([MODULE]:)?[BEAN]`. If the module name is omitted, the compiler will look for beans in the current module. When defining a wire for a single socket, only one bean qualified name is expected.

The `into` attribute is a bean socket qualified name of the form `([MODULE]|([MODULE]:)?[BEAN]):[SOCKET_NAME]`. When specifying a wire on a bean socket name which is necessarily defined in a bean in the current module, the module name can be omitted.

The module name is in fact only necessary when specifying a wire on a socket bean of a module composed in a [composite module](#).

Obviously, multiple `@Wire` annotations can be specified on a module definition. If a specified bean does not exist, if the specified socket does not exist, if the specified beans does not match the specified socket or if multiple beans were specified for a single socket, the Winter compiler will raise compilation errors.

Resolving conflicts is one way of using explicit wiring, but in the case of a multiple socket, you can also use a wire to explicitly select which beans you want to inject using the `@Wire` annotation. For instance, let's say we have a module with four beans of type `SomeType`: `beanA`, `beanB`, `beanC`, `beanD` and another bean which defines a multiple socket of the same type (eg. `List<SomeType>`), if you do nothing, by default the Winter compiler will automatically wire all four into the multiple socket, if you want to inject only `beanA` and `beanB` you can specify the following on the module definition:

```
@Module
@Wire(beans={"beanA", "beanB"}, into="someBean:multipleSomeType")
Module someModule {
    ...
}
```

It's interesting to see that it is not on the socket that the conflict is resolved but on the module that actually created that conflict. This is quite different than other DI frameworks that use qualifiers specified on the conflicting injection point. With these approaches, in order to properly separate the concerns a bean should not know the name of the actual bean that will be injected, as a result it is up to the bean to define the qualifiers and up to the other beans to be named or aliased after these qualifiers but this means the bean still know that a conflict exist otherwise it wouldn't need to specify any qualifier. The Winter framework eliminates this issue to enforce proper separation of concerns.

Selector

Beans are always wired to sockets based on their types, selectors provide another level of filtering. They are used to specify what compile time properties a bean type must have to be wired to a particular socket.

Selectors are annotations annotated with `@Selector` that can be specified on both bean sockets and socket beans. The framework currently supports the `@AnnotationSelector` that lets you filter beans based on a particular annotation.

Let's say, you finally decided to provide a milk frother to the coffee maker which is unfortunately only compatible with milk frothers of a particular brand. To do so, you can define a `@SuperSteam` annotation for the brand and tell the Winter compiler to make sure the milk forther wired to the coffee maker is annotated with it.

```
public @interface SuperSteam {}
```

```

@Bean
public class CoffeMakerImpl implements CoffeMaker {

    ...
    public void setMilkFrother(@AnnotationSelector(SuperSteam.class) MilkFrother milkFrother) {
        ...
    }
}

```

If no bean of type `MilkFrother` annotated with `@SuperSteam` exists, a compilation error is raised.

It's important to understand here that the Winter compiler considers the declared type of a bean which is not necessarily the actual type of the runtime instance. This is especially true when defining a [provided type](#) in a bean class, the selector annotation must then be specified on the provided type and not the actual bean class.

Modular application

Modularity is at the heart of the Winter framework, it has been built on the idea that flexibility, maintainability and stability, especially on large and complex applications can only be achieved through a proper modularization and strict [separation of concerns](#).

So far, we explored how to define and compose beans inside a module to implement a wider component or a standalone application but the Winter framework also allows the composition of modules to create even more complex components and applications.

Composite module

A **composite module** is literally a module composed of multiple Winter modules. Concretely, all public beans exposed in a component module are considered for dependency injection in the composite module. In the same way, socket beans defined in a component module are resolved with the beans available in the composite module.

By default, any Winter module required in the module descriptor of a Winter module are composed by the Winter compiler inside the module class. Component modules public beans are encapsulated in the composite module class and then only accessible from within that module. At runtime, component modules are instantiated and started along with the composite module which wires their public beans into the module's beans sockets or into other component modules socket beans.

Let's assume module `io.winterframework.sample.milkFrotherModule` provides a `MilkFrother` bean compatible with the coffee maker. You can simply declare it as required in the module descriptor of the `io.winterframework.sample.coffeeMakerModule` module to get the milk frother module created and started along with the coffee maker module and eventually wire the milk frother into the coffee maker.

```

@Module
Module io.winterframework.sample.coffeeMakerModule {
    ...
    requires io.winterframework.sample.milkFrotherModule;
    ...
}

```

The Winter compiler will find out that the milk frother module provides a bean matching coffee maker optional milk frother socket and do the wiring in the module class.

In some situations, you might want to explicitly include or exclude required modules from the module composition, you can do this using **includes** and **excludes** attributes in the **@Module** annotation. This is useful when you just want to use types from another module without instantiating it.

```

@Module(includes={"moduleA", "moduleB"})
Module someModule {
    ...
    requires moduleA;
    requires moduleB;
    requires moduleC; // moduleC will be ignored by the Winter compiler
    ...
}

```

In order for the module to compile, all required socket beans defined in component modules must be resolved. They can be resolved with any beans available in the composite module including beans, socket beans or any public beans provided in other component modules.

Explicit wiring can be used as described before using fully qualified names for component modules public beans or socket beans.

```

@Module
@Wire(beans="moduleA:bean1", into="someBean:socket") // Explicitly wire bean 'bean1' of
component module 'moduleA' into bean socket 'socket' in bean 'someBean' of module 'someModule'
@Wire(beans="moduleB:bean2", into="moduleC:socketBean") // Explicitly wire bean 'bean2' of
component module 'moduleB' into socket bean 'socketBean' of module 'moduleC'
Module someModule {
    ...
    requires moduleA;
    requires moduleB;
    requires moduleC;
    ...
}

```

Module composition offers greater flexibility when using or designing modules. A typical Winter application module would be a simple composition of multiple Winter modules implementing different aspects. Multiple modules inside an application can depend on the same module but with different instances which limits the possibility of collisions and increases reusability. Indeed when developing a module you don't have to worry about the context in which it will be used or executed, you can focus on the feature it provides, external dependencies can be provided internally through module composition or externally through socket beans.

Provided type

By default, the type of a bean is given by the class defining the bean, for a module bean it is the annotated class and for a wrapper bean it is the formal parameter specified in the `Supplier<E>` interface.

This basically means that the type of a public bean must be accessible from outside the module, its package must then be exported in the module descriptor. However, you might, and probably will, need to hide bean implementations and only expose public API types.

You can control which type is actually provided by a bean using the `@Provide` annotation. A bean can only provide one type.

Let's see how it works for the `io.winterframework.sample.coffeeMakerModule:coffeeMakerImpl` bean:

```
@Bean
public class CoffeeMakerImpl implements @Provide CoffeeMaker {
    ...
}
```

The `CoffeeMakerImpl` class can implement several types but it will be exposed as a `CoffeeMaker` in the module class.

The `@Provide` annotation is only useful on module bean, for w beans the implementation type is already hidden in the `Supplier#get()` method and the provided type is the formal parameter specified in the `Supplier<E>` interface.

The provided type is only considered outside the module when used in a composite module or in an application. Inside the module, the actual bean type is always used for dependency injection.

Hiding implementation and only expose public API is very convenient when you developed a component module and it is a best practice in general if you want to enforce modularity inside an application. Most of the time modules should always depend on public API so from a dependency injection perspective it doesn't really matters whether a module expose implementation classes but you can't guarantee that nobody will ever create a dependency on an implementation class if that class is accessible which would be quite bad for maintainability. Being able to control the types actually exposed in a module enforces a proper isolation.

Particular care must be taken when using [selectors](#) in a composite module, the type of component modules beans considered by the Winter compiler will be the provided types, so if you want to specify properties matching selectors, you have to specify them on the provided types and not the actual beans types.

5

Winter Modules

Motivation

Built on top of the [Winter core IoC/DI framework](#), Winter modules suite aimed to provide a complete set of features to develop high end production-grade applications.

The advent of cloud computing and highly distributed architecture based on microservices has changed the way applications should be conceived, maintained, executed and operated. While it was perfectly fine to have application started in couple of seconds or even minutes some years ago with long release cycles, today's application must be highly efficient, agile in terms of development and deployment and start in a heart beat.

The Winter framework was created to reduce framework overhead at runtime to the minimum, allowing to create applications that start in milliseconds. Winter modules extend this approach to provide functionalities with low footprint, relying on the compiler when it makes sense to generate human-readable code for easy maintenance and improved performance.

An agile application is naturally modular which is the essence of the Winter framework, but it must also be highly configurable and customizable in many ways using configuration data distributed in various data stores and that greatly depend on the context such as an execution environment: test, production..., a location: US, Europe, Asia..., a particular customer, a particular user... Advanced configuration capabilities are then essential to build modern applications.

Traditional application servers and frameworks used to be based on inefficient threading models that didn't make fair use of hardware resources which make them bad cloud citizens. Winter applications are one hundred percent reactive making maximum use of the allocated resources.

The primary goals can be summarized as follows:

- provide a complete set of common features to build any kind of applications
- maintain a high level of performance...
- ...but always choose modularity and maintainability over performance to favor agility
- be explicit and consistent, there's nothing worse than ambiguity and disparateness, the *you have to know*s must be minimal and logical.
- provide advanced configuration and customization features

Prerequisites

Before we can dig into the various modules provided in the framework, it is important to understand how to setup a modular Winter project, so please have a look at the [Winter distribution documentation](#) which describes in details how to create, build, run, package and distribute a modular Winter component or application.

Winter modules are built on top of the Winter core IoC/DI framework, please refer to the [Winter core documentation](#) to understand how IoC/DI is working in the framework.

The framework is fully reactive thanks to [Project Reactor Core library](#), it is strongly recommended to also look at [the reference documentation](#).

Overview

The basic Winter application is a Winter module composing the *boot* module which provides common services. Other Winter modules can then be added by defining the corresponding dependencies in the module descriptor.

```
@io.winterframework.core.annotation.Module
module io.winterframework.example.app {
    requires io.winterframework.mod.boot;
    // Other modules...
}
```

Declaring a dependency to the *boot* module automatically includes core IoC/DI modules as well as *base* module, *configuration* module and reactive framework dependencies.

A basic application can then be created as follows:

```
import io.winterframework.core.v1.Application;

public class Main {

    public static void main(String[] args) {
        Application.with(new App.Builder()).run();
    }
}
```

Winter modules are fully integrated which means they have been designed to work together in a Winter component or application but this doesn't mean it's not possible to embed them independently in any kind of application following the agile principle. For instance, the *configuration* module, can be easily used in any application with limited dependency overhead. More generally, a Winter module can be created and started very easily in pure Java thanks to the Winter core IoC/DI framework.

For instance, an application can embed a HTTP server as follows:

```
Boot boot = new Boot.Builder().build();
boot.start();

Server httpServer = new Server.Builder(boot.netService(), boot.resourceService())
    .setHttpServerConfiguration(HttpServerConfigurationLoader.load(conf -> conf.server_port(8080)))
    .setRootHandler(
        exchange -> exchange
            .response()
            .body()
            .raw()
            .value(Unpooled.unreleasableBuffer(Unpooled.copiedBuffer("Hello, world!",
Charsets.DEFAULT)))
    )
    .build();

httpServer.start();
...
httpServer.stop();
boot.stop();
```

Note that as for any Winter module, dependencies are clearly specified and must be provided when creating a module, in the previous example the HTTP server requires a **NetService** and a **ResourceService** which are normally provided by the boot module but custom implementations can be provided. It is also possible to create a Winter module composing the *boot* and *http-server* modules to let the framework deal with dependency injection.

Base

The Winter *base* module defines the foundational APIs used across all modules, it can be seen as an extension to the *java.base* module.

In order to use the Winter *base* module, we need to declare a dependency in the module descriptor:

```
module io.winterframework.example.app {
    requires io.winterframework.mod.base;
    ...
}
```

The *base* module declares transitive dependencies to reactive APIs which don't need to be re-declared.

We also need to declare that dependency in the build descriptor:

Using Maven:

```

<project>
  <dependencies>
    <dependency>
      <groupId>io.winterframework.mod</groupId>
      <artifactId>winter-base</artifactId>
    </dependency>
  </dependencies>
</project>

```

Using Gradle:

```

...
compile 'io.winterframework.mod:winter-base:1.0.0'
...

```

The *base* module is usually provided as a transitive dependency by other modules, mainly the *boot* module, so defining a direct dependency is usually not necessary at least for an application module.

Converter API

The converter API provides interfaces and classes for building converters, decoders or encoders which are basically used to decode/encode objects of a given type from/to objects of another type.

Basic converter

The **Converter** interface defines a basic converter. It simply extends **Decoder** and **Encoder** interfaces which defines respectively the basic decoder and the basic encoder.

A basic decoder is used to decode an object of a source type to an object of a target type. For instance, we can create a simple string to integer decoder as follows:

```

public class StringToIntegerDecoder {

    @Override
    public <T extends Integer> T decode(String value, Class<T> type) throws ConverterException {
        return (T)Integer.valueOf(value);
    }

    @Override
    public <T extends Integer> T decode(String value, Type type) throws ConverterException {
        return (T)Integer.valueOf(value);
    }
}
Decoder<String, Integer>

```

A basic encoder is used to encode an object of a source type to an object of a target type. For instance, we can create a simple integer to string encoder as follows:


```

public class IntegerToStringEncoder implements Encoder<Integer, String> {

    @Override
    public <T extends Integer> String encode(T value) throws ConverterException {
        return value.toString();
    }

    @Override
    public <T extends Integer> String encode(T value, Class<T> type) throws ConverterException {
        return value.toString();
    }

    @Override
    public <T extends Integer> String encode(T value, Type type) throws ConverterException {
        return value.toString();
    }
}

```

A string to integer converter can then be created by combining both implementations.

The previous example while not very representative illustrates the basic decoder and encoder API, you should now wonder how to use this properly in an application and what is the fundamental difference between a decoder and an encoder, the answer actually lies in the names. A decoder is meant to *decode* data formatted in a particular way into a representation that can be used in an application whereas an encoder is meant to *encode* an object in an application into data formatted in a particular way. From there, we understand that a converter can be used to read or write raw data (JSON data in an array of bytes for instance) to or from actual usable representations in the form of Java objects but it can also be used as an object mapper to convert from one representation to another (domain object to data transfer object for instance).

A more realistic example would then be a JSON string to object converter:

```

public class JsonToObjectConverter implements Converter<String, Object> {

    private ObjectMapper mapper = new ObjectMapper();

    @Override
    public <T> T decode(String value, Class<T> type) throws ConverterException {
        try {
            return this.mapper.readValue(value, type);
        }
        catch (JsonProcessingException e) {
            throw new ConverterException(e);
        }
    }

    @Override
    public <T> T decode(String value, Type type) throws ConverterException {
        ...
    }

    @Override
    public <T> String encode(T value) throws ConverterException {
        try {
            return this.mapper.writeValueAsString(value);
        }
        catch (JsonProcessingException e) {
            throw new ConverterException(e);
        }
    }

    @Override
    public <T> String encode(T value, Class<T> type) throws ConverterException {
        ...
    }

    @Override
    public <T> String encode(T value, Type type) throws ConverterException {
        ...
    }
}

```

The API provides other interfaces to create converters, decoders and encoders with more capabilities.

Splittable decoder and Joinable encoder

A **SplittableDecoder** is a particular decoder which allows to decode an object of a source type into multiple objects of a target type. It specifies methods to decode one source instance into an array, a list or a set of target instances.

In the same way, a **JoinableEncoder** is a particular encoder which allows to encode multiple objects of a source type into one single object of a target type. It specifies methods to encode an array, a list or a set of source instances into a single target instance.

The **StringConverter** is a typical implementation that can decode or encode multiple parameters values.

```
StringConverter converter = new StringConverter();

// List.of(1, 2, 3)
List<Integer> l = converter.decodeToList("1,2,3", Integer.class);
// "1,2,3"
String s = converter.encodeList(List.of(1, 2, 3));
```

Primitive decoder and encoder

A **PrimitiveDecoder** is fundamentally an object decoder which provides bindings to decode an object of a source type into an object of primitive (boolean, integer...) or common type (string, date, URI...).

In the same way, a **PrimitiveEncoder** is fundamentally an object encoder which provides bindings to encode an object of a primitive or common type to an object of a target type.

The **StringConverter** which is meant to convert parameter values is again a typical use case of primitive decoder and encoder.

```
StringConverter converter = new StringConverter();

// 123l
long l = converter.decodeLong("123");
// ISO-8601 date: "yyyy-MM-dd"
String s = converter.encode(LocalDate.now());
```

The **SplittablePrimitiveDecoder** and **JoinablePrimitiveEncoder** are primitive decoder and encoder that respectively extends **SplittableDecoder** and **JoinableEncoder**.

Object converter

An **ObjectConverter** is a convenient interface for building **Object** converters. It extends **Converter**, **SplittablePrimitiveDecoder** and **JoinablePrimitiveEncoder**.

Reactive converter

A **ReactiveConverter** is a particular converter which extends **ReactiveDecoder** and **ReactiveEncoder** for building reactive converters which are particularly useful to convert data from non-blocking I/O channels.

The **ReactiveDecoder** interface defines methods to decode one or many objects of a target type from a stream of objects of a source type. In the same way, the **ReactiveEncoder** interface defines methods to encode one or many objects of a source type into a stream of objects of target type.

The **ByteBufConverter** is a typical use case, it is meant to convert data from non-blocking channels like the request or response payloads in a network server or client, or the content of a resource read asynchronously.

```

ByteBufConverter converter = new ByteBufConverter(new StringConverter());

Publisher<ByteBuf> dataStream = ... // comes from a request or resource

// On subscription, chunk of data accumulates until a complete response can be emitted
Mono<ZonedDateTime> dateTimeMono = converter.decodeOne(dataStream, ZonedDateTime.class);

// On subscription, a stream of integer is mapped to a publisher of ByteBuf
Publisher<ByteBuf> integerStream = converter.encodeMany(Flux.just(1,2,3,4));

```

Media type converter

A **MediaTypeConverter** is a particular kind of object converter which supports a specific format specified as a [media type](#) and converts object from/to raw data in the supported format. A typical example would be a JSON media type converter used to decode/encode raw JSON data.

The *web* module relies on such converters to respectively decode and encode HTTP request and HTTP response payloads based on the content type specified in the message headers.

Composite converter

A **CompositeConverter** is an extensible object converter based on a **CompositeDecoder** and a **CompositeEncoder** which themselves rely on multiple **CompoundDecoder** and **CompoundEncoder** to extend or override respectively the decoding and encoding capabilities of the converter. In practical terms, it is possible to make a converter able to decode or encode any type of object by providing ad hoc compound decoders and encoders.

The **StringCompositeConverter** is a composite converter implementation which uses a default **StringConverter** to convert primitive and common types of objects, it can be extended to convert other types of object.

For instance, let's consider the following **Message** class:

```

public static class Message {

    private String message;

    // constructor, getter, setter
    ...
}

```

We can create specific compound decoder and encoder to respectively decode and encode a **Message** from/to a string as follows:

```

public static class MessageDecoder implements CompoundDecoder<String, Message> {

    @SuppressWarnings("unchecked")
    @Override
    public <T extends Message> T decode(String value, Class<T> type) throws ConverterException {
        return (T) new Message(value);
    }

    @SuppressWarnings("unchecked")
    @Override
    public <T extends Message> T decode(String value, Type type) throws ConverterException {
        return (T) new Message(value);
    }

    @Override
    public <T extends Message> boolean canDecode(Class<T> type) {
        return Message.class.equals(type);
    }

    @Override
    public boolean canDecode(Type type) {
        return Message.class.equals(type);
    }
}

public static class MessageEncoder implements CompoundEncoder<Message, String> {

    @Override
    public <T extends Message> String encode(T value) throws ConverterException {
        return value.getMessage();
    }

    @Override
    public <T extends Message> String encode(T value, Class<T> type) throws ConverterException {
        return value.getMessage();
    }

    @Override
    public <T extends Message> String encode(T value, Type type) throws ConverterException {
        return value.getMessage();
    }

    @Override
    public <T extends Message> boolean canEncode(Class<T> type) {
        return Message.class.equals(type);
    }

    @Override
    public boolean canEncode(Type type) {
        return Message.class.equals(type);
    }
}

```

And inject them into a string composite converter which can then decode/encode **Message** object:

```
CompoundDecoder<String, Message> messageDecoder = new MessageDecoder();
CompoundEncoder<Message, String> messageEncoder = new MessageEncoder();

StringCompositeConverter converter = new StringCompositeConverter();
converter.setDecoders(List.of(messageDecoder));
converter.setEncoders(List.of(messageEncoder));

Message decodedMessage = converter.decode("this is an encoded message", Message.class);
String encodedMessage = converter.encode(new Message("this is a decoded message"));
```

Net API

The Net API provides interfaces and classes to manipulate basic network elements such as URIs or to create basic network clients and servers.

URIs

A URI follows the standard defined by [RFC 3986](#), it is mostly used to identify resources such as file or more specifically a route in a Web server. The JDK provides a standard implementation which is not close to what is required by the *web* module to name just one.

The `URIs` utility class is the main entry point for working on URIs in any ways imaginable. It defines methods to create a blank URI or a URI based on a given path or URI. These methods return a `URIBuilder` instance which is then used to build a URI, a path, a query string or a URI pattern.

A simple URI can then be created as follows:

```
// http://localhost:8080/path/to/resource?parameter=value
URI uri = URIs.uri()
    .scheme("http")
    .host("localhost")
    .port(8080)
    .path("/path/to/resource")
    .queryParameter("parameter", "value")
    .build();
```

or from an existing URI as follows:

```
// https://test-server/path/to/resource
URI uri = URIs.uri(URI.create("http://localhost:8080/path/to?parameter=value"))
    .scheme("https")
    .host("test-server")
    .port(null)
    .segment("resource")
    .clearQuery()
    .build();
```

A URI can be normalized by enabling the `URIs.Option.NORMALIZED` option:

```
// path/to/other
URI uri = URIs.uri("path/to/resource", URIs.Option.NORMALIZED)
    .segment("..")
    .segment("other")
    .build();
```

A parameterized URI can be created by enabling the `URIs.Option#PARAMETERIZED` option and specifying parameters of the form `{[<name>][:<pattern>]}` in the components of the URI. This allows to create URI templates that can be used to generate URIs from a set of parameters.

```
URIBuilder uriTemplate = URIs.uri(URIs.Option.PARAMETERIZED)
    .scheme("{scheme}")
    .host("{host}")
    .path("/path/to/resource")
    .segment("{id}")
    .queryParameter("format", "{format}");

// http://localhost/path/to/resource/1?format=text
URI uri1 = uriTemplate.build("http", "localhost", "1", "text");

// https://production/path/to/resource/32?format=json
URI uri2 = uriTemplate.build("https", "production", "32", "json");
```

The `URIBuilder` also defines methods to create string representations of the whole URI, the path component or the query component.

```
URIBuilder uriBuilder = URIs.uri()
    .scheme("http")
    .host("localhost")
    .port(8080)
    .path("/path/to/resource")
    .queryParameter("parameter", "value");

// http://localhost:8080/path/to/resource?parameter=value
String uri = uriBuilder.buildString();

// path/to/resource
String path = uriBuilder.buildPath();

// parameter=value
String query = uriBuilder.buildQuery();
```

It can also create `URIPattern` to match a given input against the pattern specified by the URI while extracting parameter values when the URI is parameterized.

```
URIPattern uriPattern = URIs.uri(URIs.Option.PARAMETERIZED)
    .scheme("{scheme}")
    .host("{host}")
    .path("/path/to/resource")
    .segment("{id}")
    .queryParameter("format", "{format}")
    .buildPattern();

URIMatcher matcher = uriPattern.matcher("http://localhost:8080/path/to/resource/1?format=text");
if(matcher.matches()) {
    // scheme=http, host=localhost, id=1, format=text
    Map<String, String> parameters = matcher.getParameters();
    ...
}
```

Network service

The `NetService` interface specifies a service for building optimized network clients and servers based on Netty. The *base* module doesn't provide any implementation, a base implementation is provided in the *boot* module.

This service especially defines methods to obtain `EventLoopGroup` instances backed by a root event loop group in order to reuse event loops across different network servers or clients running in the same application.

It also defines methods to create basic network client and server bootstraps.

Reflection API

The reflection API provides classes and interfaces for building `java.lang.reflect.Type` instances in order to represent parameterized types at runtime which is otherwise not possible due to type erasure. Such `Type` instances are used when decoding data into objects of parameterized types.

The `Types` class is the main entry point for building any kind of Java types.

```
// java.util.List<? extends java.lang.Comparable<java.lang.String>>
Type type = Types.type(List.class)
    .wildcardType()
    .upperBoundType(Comparable.class)
    .type(String.class).and()
    .and()
    .build();
```

The reflection API is particularly useful to specify a parameterized type to an [object converter](#). For instance, let's imagine we have a `ByteBuf` we want to decode to a `List<String>`, we can do:

```
ByteBuf input = ...;
ObjectConverter<ByteBuf> converter = ...;

Type listOfStringType = Types.type(List.class)
    .type(String.class).and()
    .build();
List<String> decode = converter.<List<String>>decode(input, listOfStringType);
```

Resource API

The resource API provides classes and interfaces for accessing resources of different kinds and locations (file, zip, jar, classpath, module...) in a consistent way using a unique `Resource` interface.

A resource can be created directly using the implementation corresponding to the kind of resource. For instance, in order to access a resource on the class path, you need to choose the `ClasspathResource` implementation:

```
ClasspathResource resource = new ClasspathResource(URI.create("classpath:/path/to/resource"));
```


A resource is identified by a URI whose scheme specifies the kind of resources. The *base* module provides several implementations with a corresponding scheme.

Type	URI	Implementation
file	file:/path/to/resource	FileResource
zip	zip:/path/to/zip!/path/to/resource	ZipResource
jar	jar:/path/to/jar!/path/to/resource	JarResource
url	http https ftp://host/path/to/resource	URLResource
classpath	classpath:/path/to/resource	ClasspathResource
module	module://[MODULE_NAME]/path/to/resource	ModuleResource

The `ResourceService` interface specifies a service which provides a unified access to resources based only on the resource URI. The *base* module doesn't provide any implementation, a base implementation is provided in the *boot* module.

A typical use case is to get a resource from a URI without knowing the actual kind of the resource.

```
ResourceService resourceService = ...
```

```
Resource resource = resourceService.getResource(URI.create("classpath:/path/to/resource"));
```

The resource service can also be used to list resources at a given location. Nonetheless this actually depends on the implementation and the kind of resource, although it is clearly possible to list resources from a file location, it might not be supported to list resources from a class path or URL location.

The *boot* module [implementation](#) supports for instance the listing of resources that match a specific path pattern:

```
ResourceService resourceService = ...
```

```
Stream<Resource> resources =  
resourceService.getResources(URI.create("file:/path/to/resources/**/*"));
```

The `MediaTypeService` interface specifies a service used to determine the media type of a resource based on its extension, name, path or URI. As for the resource service, a base implementation is provided in the *boot* module.

```
MediaTypeService mediaTypeService = ...
```

```
// image/png  
String mediaType = mediaTypeService.getForExtension("png");
```

Boot

The Winter *boot* module provides basic services to applications including several base implementation for interfaces defined in the *base* module.

The *Winter boot* module is the basic building block for any application and as such it must be the first module to declare in an application module descriptor.

```
@io.winterframework.core.annotation.Module
module io.winterframework.example.app {
    requires io.winterframework.mod.boot;
    // Other modules...
}
```

The *boot* module declares transitive dependencies to the core IoC/DI modules as well as *base* and *configuration* modules. They don't need to be re-declared.

This dependency must also be declared in the build descriptor:

Using Maven:

```
<project>
  <dependencies>
    <dependency>
      <groupId>io.winterframework.mod</groupId>
      <artifactId>winter-boot</artifactId>
    </dependency>
  </dependencies>
</project>
```

Using Gradle:

```
...
compile 'io.winterframework.mod:winter-boot:1.0.0'
...
```

Configuration

The *boot* module defines specific configuration for the services it exposes, they can be specified when starting the module to override default values.

For instance the [NetConfiguration](#) is used to configure the net service.

Please refer to the [API documentation](#) to have an exhaustive description of the different configuration properties.

Net service

The module provides a base [NetService](#) implementation exposed as a bean for building network applications based on Netty.

Media type service

The module provides a base [MediaTypeService](#) implementation based on the JDK (see [Files.probeContentType\(Path\)](#)) and exposed as an overridable bean allowing custom implementations to be provided.

Resource service

The module provides a base `ResourceService` implementation exposed as a bean for accessing resources.

This base implementation supports the following schemes: `file`, `zip`, `jar`, `classpath`, `module`, `http`, `https` and `ftp` and it allows to list resources for `file`, `zip` and `jar` schemes.

When supported, resources are listed from a base URI specifying a path pattern using the following rules:

- `?` matches one character
- `*` matches zero or more characters
- `**` matches zero or more directories in a path

For instance:

```
ResourceService resourceService = ...

// Return: '/base/test1/a', '/base/test1/a/b', '/base/test2/c'...
Stream<Resource> resources = resourceService.getResources(URI.create("file:/base/test?/**/*"));
```

It is also possible to resolve all resources with a specific name defined in all application modules by specifying `'*'` instead of the module name in a module URI:

```
ResourceService resourceService = ...

// all resources named '/path/to/resource' in all application modules
Stream<Resource> resources =
resourceService.getResources(URI.create("module://*/path/to/resource"));
```

This service can be extended by injecting custom `ResourceProvider` providing resources for a custom URI scheme. For instance, if we create a custom `Resource` and corresponding `ResourceProvider` implementations mapped to URI scheme `custom`, we can extend the resource service so it can create such custom resources.

```
Boot boot = new Base.Boot()
    .setResourceProviders(List.of(new CustomResourceProvider()))
    .build();

boot.start();

Resource customResource = boot.resourceService().get(URI.create("custom:..."));
...

boot.stop();
```

Converters

The module exposes various `Converter` implementations used across an application to convert parameter values or message payloads.

This includes the following also exposed as beans:

- a parameter converter for converting strings from/to objects, this converter can be extended by injecting specific compound decoders and encoders in the module as described in the [composite converter documentation](#).
- a JSON `ByteBuf` converter for converting raw JSON data in `ByteBuf` from/to objects in the application.
- an `application/json` media type converter for converting message payloads from/to JSON.
- an `application/x-ndjson` media type converter for converting message payloads from/to [Newline Delimited JSON](#)
- a `text/plain` media type converter for converting message payloads from/to plain text.

Worker pool

A Winter application must be fully reactive, most of the processing is performed in non-blocking I/O threads but sometimes blocking operations might be needed, in such cases, the worker thread pool should be used to execute these blocking operations without impacting the I/O event loop.

The default worker pool bean is a simple [cached Thread pool](#) which can be overridden by providing a different instance to the `boot` module.

Object mapper

A standard JSON reader/writer based on Jackson `ObjectMapper` is also provided. This instance is used across the application to perform JSON conversion operations, a global configuration can then be applied to that particular instance or it can be overridden when creating the `boot` module.

Configuration

The Winter *configuration* module defines a unified configuration API for building agile highly configurable applications.

Configuration is one of the most important aspect of an application and sadly one of the most neglected. There are very few decent configuration frameworks and most of the time they relate to one part of the issue. It is important to approach configuration by considering it as a whole and not as something that can be solved by a property file here and a database there. Besides, it must be the first issue to tackle during the design phase as it will impact all aspects of the application. For instance, we can imagine an application where configuration is defined in simple property files, a complete configuration would probably be needed for each environment where the application is deployed, maintenance would be probably problematic even more when we know that configuration properties can be added, modified or removed over time.

In its most basic form, a configuration is not more than a set of properties associating a value to a key. It would be naive to think that this is enough to build an agile and customizable application, but in the end, a property should still be considered as the basic building block for configurations.

Now, the first thing to notice is that any part of an application can potentially be configurable, from a server IP address to a color of a button in a user interface, there are multiple forms of configuration with different expectations that must coexist in an application. For instance, some parts of the configuration are purely static and do not change during the operation of an application, this is the case of a bootstrap configuration which mostly relates to the operating environment (eg. a server port). Some other parts, on the other hand, are more dynamic and can change during the operation of an application, this is the case of tenant specific configuration or even user preferences.

Following this, we can see that a configuration greatly depends on the context in which it is loaded. The definition of a configuration, which is basically a list of property names, is dictated by the application, so when the application is running, this definition should be fixed but the context is not. For instance, the bootstrap configuration is different from one operating environment to another, user preferences are not the same from one user to another...

We can summarize this as follows:

- a configuration is a set of configuration properties.
- the configuration of an application is actually composed of multiple configurations with their own specificities.
- the definition of a configuration is bound to the application as a result the only way to change it is to change the application.
- a configuration depends on a particular context which must be considered when setting or getting configuration properties.

The configuration API has been created to address previous points, giving a maximum flexibility to precisely design how an application should be configured.

In order to use the *Winter configuration* module, we need to declare a dependency in the module descriptor:

```
module io.winterframework.example.app {  
    ...  
    requires io.winterframework.mod.configuration;  
    ...  
}
```

And also declare that dependency in the build descriptor:

Using Maven:

```
<project>  
  <dependencies>  
    <dependency>  
      <groupId>io.winterframework.mod</groupId>  
      <artifactId>winter-configuration</artifactId>  
    </dependency>  
  </dependencies>  
</project>
```

Using Gradle:

```
...  
compile 'io.winterframework.mod:winter-configuration:1.0.0'  
...
```

Configuration source

A configuration source can be any data store that holds configuration data, the API abstracts configuration data sources to provide a unified access to configuration data through the `ConfigurationSource` interface. Specific implementations should be considered depending on the type of configuration: a bootstrap configuration is most likely to be static and stored in configuration files or environment variables whereas a tenant specific configuration is most likely to be stored in a distributed data store. However this is not a universal rule, depending on the needs we can very well consider any kind of configuration source for any kind of configuration. The configuration source abstracts these concerns from the the rest of the application.

The `ConfigurationSource` is the main entry point for accessing configuration properties, it shall be used every time there's a need to retrieve configuration properties. It defines only one method for creating a `ConfigurationQuery` instance eventually executed in order to retrieve one or more configuration properties.

For instance, property `server.uri` can be retrieved as follows:

```
ConfigurationSource<?, ?, ?> source = ...
```

```
source.get("server.url")           // 1
  .execute()                       // 2
  .single()                        // 3
  .map(queryResult -> queryResult
    .getResult()                   // 4
    .flatMap(property -> property.asURI()) // 5
    .orElse(URI.create("http://localhost"))) // 6
  )
  .subscribe(serverURI -> ...);    // 7
```

In the preceding example:

1. creates a configuration query to retrieve the `server.url` property
2. executes the query, the API is reactive so nothing will happen until a subscription is actually made on the resulting `Flux` of `ConfigurationQueryResult`
3. transforms the `Flux` to a `Mono` since we expect a single result
4. gets the resulting configuration property, a query result is always returned even if the property does not exist in the source therefore `getResult()` returns an `Optional` that lets you decide what to do if the property is missing
5. converts the property value to URI if present, a property can be defined in a source with a null value which explains why the property value is also an `Optional` and why we need to use `flatMap()`
6. returns the actual value if it exists or the specified default value
7. we subscribe to the `Mono` which actually runs the query in the source and returns the property value or the default value if the property value is null or not defined in the source

This seems to be a lot of steps to simply retrieve one property value, but if you look closely you'll understand that each of them is actually necessary:

- we want to be able to retrieve multiple properties and/or create more complex queries in a batch so `.execute()` is required to mark the end of a batch of queries
- we want to be reactive so `.single().map()` and `subscribe()` are required

- we want to have access to the configuration query key at the origin of a property for troubleshooting as a result the a query result must expose `getQueryKey()` and `getResult()` methods
- we want to be able to convert a property value and provide different behaviors when a property does not exist in a source or when it does exist but with a null value, as a result `.flatMap(property -> property.asURI()).orElse(URI.create("http://localhost"))` is required

As we said earlier, a configuration depends on the context: a given property might have different values when considering different contexts. The configuration API defines a configuration property with a name, a value and a set of parameters specifying the context for which the property is defined. Such configuration property is referred to as a **parameterized configuration property**.

Some configuration source implementations do not support parameterized configuration property, they simply ignore parameters specified in queries and return the value associated to the property name. This is especially the case of environment variables which don't allow to specify property parameters.

In order to retrieve a property in a particular context we can then parameterized the configuration query as follows:

```
source.get("server.url")
    .withParameters("environment", "production", "zone", "us")
    .execute()
...

```

In the preceding example, we query the source for property `server.url` defined for the production environment in zone US. To state the obvious, both the list of parameters and their values can be determined at runtime using actual contextual values. This is what makes parameterized properties so powerful as it is suitable for a wide range of use cases. This is all the more true when using a configuration source which supports some kind of defaulting such as the [Composite Configuration source](#).

Whether the exact or nearest value matching the query is returned by a configuration source is implementation dependent but since the *configuration* module provides the [Composite Configuration source](#) which can wrap any configuration source to add support for defaulting, it is a good practice to implement configuration sources that only support exact matching of a configuration query key (ie. including name and parameters).

As said before the API let's you fluently query multiple properties in a batch and map the results in a configuration object.

```

source
  .get("server.port", "db.url", "db.user", "db.password").withParameters("environment",
"production", "zone", "us")
  .and()
  .get("db.schema").withParameters("environment", "production", "zone", "us", "tenant",
"someCompany")
  .execute()
  .collectMap(queryResult -> queryResult.getQueryKey().getName(), queryResult ->
queryResult.getResult())
  .map(properties -> {
    ApplicationConfiguration config = new ApplicationConfiguration();

    properties.get("server.port").flatMap(property ->
property.asInteger()).ifPresent(config::setServerPort);
    properties.get("db.url").flatMap(property -> property.asURL()).ifPresent(config::setDbURL);
    properties.get("db.user").flatMap(property ->
property.asString()).ifPresent(config::setDbUser);
    String dbPassword = properties.get("db.password").flatMap(property ->
property.asString()).ifPresent(config::setDbPassword);
    String dbSchema = properties.get("db.schema").flatMap(property ->
property.asString()).ifPresent(config::setDbSchema);

    return config;
  })
  .subscribe(config -> {
    ...
  });

```

The beauty of being reactive is that it comes with a lot of cool features such as the ability to re-execute a query or caching the result. A **Flux** or a **Mono** executes on subscriptions, which means we can create a complex query to retrieve a whole configuration, keep the resulting Reactive Streams **Publisher** and subscribe to it when needed. A Reactive Stream publisher can also cache configuration results.

```

Mono<ApplicationConfiguration> configurationLoader = ... // see previous example

```

```

// Query the source on each subscriptions
configurationLoader.subscribe(config -> {
  ...
});

```

```

// Cache the configuration for five minutes
Mono<ApplicationConfiguration> cachedConfigurationLoader =
configurationLoader.cache(Duration.ofMinutes(5));

```

```

// Query the source on first subscriptions, further subscriptions within a window of 5 minutes will
get the cached configuration
cachedConfigurationLoader.subscribe(config -> {
  ...
});

```

Although publisher caching is a cool feature, it might not be ideal for complex caching use cases and more solid solution should be considered.

A configuration source relies on a `SplittablePrimitiveDecoder` to decode property values. Configuration source implementations usually provide a default decoder but it is possible to inject custom decoders to decode particular configuration values. The expected decoder implementation depends on the configuration source implementation but most of the time a string to object decoder is expected.

```
SplittablePrimitiveDecoder<String> customDecoder = ...
```

```
PropertyFileConfigurationSource source = new PropertyFileConfigurationSource(new  
ClasspathResource(URI.create("classpath:/path/to/configuration")), customDecoder)
```

Map configuration source

The map configuration is the most basic configuration source implementation. It exposes configuration properties stored in a map in memory. It doesn't support parameterized properties, regardless of the parameters specified in a query, only the property name is considered when resolving a value.

```
MapConfigurationSource source = new MapConfigurationSource(Map.of("server.url", new  
URL("http://localhost")));  
...
```

This implementation can be used for testing purpose in order to provide a mock configuration source.

System environment configuration source

The system environment configuration source exposes system environment variables as configuration properties. As for the map configuration source, this implementation doesn't support parameterized properties.

```
$ export SERVER_URL=http://localhost
```

```
SystemEnvironmentConfigurationSource source = new SystemEnvironmentConfigurationSource();  
...
```

This implementation can be used to bootstrap an application using system environment variables.

System properties configuration source

The system properties configuration source exposes system properties as configuration properties. As for the two previous implementations, it doesn't support parameterized properties.

```
$ java -Dserver.url=http://localhost ...
```

```
SystemPropertiesConfigurationSource source = new SystemPropertiesConfigurationSource();  
...
```

This implementation can be used to bootstrap an application using system properties.

Command line configuration source

The command line configuration source exposes configuration properties specified as command line arguments of the application. This implementation supports parameterized properties.

Configuration properties must be specified as application arguments using the following syntax:

`--property[parameter_1=value_1...parameter_n=value_n]=value` where property and parameter names are valid Java identifiers and property and parameter values are Java primitives such as integer, boolean, string... A complete description of the syntax can be found in the [API documentation](#).

For instance the following are valid configuration properties specified as command line arguments:

```
$ java ... Main \  
--web.server_port=8080 \  
--web.server_port[profile="ssl"]=8443 \  
--db.url[env="dev"]="jdbc:oracle:thin:@dev.db.server:1521:sid" \  
--db.url[env="prod",zone="eu"]="jdbc:oracle:thin:@prod_eu.db.server:1521:sid" \  
--db.url[env="prod",zone="us"]="jdbc:oracle:thin:@prod_us.db.server:1521:sid"  
  
public static void main(String[] args) {  
    CommandLineConfigurationSource source = new CommandLineConfigurationSource(args);  
    ...  
}  
...
```

.properties file configuration source

The `.properties` file configuration source exposes configuration properties specified in a `.properties` file. This implementation supports parameterized properties.

Configuration properties can be specified in a property file using a syntax similar to the command line configuration source for the property key. Some characters must be escaped with respect to the `.properties` file format. Property values don't need to follow Java's notation for strings since they are considered as strings by design.

```
web.server_port=8080  
web.server_port[profile\="ssl"]=8443  
db.url[env\="dev"]="jdbc:oracle:thin:@dev.db.server:1521:sid"  
db.url[env\="prod",zone\="eu"]="jdbc:oracle:thin:@prod_eu.db.server:1521:sid"  
db.url[env\="prod",zone\="us"]="jdbc:oracle:thin:@prod_us.db.server:1521:sid"  
  
PropertyFileConfigurationSource source = new PropertyFileConfigurationSource(new  
ClasspathResource(URI.create("classpath:/path/to/file")));  
...
```

.cprops file configuration source

The `.cprops` file configuration source exposes configuration properties specified in a `.cprops` file. This implementation supports parameterized properties.

The `.cprops` file format has been introduced to facilitate the definition and reading of parameterized properties. In particular it allows to regroup the definition of properties with common parameters into sections and many more.

For instance:

```
server.port=8080
db.url=jdbc:oracle:thin:@localhost:1521:sid
db.user=user
db.password=password
log.level=ERROR
application.greeting.message=""
=== Welcome! ===

    This is
    a formatted
    message.

=====
"""

[ environment="test" ] {
    db.url=jdbc:oracle:thin:@test:1521:sid
    db.user=user_test
    db.password=password_test
}

[ environment="production" ] {
    db.url=jdbc:oracle:thin:@production:1521:sid
    db.user=user_production
    db.password=password_production

    [ zone="US" ] {
        db.url=jdbc:oracle:thin:@production.us:1521:sid
    }

    [ zone="EU" ] {
        db.url=jdbc:oracle:thin:@production.eu:1521:sid
    }

    [ zone="EU", node="node1" ] {
        log.level=DEBUG
    }
}
```

A complete [JavaCC grammar](#) is available in the source of the configuration module.

```
CPropsFileConfigurationSource source = new CPropsFileConfigurationSource(new
ClasspathResource(URI.create("classpath:/path/to/file")));
...
```

Bootstrap configuration source

The bootstrap configuration source is a [composite configuration source](#) preset with configuration sources typically used when bootstrapping an application.

This implementation resolves configuration properties from the following sources in that order, from the highest priority to the lowest:

- command line
- system properties
- system environment variables
- the `configuration.cprops` file in `./conf/` or `${winter.conf.path}/` directories if one exists (if the first one exists the second one is ignored)

- the `configuration.cprops` file in `${java.home}/conf/` directory if it exists
- the `configuration.cprops` file in the application module if it exists

This source is typically created in a `main` method to load the bootstrap configuration on startup.

```
public class Application {

    public static void main(String[] args) {
        BootstrapConfigurationSource source = new
BootstrapConfigurationSource(Application.class.getModule(), args);

        // Load configuration
        ApplicationConfiguration configuration = ConfigurationLoader
            .withConfiguration(ApplicationConfiguration.class)
            .withSource(source)
            .load()
            .block();

        // Start the application with the configuration
        ...
    }
}
```

Redis configuration source

The [Redis](#) configuration source exposes configuration properties stored in a Redis data store. This implementation supports parameterized properties and it is also configurable which means it can be used to set configuration properties in the data store at runtime.

It also provides a simple but effective versioning system which allows to set multiple properties and activate or revert them atomically. A global revision keeps track of the whole data store but it is also possible to version a particular branch in the tree of properties.

The following example shows how to set configuration properties for the `dev` and `prod` environment and activates them globally or independently.

```
RedisClient redisClient = ...
RedisConfigurationSource source = new RedisConfigurationSource(redisClient);

source
    .set("db.url", "jdbc:oracle:thin:@dev.db.server:1521:sid").withParameters("environment",
"dev").and()
    .set("db.url", "jdbc:oracle:thin:@prod_eu.db.server:1521:sid").withParameters("environment",
"prod", "zone", "eu").and()
    .set("db.url", "jdbc:oracle:thin:@prod_us.db.server:1521:sid").withParameters("environment",
"prod", "zone", "us")
    .execute()
    .blockLast();

// Activate working revision globally
source.activate().block();

// Activate working revision for dev environment and prod environment independently
source.activate("environment", "dev").block();
source.activate("environment", "prod").block();
```

It is also possible to fallback to a particular revision by specifying it in the `activate()` method:

```
// Activate revision 2 globally
source.activate(2).block();
```

This implementation is particularly suitable to load tenant specific configuration in a multi-tenant application, or user preferences... basically any kind of configuration that can and will be dynamically changed at runtime.

Parameterized properties and versioning per branch are two simple yet powerful features but it is important to be picky here otherwise there is a real risk of messing things up. You should thoughtfully decide when a configuration branch can be versioned, for instance the versioned sets of properties must be disjointed (if this is not obvious, think again), this is actually checked in the Redis configuration source and an exception will be thrown if you try to do things like this, basically trying to version the same property twice.

Composite Configuration source

The composite configuration source is a configuration source implementation with two particular features: first it allows to compose multiple configuration sources into one configuration source and then it supports defaulting strategies to determine the best matching value for a given configuration query key.

The property returned for a configuration query key then depends on two factors: the order in which configuration sources were defined in the composite configuration source, from the highest priority to the lowest, and then how close is a property from the configuration query key.

The `CompositeConfigurationSource` resolves a configuration property by querying its sources in sequence from the highest priority to the lowest. It relies on a `CompositeConfigurationStrategy` to determine at each round which queries to execute and retain the best matching property from the results. The best matching property is the property whose key is the closest to the original configuration query key according to a metric implemented in the strategy. The algorithm stops when an exact match is found or when there's no more configuration source to query.

The `DefaultCompositeConfigurationStrategy` defines the default strategy implementation. It determines the best matching property for a given original query by prioritizing query parameters from left to right: the best matching property is the one matching the most continuous parameters from right to left. In practice, if we consider query key `property[p1=v1, ... pn=vn]`, it supersedes key `property[p2=v2, ... pn=vn]` which supersedes key `property[p3=v3, ... pn=vn]`... which supersedes key `property[...]`. As a result, an original query with `n` parameters results in `n+1` queries being actually executed if no property was retained in previous rounds and `n-p` queries if a property with `p` parameters (`p < n`) was retained in previous rounds. The order into which parameters are specified in the original query is then significant: `property[p1=v1, p2=v2] != property[p2=v2, p1=v1]`.

When defining configuration parameters, we should then order them from the most specific to the most general when querying a composite source. For example, the `node` parameter which is more specific than the `zone` parameter should come first then the `zone` parameter which is more specific than the `environment` parameter should come next and finally the `environment` parameter which is the most general should come last.

For instance, we can consider two parameterized configuration sources: `source1` and `source2`.

`source1` holds the following properties:

- `server.url[]=null`
- `server.url[zone="US", environment="production"]="https://prod.us"`
- `server.url[zone="EU"]="https://default.eu"`

`source2` holds the following properties:

- `server.url[]="https://default"`
- `server.url[environment="test"]="https://test"`
- `server.url[environment="production"]="https://prod"`

We can compose them in a composite configuration source as follows:

```
ConfigurationSource<?, ?, ?> source1 = ...
```

```
ConfigurationSource<?, ?, ?> source2 = ...
```

```
CompositeConfigurationSource source = new CompositeConfigurationSource(List.of(source1, source2));
```

```
source // 1
    .get("server.url")
    .withParameters("zone", "US", "environment", "production")
    .execute()
    ...
```

```
source // 2
    .get("server.url")
    .withParameters("environment", "test")
    .execute()
    ...
```

```
source // 3
    .get("server.url")
    .execute()
    ...
```

```
source // 4
    .get("server.url")
    .withParameters("zone", "EU", "environment", "production")
    .execute()...
```

```
source // 5
    .get("server.url")
    .withParameters("environment", "production", "zone", "EU")
    .execute()
    ...
```

In the example above:

1. `server.url[environment="production",zone="US"]` is exactly defined in `source1` => `https://prod.us` defined in `source1` is returned
2. `server.url[environment="test"]` is not defined in `source1` but exactly defined in `source2` => `https://test` defined in `source2` is returned
3. Although `server.url[]` is defined in both `source1` and `source2`, `source1` has the highest priority and therefore => `null` is returned
4. There is no exact match for `server.url[zone="EU", environment="production"]` in both `source1` and `source2`, the priority is given to the parameters from left to right, the property matching `server.url[environment="production"]` shall be returned => `https://prod` defined in `source2` is returned
5. Here we've simply changed the order of the parameters in the previous query, again the priority is given to parameters from left to right, since there is no match for `server.url[environment="production", zone="EU"]`, `server.url[zone="EU"]` is considered => `https://default.eu` defined in `source1` is returned

As you can see, the order into which parameters are specified in a query is significant and different results might be returned.

When considering multiple configuration sources, properties can be defined with the exact same key in two different sources, the source with the highest priority wins. In the last example we've been able to set the value of `server.url[]` to `null` in `source1`, however `null` is itself a value with a different meaning than a missing property, the `unset` value can be used in such situation to *unset* a property defined in a source with a lower priority.

For instance, considering previous example, we could have defined `server.url[]=unset` instead of `server.url[]=null` in `source1`, the query would then have returned an empty query result indicating an undefined property.

Configurable configuration source

A configurable configuration source is a particular configuration source which supports configuration properties updates. The [Redis configuration source](#) is an example of configurable configuration source.

The `ConfigurableConfigurationSource` interface is the main entry point for updating configuration properties, it shall be used every time there's a need to retrieve or set configuration properties.

It extends the `ConfigurationSource` with one method for creating a `ConfigurationUpdate` instance eventually executed in order to set one or more configuration properties in the configuration source.

For instance, a parameterized property `server.port` can be set in a configuration source as follows:

```
ConfigurableConfigurationSource<?, ?, ?, ?, ?, ?> source = null;

source.set("server.port", 8080)
    .withParameters("environment", "production", "zone", "us")
    .execute()
    .single()
    .subscribe(
        updateResult -> {
            try {
                updateResult.check();
                // Update succeeded
                ...
            }
            catch(ConfigurationSourceException e) {
                // Update failed
                ...
            }
        }
    );
```

A configurable configuration source relies on a `JoinablePrimitiveEncoder` to encode property values. Implementations usually provide a default encoder but it is possible to inject custom encoders to encode particular configuration values. The expected encoder implementation depends on the configuration source implementation but most of the time an object to string encoder is expected.

```
RedisClient redisClient = ...
JoinablePrimitiveEncoder<String> customEncoder = ...
SplittablePrimitiveDecoder<String> customDecoder = ...
```

```
RedisConfigurationSource source = new RedisConfigurationSource(redisClient, customEncoder,
    customDecoder)
```

Configuration loader

The API offers a great flexibility but as we've seen it might require some efforts to load a configuration in a usable explicit Java bean. Hopefully, this has been anticipated and the configuration module provides a configuration loader to smoothly load configuration objects in the application.

The `ConfigurationLoader` interface is the main entry point for loading configuration objects from a configuration source. It can be used in two different ways, either dynamically using Java reflection or statically using the Winter compiler.

Dynamic loader

A dynamic loader can be created by invoking static method `ConfigurationLoader#withConfiguration()` which accepts a single `Class` argument specifying the type of the configuration that must be loaded.

A valid configuration type must be an interface defining configuration properties as non-void no-argument methods whose names correspond to the configuration properties to retrieve and to map to the resulting configuration object, default values can be specified in default methods.

For instance the following interface represents a valid configuration type which can be loaded by a configuration loader:

```
public interface AppConfiguration {

    // query property 'server_host'
    String server_host();

    // query property 'server_port'
    default int server_port() {
        return 8080;
    }
}
```

It can be loaded at runtime as follows:

```
ConfigurationSource<?, ?, ?> source = ...
```

```
ConfigurationLoader
    .withConfiguration(AppConfiguration.class)
    .withSource(source)
    .withParameters("environment", "production")
    .load()
    .map(configuration -> startServer(configuration.server_host(), configuration.server_port()))
    .subscribe();
```

In the above example, the configuration source is queried for properties `server_host[environment="production"]` and `server_port[environment="production"]`.

The dynamic loader also supports nested configurations when the return type of a method is an interface representing a valid configuration type.

```
public interface ServerConfiguration {

    // query property 'server_host'
    String server_host();

    // query property 'server_port'
    default int server_port() {
        return 8080;
    }
}

public interface AppConfiguration {

    // Prefix child property names with 'server_configuration'
    ServerConfiguration server_configuration();
}
```

In the above example, the configuration source is queried for properties `server_configuration.server_host[environment="production"]` and `server_configuration.server_port[environment="production"]`.

It is also possible to load a configuration by invoking static method `ConfigurationLoader#withConfigurator()` which allows to load any type of configuration (not only interface) by relying on a configurator and a mapping function.

A configurator defines configuration properties as void single argument methods whose names correspond to the configuration properties to retrieve and inject into a configurator instance using a dynamic configurer `Consumer<Configurator>`. The mapping function is finally applied to that configurer to actually create the resulting configuration object.

For instance, previous example could have been implemented as follows:

```
public class AppConfiguration {

    private String server_host;
    private String server_port = 8080;

    // query property 'server_host'
    public void server_host(String server_host) {
        this.server_host = server_host;
    }

    // query property 'server_port'
    public void server_port(int server_port) {
        this.server_port = server_port;
    }

    public String server_host() {
        return server_host;
    }

    public int server_port() {
        return server_port;
    }
}
```

```
ConfigurationSource<?, ?, ?> source = ...
```

```
ConfigurationLoader
    .withConfigurator(AppConfiguration.class, configurer -> {
        AppConfiguration configuration = new AppConfiguration();
        configurer.apply(configuration);
        return configuration;
    })
    .withSource(source)
    .withParameters("environment", "production")
    .load()
    .map(configuration -> startServer(configuration.server_host(), configuration.server_port()))
    .subscribe();
```

Static loader

Dynamic loading is fine but it relies on Java reflection which induces extra processing at runtime and might cause unexpected runtime errors due to the lack of static checking. This is all the more true as most of the time configuration definitions are known at compile time. For these reasons, it is better to create adhoc configuration loader implementations. Fortunately, the configuration Winter compiler plugin can generate these for us.

In order to create a configuration bean in a Winter module, we simply need to create an interface for our configuration as specified above and annotates it with `@Configuration`, this will tell the configuration Winter compiler plugin to generate a corresponding configuration loader implementation as well as a module bean making our configuration directly available inside our module.

```

@Configuration
public interface AppConfiguration {

    // query property 'server_host'
    String server_host();

    // query property 'server_port'
    int server_port();
}

```

The preceding code will result in the generation of class `AppConfigurationLoader` which can then be used to load configuration at runtime without resorting to reflection.

```
ConfigurationSource<?, ?, ?> source = ...
```

```

new AppConfigurationLoader()
    .withSource(source)
    .withParameters("environment", "production")
    .load()
    .map(configuration -> startServer(configuration.server_host(), configuration.server_port()))
    .subscribe();

```

A configuration can also be obtained *manually* as follows:

```
AppConfiguration defaultConfiguration = AppConfigurationLoader.load(configurator ->
configurator.server_host("0.0.0.0"));
```

```
AppConfiguration customConfiguration = AppConfigurationLoader.load(configurator ->
configurator.server_host("0.0.0.0"));
```

By default, the generated loader also defines an overridable module bean which loads the configuration in the module. This bean defines three optional sockets:

- **configurationSource** indicates the configuration source to query when initializing the configuration bean
- **parameters** indicates the parameters to consider when querying the source
- **configurer** provides a way to overrides default values

If no configuration source is present, a default configuration is created, otherwise the configuration source is queried with the parameters, the resulting configuration is then *patched* with the configurer if present. The bean is overridable by default which means we can inject our own implementation if we feel like it.

It is possible to disable the activation of the configuration bean or make it non overridable in the `@Configuration` interface:

```

@Configuration(generateBean = false, overridable = false)
public interface AppConfiguration {
    ...
}

```

Finally, nested beans can be specified in a configuration which is convenient when a module is composing multiple modules and we wish to aggregate all configurations into one single representation in the composite module.

For instance, we can have the following configuration defined in a component module:

```
@Configuration
public interface ComponentModuleConfiguration {
    ...
}
```

and the following configuration defined in the composite module:

```
@Configuration
public interface CompositeModuleConfiguration {

    @NestedBean
    ComponentModuleConfiguration component_module_configuration();
}
```

In the preceding example, we basically indicate to the Winter framework that the `ComponentModuleConfiguration` defined in the `CompositeModuleConfiguration` must be injected into the component module instance.

HTTP Base

The Winter *http-base* module defines the foundational API for creating HTTP clients and servers. It also provides common HTTP services such as the header service.

In order to use the Winter *http-base* module, we need to declare a dependency in the module descriptor:

```
module io.winterframework.example.app {
    requires io.winterframework.mod.http.base;
    ...
}
```

And also declare that dependency in the build descriptor:

Using Maven:

```
<project>
  <dependencies>
    <dependency>
      <groupId>io.winterframework.mod</groupId>
      <artifactId>winter-http-base</artifactId>
    </dependency>
  </dependencies>
</project>
```

Using Gradle:

```
...
compile 'io.winterframework.mod:winter-http-base:1.0.0'
...
```

The *http-base* module is usually provided as a transitive dependency by other HTTP modules, the *http-server* module or the *web* module in particular, so this might not be necessary.

HTTP base API

The base HTTP API defines common classes and interfaces for implementing applications or modules using HTTP/1.x or HTTP/2 protocols. This includes:

- HTTP methods and status enumerations
- Exception bindings for HTTP errors: `BadRequestException`, `InternalServerErrorException`...
- basic building blocks such as `Parameter` which defines the base interface for any HTTP component that can be represented as a key/value pair (eg. query parameter, header, cookie...)
- Cookie types: `Cookie` and `SetCookie`
- Common HTTP header names (`Headers.NAME_*`) and values (`Headers.VALUE_*`) constants
- Common HTTP header types: `Headers.ContentType`, `Headers.Accept`...
- HTTP header codec API for implementing HTTP header codec used to decode a raw HTTP header in a specific `Header` object
- A HTTP header service used to encode/decode HTTP headers from/to specific `Header` objects

HTTP header service

The HTTP header service is the main entry point for decoding and encoding HTTP headers.

The `HeaderService` interface defines method to decode/encode `Header` object from/to `String` or `ByteBuf`.

For instance, a `content-type` header can be parsed as follows:

```
HeaderService headerService = ...
```

```
Headers.ContentType contentType = headerService.<Headers.ContentType>decode("content-type",  
"application/xml;charset=utf-8");
```

```
// application/xml  
String mediaType = contentType.getMediaType();  
// utf-8  
Charset charset = contentType.getCharset();
```

The `http-base` module provides a default implementation exposed as a bean which relies on a set of `HeaderCodec` to support specific headers. Custom header codecs can then be injected in the module to extend its capabilities.

For instance, we can create an `ApplicationContextHeaderCodec` codec in order for the header service to decode custom `application-context` headers to `ApplicationContextHeader` instances. The codec must be injected in the `http-base` module either explicitly when creating the module or through dependency injection.

```

Base httpBase = new Base.Builder()
    .setHeaderCodecs(List.of(new ApplicationContextHeaderCodec()))
    .build();

httpBase.start();

ApplicationContextHeaderCodec decodedHeader = httpBase.headerService().
<ApplicationContextHeaderCodec>.decode("...")
...

httpBase.stop();

```

Most of the time the *http-base* module is composed in a composite module and as a result dependency injection should work just fine, so we simply need to declare the codec as a bean in the module composing the *http-base* module to extend the header service.

By default, the *http-base* module provides codecs for the following headers:

- `accept` as defined by [RFC 7231 Section 5.3.2](#)
- `accept-language` as defined by [RFC 7231 Section 5.3.5](#)
- `content-disposition` as defined by [RFC 6266](#)
- `content-type` as defined by [RFC 7231 Section 3.1.1.5](#)
- `cookie` as defined by [RFC 6265 Section 4.2](#)
- `set-cookie` as defined by [RFC 6265 Section 4.1](#)

HTTP Server

The Winter *http-server* module provides fully reactive HTTP/1.x and HTTP/2 server based on [Netty](#).

It especially supports:

- HTTP/1.x pipelining
- HTTP/2 over cleartext
- HTTP Compression
- TLS
- `application/x-www-form-urlencoded` body decoding
- `multipart/form-data` body decoding
- Server-sent events
- Cookies
- zero-copy file transfer when supported for fast resource transfer
- parameter conversion

The server is fully reactive, based on the reactor pattern and non-blocking sockets which means it requires a limited number of threads to supports thousands of connections with high end performances. This design offers multiple advantages starting with maximizing the usage of resources. It is also easy to scale the server up and down by specifying the number of threads we want to allocate to the server, which ideally corresponds to the number of CPU cores. All this makes it a perfect choice for microservices applications running in containers in the cloud.

This module lays the foundational service and API for building HTTP servers with more complex and advanced features, that is why you might sometimes find it a little bit low level but that is the price of performance. If you require higher level functionalities like request routing, content negotiation and automatic payload conversion please consider the [web module](#).

This module requires basic services like a [net service](#) and a [resource service](#) which are usually provided by the *boot* module, so in order to use the Winter *http-server* module, we should declare the following dependencies in the module descriptor:

```
@io.winterframework.core.annotation.Module
module io.winterframework.example.app_http {
    requires io.winterframework.mod.boot;
    requires io.winterframework.mod.http.server;
}
```

The *http-base* module which provides the header service used by the HTTP server is composed as a transitive dependency in the *http-server* module and as a result it doesn't need to be specified here nor provided in an enclosing module.

We also need to declare these dependencies in the build descriptor:

Using Maven:

```
<project>
  <dependencies>
    <dependency>
      <groupId>io.winterframework.mod</groupId>
      <artifactId>winter-boot</artifactId>
    </dependency>
    <dependency>
      <groupId>io.winterframework.mod</groupId>
      <artifactId>winter-http-server</artifactId>
    </dependency>
  </dependencies>
</project>
```

Using Gradle:

```
...
compile 'io.winterframework.mod:winter-boot:1.0.0'
compile 'io.winterframework.mod:winter-http-server:1.0.0'
...
```

HTTP Server exchange API

The module defines classes and interfaces to implement HTTP server exchange handlers used to handle HTTP requests sent by a client to the server.

A server *ExchangeHandler* is defined to handle a server *Exchange* composed of the *Request* and *Response* pair in a HTTP communication between a client and a server. The API has been designed to be fluent and reactive in order for the request to be *streamed* down to the response.

Basic exchange

The `ExchangeHandler` is a functional interface, a basic exchange handler can then be created as follows:

```
ExchangeHandler<Exchange> handler = exchange -> {
    exchange.response()
        .headers(headers -> headers.contentType(MediaType.TEXT_PLAIN))
        .body().raw().value(Unpooled.unreleasableBuffer(Unpooled.copiedBuffer("Hello, world!",
Charsets.DEFAULT)));
};
```

The above code creates an exchange handler sending a `Hello, world!` message in response to any request.

We might also want to send the response in a reactive way in a stream of data in case the entire response payload is not available right away, if it doesn't fit in memory or if we simply want to send a response in multiple parts as soon as they become available (eg. progressive display).

```
ExchangeHandler<Exchange> handler = exchange -> {
    Flux<ByteBuffer> dataStream = Flux.just(
        Unpooled.unreleasableBuffer(Unpooled.copiedBuffer("Hello", Charsets.DEFAULT)),
        Unpooled.unreleasableBuffer(Unpooled.copiedBuffer(", world!", Charsets.DEFAULT))
    );

    exchange.response()
        .body().raw().stream(dataStream);
};
```

Request body

Request body can be handled in a similar way. The reactive API allows to process the payload of a request as the server receives it and therefore progressively build and send the corresponding response.

```
ExchangeHandler<Exchange> handler = exchange -> {
    exchange.response()
        .body().raw().stream(exchange.request().body()
            .map(body -> Flux.from(body.raw().stream()).map(chunk ->
Unpooled.unreleasableBuffer(Unpooled.buffer(4).writeInt(chunk.readableBytes()))))
            .orElse(Flux.just(Unpooled.unreleasableBuffer(Unpooled.buffer(4).writeInt(0))))
        );
};
```

In the above example, if a client sends a payload in the request, the server responds with the number of bytes of each chunk of data it receives or it responds `0` if the request payload is empty. This simple example illustrates how we can process requests as flow of data

URL Encoded form

HTML form data are sent in the body of a POST request in the form of key/value pairs encoded in [application/x-www-form-urlencoded format](#). The resulting list of `Parameter` can be obtained as follows:


```

ExchangeHandler<Exchange> handler = exchange -> {
    exchange.response()
        .body().raw().stream(Flux.from(exchange.request().body().get().urlEncoded().stream())
            .map(parameter -> Unpooled.copiedBuffer(Unpooled.copiedBuffer("Received parameter " +
parameter.getName() + " with value " + parameter.getValue(), Charsets.DEFAULT)))
        );
}

```

In the above example, for each form parameters the server responds with a message describing the parameters it just received. Again this shows that the API is fully reactive and form parameters can be processed as they are decoded.

A more traditional example though would be to obtain the map of parameters grouped by names (because multiple parameters with the same name can be sent):

```

ExchangeHandler<Exchange> handler = exchange -> {
    exchange.response()
        .body().raw().stream(Flux.from(exchange.request().body().get().urlEncoded().stream())
            .collectMultimap(Parameter::getName)
            .map(formParameters -> Unpooled.unreleasableBuffer(Unpooled.copiedBuffer("User selected
options: " +
formParameters.get("options").stream().map(Parameter::getValue).collect(Collectors.joining(", ")),
Charsets.DEFAULT)))
        );
}

```

Here we may think that the aggregation of parameters in a map could *block* the I/O thread but this is definitely not true, when a parameter is decoded, the reactive framework is notified and the parameter is stored in a map, after that the I/O thread can be reallocated. When the parameters publisher completes the resulting map is emitted to the mapping function which build the response. During all this process, no thread is ever waiting for anything.

Multipart form

A [multipart/form-data](#) request can be handled in a similar way. Form parts can be obtained as follows:

```

ExchangeHandler<Exchange> handler = exchange -> {
    exchange.response()
        .body().raw().stream(Flux.from(exchange.request().body().get().multipart().stream())
            .map(part -> Unpooled.copiedBuffer(Unpooled.copiedBuffer("Received part " +
part.getName(), Charsets.DEFAULT)))
        );
};

```

In the above example, the server responds with the name of the part it just received. Parts are decoded and can be processed along the way, a part is like a body embedded in the request body with its own headers and payload.

Multipart form data is most commonly used for uploading files over HTTP. Such handler can be implemented as follows using the [resource API](#) to store uploaded files:

```

ExchangeHandler<Exchange> handler = exchange -> {
    exchange.response()
        .body().raw().stream(Flux.from(exchange.request().body().get().multipart().stream()))
// 1
        .single()
// 2
        .flatMapMany(part -> part.getFilename()
// 3
            .map(fileName -> Flux.<ByteBuf, FileResource>using(
// 4
                () -> new FileResource("uploads/" + part.getFilename().get()),
// 5
                file -> file.write(part.raw().stream()).map(Flux::from).get()
// 6
                    .reduce(0, (acc, cur) -> acc + cur)
                    .map(size -> Unpooled.unreleasableBuffer(Unpooled.copiedBuffer("Uploaded
" + fileName + "(" + part.headers().getContentType() + "): " + size + " Bytes\n",
Charsets.DEFAULT))),
                    FileResource::close
// 7
                )
            )
        .orElseThrow(() -> new BadRequestException("Not a file part"))
// 8
    );
};

```

The above code uses multiple elements and deserves a detailed explanation:

1. get the stream of parts
2. make sure we only have one part in the request for the sake of simplicity
3. map the part to the response stream by starting to determine whether the part is a file part
4. if the part is a file part indeed, map the part to the response stream by creating a Flux with a file resource
5. in this case the resource is the target file where the uploaded file will be stored
6. stream the part's payload to the target file resource and eventually provides the response in the form of a message stating that a file with a given size and media type has been uploaded
7. close the file resource when the publisher completes
8. if the part is not a file part respond with a bad request error

The `Flux.using()` construct is the reactive counterpart of a try-with-resource statement. It is interesting to note that the content of the file is streamed up to the file and it is then never entirely loaded in memory. From there, it is quite easy to stop the upload of a file if a given size threshold is exceeded. We can also imagine how we could create a progress bar in a client UI to show the progression of the upload.

In the above code we uploaded a file and stored its content on the local file system and during all that process, the I/O thread was never blocked.

Resource

A [resource](#) can be sent as a response to a request. When this is possible the server uses low-level ([zero-copy](#)) API for fast resource transfer.

```
ExchangeHandler<Exchange> handler = exchange -> {
    exchange.response()
        .body().resource().value(new FileResource("/path/to/resource"));
};
```

The media type of the resource is resolved using a [media type service](#) and automatically set in the response `content-type` header field.

If a specific resource is created as in above example the media type service used is the one defined when creating the resource or a default implementation if none was specified. If the resource is obtained with the resource service provided in the *boot* module the media type service used is the one provided in the *boot* module.

Server-sent events

[Server-sent events](#) provide a way to send server push notifications to a client. It is based on [chunked transfer encoding](#) over HTTP/1.x and regular streams over HTTP/2. The API provides an easy way to create SSE endpoints.

```
ExchangeHandler<Exchange> handler = exchange -> {
    exchange.response().body().sse().from(
        (events, data) -> data.stream(Flux.interval(Duration.ofSeconds(1))
            .map(seq -> events.create(event -> event
                .id(Long.toString(seq))
                .event("seq")
                .comment("Some comment")
                .value(Unpooled.unreleasableBuffer(Unpooled.copiedBuffer("Event #" + seq,
Charsets.DEFAULT))))))
    );
};
```

In the above example, server-sent events are emitted every second and streamed to the response. This is done in a function accepting the server-sent event factory used to create events and the response data producer.

Error exchange handler

An error exchange handler is a particular exchange handler which is defined to handle server error exchange. In other words it is used by the server to handle exceptions thrown during the processing of a regular exchange in order to send an appropriate response to the client when this is still possible (ie. assuming response headers haven't been sent yet).

```

ExceptionHandler<Throwable> errorHandler = errorExchange -> {
    if(errorExchange.getError() instanceof BadRequestException) {
        errorExchange.response()
            .headers(headers -> headers.status(Status.BAD_REQUEST))
            .body().raw().value(Unpooled.unreleasableBuffer(Unpooled.copiedBuffer("client sent an
invalid request", Charsets.DEFAULT)));
    }
    else {
        errorExchange.response()
            .headers(headers -> headers.status(Status.INTERNAL_SERVER_ERROR))
            .body().raw().value(Unpooled.unreleasableBuffer(Unpooled.copiedBuffer("Unknown server
error", Charsets.DEFAULT)));
    }
};

```

Misc

The API is fluent and mostly self-describing as a result it should be easy to find out how to do something in particular, even so here are some miscellaneous elements.

Request headers

A particular request header can be obtained as follows, if there are multiple headers with the same name, the first one shall be returned:

```

ExceptionHandler<Exchange> handler = exchange -> {
    ...
    // get the raw value of a header
    String someHeader = exchange.request().headers().get("some-header").orElseThrow(() -> new
BadRequestException("Missing some-header"));

    // get a header as a parameter that can be converted using the parameter converter
    LocalDateTime someDateTime = exchange.request().headers().getParameter("some-date-
time").map(Parameter::asLocalDateTime).orElseThrow(() -> new BadRequestException("Missing some-date-
time"));

    // get a decoded header using the header service
    CustomHeader customHeader = exchange.request().headers().<CustomHeader>getHeader("custom-
header").orElseThrow(() -> new BadRequestException("Missing some-date-time"));
    ...
};

```

All headers with a particular names can be obtained as follows:

```

ExceptionHandler<Exchange> handler = exchange -> {
    ...
    // get all raw values defined for a given header
    List<String> someHeaderList = exchange.request().headers().getAll("some-header");

    // get all headers with a given header as parameters that can be converted using the parameter
converter
    LocalDateTime someDateTime = exchange.request().headers().getParameter("some-date-
time").map(Parameter::asLocalDateTime).orElseThrow(() -> new BadRequestException("Missing some-date-
time"));

    // get all headers with a given name decoded using the header service
    CustomHeader customHeader = exchange.request().headers().<CustomHeader>getHeader("custom-
header").orElseThrow(() -> new BadRequestException("Missing some-date-time"));
    ...
};

```

Finally we can retrieve all headers as follows:

```
ExchangeHandler<Exchange> handler = exchange -> {
    ...
    // get all headers with raw values
    List<Map.Entry<String, String>> requestHeaders = exchange.request().headers().getAll();

    // get all headers as parameters that can be converted using the parameter converter
    List<Parameter> requestHeaderParameters = exchange.request().headers().getAllParameter();

    // get all headers decoded using the header service
    List<Header> requestDecodedHeaders = exchange.request().headers().getAllHeader();
    ...
};
```

Query parameters

Query parameters in the request can be obtained as follows:

```
ExchangeHandler<Exchange> handler = exchange -> {
    ...
    // get a specific query parameter, if there are multiple parameters with the same name, the
    first one is returned
    int someInteger = exchange.request().queryParameters().get("some-
integer").map(Parameter::asInteger).orElseThrow(() -> new BadRequestException("Missing some-
integer"));

    // get all query parameters with a given name
    List<Integer> someIntegers = exchange.request().queryParameters().getAll("some-
integer").stream().map(Parameter::asInteger).collect(Collectors.toList());

    // get all query parameters
    Map<String, List<Parameter>> queryParameters = exchange.request().queryParameters().getAll();
    ...
};
```

Request cookies

Request cookie can be obtained in a similar way as follows:

```
ExchangeHandler<Exchange> handler = exchange -> {
    ...
    // get a specific cookie, if there are multiple cookie with the same name, the first one is
    returned
    int someInteger = exchange.request().cookies().get("some-
integer").map(Parameter::asInteger).orElseThrow(() -> new BadRequestException("Missing some-
integer"));

    // get all cookies with a given name
    List<Integer> someIntegers = exchange.request().cookies().getAll("some-
integer").stream().map(Parameter::asInteger).collect(Collectors.toList());

    // get all cookies
    Map<String, List<CookieParameter>> queryParameters = exchange.request().cookies().getAll();
    ...
};
```

Note that cookies can also be obtained as request headers.

Request components

The API also gives access to multiple request related information such as:

- the HTTP method
- the scheme ([http](#) or [https](#))
- the authority part of the requested URI ([host](#) header in HTTP/1.x and [:authority](#) pseudo-header in HTTP/2)
- the requested path including query string
- the absolute path which is the normalized requested path without the query string
- the [URIBuilder](#) corresponding to the requested path to build relative paths
- the query string
- the socket address of the client or last proxy that sent the request

Response headers/trailers

Response headers can be added or set fluently using a configurator as follows:

```
ExchangeHandler<Exchange> handler = exchange -> {
    exchange.response()
        .headers(headers -> headers
            .contentType(MediaType.TEXT_PLAIN)
            .set(Headers.NAME_SERVER, "winter")
            .add("custom-header", "abc")
        )
        .body().raw()...;
};
```

Response trailers can be set in the exact same way:

```
ExchangeHandler<Exchange> handler = exchange -> {
    exchange.response()
        .trailers(headers -> headers
            .add("some-trailer", "abc")
        )
        .body().raw()...;
};
```

Response status

The response status can be set in the response headers following HTTP/2 specification as defined by [RFC 7540 Section 8.1.2.4](#).

```
ExchangeHandler<Exchange> handler = exchange -> {
    exchange.response()
        .headers(headers -> headers.status(Status.OK))
        .body().raw();
};
```

Response cookies

Response cookies can be set fluently using a configurator as follows:

```
ExchangeHandler<Exchange> handler = exchange -> {
    exchange.response()
        .cookies(cookies -> cookies
            .addCookie(cookie -> cookie.name("cookie1")
                .httpOnly(true)
                .secure(true)
                .maxAge(3600)
                .value("abc")
            )
            .addCookie(cookie -> cookie.name("cookie2")
                .httpOnly(true)
                .secure(true)
                .maxAge(3600)
                .value("def")
            )
        )
        .body().raw()...;
};
```

Note that cookies can also be set or added as response headers.

HTTP Server

The HTTP server is started with the *http-server* module which requires a *NetService* and a *ResourceService* usually provided by the *boot* module, so one way to create an application with a HTTP server is to create a Winter module composing the *boot* and *http-server* modules.

```
@io.winterframework.core.annotation.Module
module io.winterframework.example.app_http {
    requires io.winterframework.mod.boot;
    requires io.winterframework.mod.http.server;
}
```

The resulting *app_http* module, thus created, can then be started as an application as follows:

```
package io.winterframework.example.app_http;

import io.winterframework.core.v1.Application;

public class Main {

    public static void main(String[] args) {
        Application.with(new App_http.Builder()).run();
    }
}
```

The above example starts a HTTP/1.x server using default configuration and default root and error handlers.

this module can also be used to embed a HTTP server in any application, unlike other application frameworks, Winter core IoC/DI framework is not pervasive and any Winter modules can be safely used in various contexts and applications.

Configuration

The first thing we might want to do is to create a configuration in the *app_http* module for easy *http-server* module setup. The HTTP server configuration is actually done in the *NetConfiguration* defined in the *boot* module for low level network configuration and *HttpServerConfiguration* in the *http-server* module configuration for the HTTP server itself.

The following configuration can then be created in the *app_http* module:

```
package io.winterframework.example.app_http;

import io.winterframework.core.annotation.NestedBean;
import io.winterframework.mod.boot.NetConfiguration;
import io.winterframework.mod.configuration.Configuration;
import io.winterframework.mod.http.server.HttpServerConfiguration;

@Configuration
public interface App_httpConfiguration {

    @NestedBean
    NetConfiguration net();

    @NestedBean
    HttpServerConfiguration http_server();
}
```

This should be enough for exposing a configuration in the *app_http* module, that let us setup the server:

```
package io.winterframework.example.app_http;

import io.winterframework.core.v1.Application;

public class Main {

    public static void main(String[] args) {
        Application.with(new App_http.Builder()
            .setApp_httpConfiguration(
                App_httpConfigurationLoader.load(configuration -> configuration
                    .http_server(server -> server
                        .server_port(8081)
                        .h2c_enabled(true)
                    )
                    .net(net -> net
                        .root_event_loop_group_size(4)
                    )
                )
            ).run();
    }
}
```

In the above code, we have set the server port to 8081, enabled HTTP/2 over cleartext and set the number of thread allocated to the root event loop group to 4.

Please refer to the [API documentation](#) to have an exhaustive description of the different configuration properties. We can for instance configure low level network settings like TCP keep alive or TCP no delay as well as HTTP related settings like compression or TLS.

You can also refer to the [configuration module documentation](#) to get more details on how configuration works and more especially how you can from here define the HTTP server configuration in command line arguments, property files...

By default, the HTTP server uses the Java NIO transport, but it is possible to use native [epoll](#) transport on Linux or [kqueue](#) transport on BSD-like systems for optimized performances. This can be done by adding the corresponding Netty dependency with the right classifier in the project descriptor:

```
<project>
  <dependencies>
    <dependency>
      <groupId>io.netty</groupId>
      <artifactId>netty-transport-native-epoll</artifactId>
      <classifier>linux-x86_64</classifier>
    </dependency>
  </dependencies>
</project>
```

or

```
<project>
  <dependencies>
    <dependency>
      <groupId>io.netty</groupId>
      <artifactId>netty-transport-native-kqueue</artifactId>
      <classifier>osx-x86_64</classifier>
    </dependency>
  </dependencies>
</project>
```

When these dependencies are declared on the JVM module path, the corresponding Java modules must be added explicitly when running the application. This is typically the case when the application is run or packaged as an application image using the Winter Maven plugin.

This can be done by defining the corresponding dependencies in the module descriptor:

```
@io.winterframework.core.annotation.Module
module io.winterframework.example.app {
  ...
  requires io.netty.transport.unix.common;
  requires io.netty.transport.epoll;
}
```

This approach is fine as long as we are sure the application will run on Linux, but in order to create a properly portable application, we should prefer adding the modules explicitly when running the application:

```
$ java --add-modules io.netty.transport.unix.common,io.netty.transport.epoll ...
```

When building an application image, this can be specified in the Winter Maven plugin configuration:

```
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>io.winterframework.tool</groupId>
        <artifactId>winter-maven-plugin</artifactId>
        <executions>
          <execution>
            <configuration>
              <vmOptions>--add-modules
io.netty.transport.unix.common,io.netty.transport.epoll</vmOptions>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Root handler

The HTTP server defines a root exchange handler to handle all HTTP requests. By default, it uses a basic handler implementation which returns **Hello** when a request is made to the root path **/** and return (404) not found errors otherwise.

In order to use our own handler, we must define an exchange handler bean in the *app_http* module:

```
package io.winterframework.example.app_http;

import io.netty.buffer.Unpooled;
import io.winterframework.core.annotation.Bean;
import io.winterframework.mod.base.Charsets;
import io.winterframework.mod.http.base.HttpException;
import io.winterframework.mod.http.server.Exchange;
import io.winterframework.mod.http.server.ExchangeHandler;

@Bean
public class CustomHandler implements ExchangeHandler<Exchange> {

    @Override
    public void handle(Exchange exchange) throws HttpException {
        exchange.response()
            .body().raw()
                .value(Unpooled.unreleasableBuffer(Unpooled.copiedBuffer("Hello from app_http module!",
Charsets.DEFAULT)));
    }
}
```

This bean will be automatically wired to the root handler socket defined by the *http-server* module overriding the default root handler. If we don't want to provide a handler implementation inside the *app_http* module, we can also define a socket bean for the root handler and provide an instance when creating the *app_http* module.

```
package io.winterframework.example.app_http;

import java.util.function.Supplier;

import io.netty.buffer.Unpooled;
import io.winterframework.core.annotation.Bean;
import io.winterframework.core.v1.Application;
import io.winterframework.mod.base.Charsets;
import io.winterframework.mod.http.server.Exchange;
import io.winterframework.mod.http.server.ExchangeHandler;

public class Main {

    @Bean
    public static interface Handler extends Supplier<ExchangeHandler<Exchange>> {}

    public static void main(String[] args) {
        Application.with(new App_http.Builder()
            .setHandler(exchange -> {
                exchange.response()
                    .body().raw()
                    .value(Unpooled.unreleasableBuffer(Unpooled.copiedBuffer("Hello from main!",
Charsets.DEFAULT)));
            })
        ).run();
    }
}
```

Note that this socket bean is optional since the root handler socket on the *http-server* module to which it is wired is itself optional.

Error handler

The HTTP server defines an error exchange handler to handle exceptions thrown when processing HTTP requests when this is still possible, basically when the response headers haven't been sent yet to the client. By default, it uses a basic error handler implementation which handles standard `HttpException` and responds empty body messages with HTTP error status corresponding to the exception.

This default implementation should be enough for a basic HTTP server but a custom handler can be provided to produce custom error pages for specific types of error. This can be done in the exact same way as the [root handler](#) by defining an error exchange handler bean:

```

package io.winterframework.example.app_http;

import io.winterframework.core.annotation.Bean;
import io.winterframework.mod.http.base.HttpException;
import io.winterframework.mod.http.server.ErrorExchange;
import io.winterframework.mod.http.server.ExchangeHandler;

@Bean
public class CustomErrorHandler implements ExchangeHandler<ErrorExchange<Throwable>> {

    @Override
    public void handle(ErrorExchange<Throwable> exchange) throws HttpException {
        if(exchange.getError() instanceof SomeCustomException) {
            ...
        }
        else if(...) {
            ...
        }
        ...
        else {
            ...
        }
    }
}

```

Or by defining a socket bean:

```

package io.winterframework.example.app_http;

import java.util.function.Supplier;

import io.netty.buffer.Unpooled;
import io.winterframework.core.annotation.Bean;
import io.winterframework.core.v1.Application;
import io.winterframework.mod.base.Charsets;
import io.winterframework.mod.http.server.ErrorExchange;
import io.winterframework.mod.http.server.Exchange;
import io.winterframework.mod.http.server.ExchangeHandler;

public class Main {

    @Bean
    public static interface Handler extends Supplier<ExchangeHandler<Exchange>> {}

    @Bean
    public static interface ErrorHandler extends Supplier<ErrorExchangeHandler<Throwable>> {}

    public static void main(String[] args) {
        Application.with(new App_http.Builder()
            .setErrorHandler(exchange -> {
                exchange.response()
                    .headers(headers -> headers.status(500))
                    .body().raw()
                    .value(Unpooled.unreleasableBuffer(Unpooled.copiedBuffer("Error: " +
exchange.getError().getMessage(), Charsets.DEFAULT)));
            })).run();
    }
}

```

HTTP compression

HTTP compression can be activated by configuration for request and/or response. For instance:

```
public class Main {  
  
    public static void main(String[] args) {  
        Application.with(new App_http.Builder()  
            .setApp_httpConfiguration(  
                App_httpConfigurationLoader.load(configuration -> configuration  
                    .http_server(server -> server  
                        .decompression_enabled(true)  
                        .compression_enabled(true)  
                        .compression_level(6)  
                    )  
            )  
        ).run();  
    }  
}
```

Now if we send a request which accepts compression to the server, we should now receive a compressed response:

```
$ curl -i --compressed -H 'accept-encoding: gzip, deflate' http://localhost:8080  
HTTP/1.1 200 OK  
content-type: text/plain  
server: winter  
content-encoding: gzip  
content-length: 39  
  
Hello
```

TLS configuration

In order to activate TLS, we need first to obtain a private key and a certificate stored in a keystore.

A self-signed certificate can be generated using `keytool`, the resulting keystore should be placed in `src/main/resources` to make it available as a module resource:

```
$ keytool -genkey -keyalg RSA -alias selfsigned -keystore keystore.jks -storepass password -validity  
360 -keysize 2048
```

Then we need to configure the server to activate TLS using the certificate:

```

public class Main {

    public static void main(String[] args) {
        Application.with(new App_http.Builder()
            .setApp_httpConfiguration(
                App_httpConfigurationLoader.load(configuration -> configuration
                    .http_server(server -> server
                        .server_port(8443)
                        .tls_enabled(true)

            .key_store(URI.create("module://io.winterframework.example.app_http/keystore.jks"))
                .key_alias("selfsigned")
                .key_store_password("password")
            )
        )
    ).run();
}
}

```

When an application using the *http-server* module is packaged as an application image, you'll need to make sure TLS related modules from the JDK are included in the runtime image otherwise TLS might not work. You can refer to the [JDK providers documentation](#) in the security developer's guide to find out which modules should be added depending on your needs. Most of the time you'll simply add `jdk.crypto.ec` module in the Winter Maven plugin configuration:

```

<project>
  <build>
    <plugins>
      <plugin>
        <groupId>io.winterframework.tool</groupId>
        <artifactId>winter-maven-plugin</artifactId>
        <executions>
          <execution>
            <configuration>
              <addModules>jdk.crypto.ec</addModules>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>

```

Extend HTTP services

The HTTP server relies on a header service and a parameter converter to respectively decode HTTP headers and convert parameter values.

The *http-server* module defines a socket to plug custom `HeaderCodec` instances so the HTTP header service can be extended to decode custom HTTP headers as described in the [HTTP header service documentation](#).

It also defines a socket to plug a custom parameter converter which is a basic `StringConverter` by default. Since we created the `app_http` module by composing `boot` and `http-server` modules, the parameter converter provided by the `boot` module should then override the default. This converter is a `StringCompositeConverter` which can be extended by injecting custom `CompoundDecoder` and/or `CompoundEncoder` instances in the `boot` module as described in the [composite converter documentation](#).

To sum up, all we have to do to extend these services is to provide `HeaderCodec`, `CompoundDecoder` or `CompoundEncoder` beans in the `app_http` module.

Wrap-up

If we put all we've just seen together, here is a complete example showing how to create a HTTP/2 server with HTTP compression using custom root and error handlers:


```

package io.winterframework.example.app_http;

import java.net.URI;
import java.util.function.Supplier;

import io.netty.buffer.Unpooled;
import io.winterframework.core.annotation.Bean;
import io.winterframework.core.v1.Application;
import io.winterframework.mod.base.Charsets;
import io.winterframework.mod.http.server.Exchange;
import io.winterframework.mod.http.server.ExchangeHandler;
import io.winterframework.mod.http.server.ExchangeHandler<Exchange>;

public class Main {

    @Bean
    public static interface Handler extends Supplier<ExchangeHandler<Exchange>> {}

    @Bean
    public static interface ErrorHandler extends Supplier<ErrorHandler<Throwable>> {}

    public static void main(String[] args) {
        Application.with(new App_http.Builder()
            .setApp_httpConfiguration(
                App_httpConfigurationLoader.load(configuration -> configuration
                    .http_server(server -> server
                        // HTTP compression
                        .decompression_enabled(true)
                        .compression_enabled(true)
                        // TLS
                        .server_port(8443)
                        .tls_enabled(true)

                    .key_store(URI.create("module://io.winterframework.example.app_http/keystore.jks"))
                        .key_alias("selfsigned")
                        .key_store_password("password")
                        // Enable HTTP/2
                        .h2_enabled(true)
                    )
                )
            )
            .setHandler(exchange -> {
                exchange.response()
                    .body().raw().value(Unpooled.unreleasableBuffer(Unpooled.copiedBuffer("Hello
from main!", Charsets.DEFAULT)));
            })
            .setErrorHandler(exchange -> {
                exchange.response()
                    .headers(headers -> headers.status(500))
                    .body().raw().value(Unpooled.unreleasableBuffer(Unpooled.copiedBuffer("Error: " +
exchange.getError().getMessage(), Charsets.DEFAULT)));
            })
        ).run();
    }
}

$ curl -i --insecure https://localhost:8443/
HTTP/2 200
content-length: 16

Hello from main!

```

Web

The *Winter web* module provides extended functionalities on top of the *http-server* module for developing Web and RESTfull applications.

It especially provides:

- HTTP request routing
- content negotiation
- automatic message payload conversion
- path parameters
- static handler for serving static resources
- version agnostic [WebJars](#) support
- easy Web/REST controller development
- [OpenAPI](#) specifications generation using Web controllers JavaDoc comments
- SwaggerUI integration
- a Winter compiler plugin providing static validation of the routes and generation of Web router configurers

The *web* module composes the *http-server* module and therefore starts a HTTP server. Just like the *http-server* module, it requires a net service and a resource service as well as a list of [media type converters](#) for message payload conversion. Basic implementations of these services are provided by the *boot* module which provides `application/json`, `application/x-ndjson` and `text/plain` media type converters. Additional media type converters can surely be provided by implementing the `MediaTypeConverter` interface.

In order to use the Winter *web* module, we should declare the following dependencies in the module descriptor:

```
@io.winterframework.core.annotation.Module
module io.winterframework.example.app_web {
    requires io.winterframework.mod.boot;
    requires io.winterframework.mod.web;
}
```

We also need to declare these dependencies in the build descriptor:

Using Maven:

```
<project>
  <dependencies>
    <dependency>
      <groupId>io.winterframework.mod</groupId>
      <artifactId>winter-boot</artifactId>
    </dependency>
    <dependency>
      <groupId>io.winterframework.mod</groupId>
      <artifactId>winter-web</artifactId>
    </dependency>
  </dependencies>
</project>
```

Using Gradle:

```
...
compile 'io.winterframework.mod:winter-boot:1.0.0'
compile 'io.winterframework.mod:winter-web:1.0.0'
...
```

Routing

The *web* module also defines an API for routing HTTP requests to the right handlers.

A **router** is a standard server exchange handler as defined by the *http-server* module API which can be used as root handler or error handler in the HTTP server, its role is to route an exchange to a handler based on a set of rules applied to the exchange.

A **route** specifies the rules that an exchange must match to be routed to a particular handler.

A **route manager** is used to manage the routes in a router or, more explicitly, to list, create, enable or disable routes in a router.

The module defines a high level routing API with **Router**, **Route**, **RouteManager** and **RouterConfigurer** that can be used as a base to implement custom routing implementations in addition to the provided Web and error routing implementations. Nevertheless, it is more of a guideline, one can choose a totally different approach to implement routing, in the end the HTTP server expects a **ExchangeHandler<ByteBuf>** what is done inside is completely opaque, the Web API only shows one way to do it.

Web routing

A **WebRouter** is used to route a **WebExchange** to the right **WebExchangeHandler**, it implements **ExchangeHandler<ByteBuf>** and it is typically used as root handler in the HTTP server.

Web Server exchange

The *web* module API extends the [server exchange API](#) defined in the *http-server* module and defines the server **WebExchangeHandler** to handle a server **WebExchange** composed of the **WebRequest** and **WebResponse** pair in a HTTP communication between a client and a server. These interfaces respectively extend the **ExchangeHandler**, **Exchange**, **Request** and **Response** interfaces which are defined in the *http-server* module. A Web exchange handler is typically attached to one or more Web routes defined in a **WebRouter**.

Exchange attributes

Attributes can be set on a **WebExchange** to propagate contextual information such as a security or functional context in a chain of Web exchange handlers:

```

WebExchangeHandler<WebExchange> handler = exchange -> {
    exchange.<SecurityContext>getAttribute("security_context")
        .filter(SecurityContext::isAuthenticated)
        .ifPresentOrElse(
            securityContext -> {
                ...
            },
            () -> exchange.response().headers(headers ->
headers.status(Status.UNAUTHORIZED)).body().empty()
        );
};

WebExchangeHandler<WebExchange> securityInterceptor = exchange -> {
    SecurityContext securityContext = ...;
    exchange.setAttribute("security_context", securityContext);

    handler.handle(exchange);
};

```

Path parameters

Path parameters are exposed in the `WebRequest`, they are extracted from the requested path by the [Web router](#) when the handler is attached to a route matching a parameterized path as defined in a [URI builder](#).

For instance, if the handler is attached to a route matching `/book/{id}`, the `id` path parameter can be retrieved as follows:

```

WebExchangeHandler<WebExchange> handler = exchange -> {
    exchange.request().pathParameters().get("id")
        .ifPresentOrElse(
            id -> {
                ...
            },
            () -> exchange.response().headers(headers ->
headers.status(Status.NOT_FOUND)).body().empty()
        );
};

```

Request body decoder

The request body can be decoded based on the content type defined in the request headers.

```

WebExchangeHandler<WebExchange> handler = exchange -> {
    Mono<Result> storeBook = exchange.request().body().get()
        .decoder(Book.class)
        .one()
        .map(book -> storeBook(book));
    exchange.response().body()
        .raw().stream(storeBook.map(result ->
Unpooled.unreleasableBuffer(Unpooled.copiedBuffer(result.getMessage(), Charsets.DEFAULT))));
};

```

When invoking the `decoder()` method, a [media type converter](#) corresponding to the request content type is selected to decode the payload. The `content-type` header MUST be specified in the request, otherwise (400) bad request error is returned indicating an empty media type. If there is no converter corresponding to the media type, an (415) unsupported media type error is returned indicating that no decoder was found matching the content type.

A decoder is obtained by specifying the type of the object to decode in the `decoder()` method, the type can be a `Class<T>` or a `java.lang.reflect.Type` which allows to decode parameterized types at runtime bypassing type erasure. Parameterized Types can be built at runtime using the [reflection API](#).

As you can see in the above example the decoder is fully reactive, a request payload can be decoded in a single object by invoking method `one()` on the decoder which returns a `Mono<T>` publisher or in a stream of objects by invoking method `many()` on the decoder which returns a `Flux<T>` publisher.

Decoding multiple payload objects is indicated when a client streams content to the server. For instance, it can send a request with `application/x-ndjson` content type in order to send multiple messages in a single request. Since everything is reactive the server doesn't have to wait for the full request and it can process a message as soon as it is received. What is remarkable is that the code is largely unchanged.

```
WebExchangeHandler<WebExchange> handler = exchange -> {
    Flux<Result> storeBook = exchange.request().body().get()
        .decoder(Book.class)
        .many()
        .map(book -> storeBook(book));
    exchange.response().body()
        .raw().stream(storeBook.map(result ->
Unpooled.unreleasableBuffer(Unpooled.copiedBuffer(result.getMessage(), Charsets.DEFAULT))));
};
```

Conversion of a multipart form data request body is also supported, the payload of each part being decoded independently based on the content type of the part. For instance we can upload multiple books in multiple files in a `multipart/form-data` request and decode them on the fly as follows:

```
WebExchangeHandler<WebExchange> handler = exchange -> {
    exchange.response()
        .body().raw().stream(Flux.from(exchange.request().body().get().multipart().stream())
// 1
        .flatMap(part -> part.decoder(Book.class).one())
// 2
        .map(book -> storeBook(book))
// 3
        .map(result -> Unpooled.unreleasableBuffer(Unpooled.copiedBuffer(result.getMessage(),
Charsets.DEFAULT))) // 4
        );
};
```

In the previous example:

1. A stream of files is received in a `multipart/form-data` request (note that we assume all parts are file parts).
2. Each part is decoded to a `Book` object, the media type must be specified in the `content-type` header field of the part.
3. The book object so obtained is processed.
4. The result for each upload is returned to the client.

All this process is done in a reactive way, the first chunk of response can be sent before all parts have been processed.

Response body encoder

As for the request body, the response body can be encoded based on the content type defined in the response headers. Considering previous example we can do the following:

```
WebExchangeHandler<WebExchange> handler = exchange -> {
    Mono<Result> storeBook = exchange.request().body().get()
        .decoder(Book.class)
        .one()
        .map(book -> storeBook(book));
    exchange.response()
        .headers(headers -> headers.contentType(MediaType.APPLICATION_JSON))
        .body()
            .encoder(Result.class)
            .one(storeBook);
};
```

When invoking the `encoder()` method, a [media type converter](#) corresponding to the response content type is selected to encode the payload. The `content-type` header MUST be specified in the response, otherwise a (500) internal server error is returned indicating an empty media type. If there is no converter corresponding to the media type, a (500) internal server error is returned indicating that no encoder was found matching the content type.

A single object is encoded by invoking method `one()` on the encoder or multiple objects can be encoded by invoking method `many()` on the encoder. Returning multiple objects in a stream is particularly suitable to implement progressive display in a Web application, for example to display search results as soon as some are available.

```
WebExchangeHandler<WebExchange> handler = exchange -> {
    Flux<SearchResult> searchResults = ...;
    exchange.response()
        .headers(headers -> headers.contentType(MediaType.APPLICATION_X_NDJSON))
        .body()
            .encoder(SearchResult.class)
            .many(searchResults);
};
```

Web router

A `WebRouter` routes a Web server exchange to an exchange handler attached to a Web route by matching it against a combination of routing rules specified in the route. A Web route can combine the following routing rules which are matched in that order: the path, method and content type of the request, the media ranges and language ranges accepted by the client. For instance, the Web router matches an exchange against the path routing rule first, then the method routing rule... Multiples routes can then match a given exchange but only one will be retained to actually process the exchange which is the one matching the highest routing rules.

If a route doesn't define a particular routing rule, the routing rule is simply ignored and matches all exchanges. For instance, if a route doesn't define any method routing rule, exchanges are matched regardless of the method.

The following is an example of a Web route which matches all exchanges, this is the simplest route that can be defined on a router:

```

router
    .route() // 1
        .handler(exchange -> { // 2
            exchange.response()
                .headers(headers ->
                    headers.contentType(MediaType.TEXT_PLAIN)
                )
                .body()
                .encoder()
                .value("Hello, world!");
        });

```

1. A new `WebRouteManager` instance is obtained from the Web router to configure a `WebRoute`
2. We only define the handler of the route as a result any exchange might be routed to that particular route unless a more specific route matching the exchange exists.

A `WebRouter` is exposed in the `web` module and wired to the `http-server` module to override the HTTP server's root handler, it can be configured in a `WebRouterConfigurer` as defined in the [Web Server documentation](#).

An exchange handler can be attached to multiple routes at once by providing multiple routing rules to the route manager, the following example actually results in 8 individual routes being created in the Web router:

```

router
    .route()
        .path("/doc")
        .path("/document")
        .method(Method.GET)
        .method(Method.POST)
        .consumes(MediaType.APPLICATION_JSON)
        .consumes(MediaType.APPLICATION_XML)
        .handler(exchange -> {
            ...
        });

```

The Web routes defined in a Web router can be queried as they are defined by invoking the `findRoutes()` method instead of the `handler()` method on the route manager. The following example select all routes matching `GET` method:

```

Set<WebRoute<WebExchange>> routes = router
    .route()
        .method(Method.GET)
        .findRoutes();

```

It is also possible to enable, disable or remove a set of routes in a similar way:

```
// Disables all GET routes
router
  .route()
    .method(Method.GET)
    .disable();

// Enables all GET routes
router
  .route()
    .method(Method.GET)
    .enable();

// remove all GET routes
router
  .route()
    .method(Method.GET)
    .remove();
```

Individual routes can also be enabled, disabled or removed as follows:

```
// Disables all GET routes producing 'application/json'
router
  .route()
    .method(Method.GET)
    .findRoutes()
    .stream()
    .filter(route -> route.getProduce().equals(MediaType.APPLICATION_JSON))
    .forEach(WebRoute::disable);

// Enables all GET routes producing 'application/json'
router
  .route()
    .method(Method.GET)
    .findRoutes()
    .stream()
    .filter(route -> route.getProduce().equals(MediaType.APPLICATION_JSON))
    .forEach(WebRoute::enable);

// Removes all GET routes producing 'application/json'
router
  .route()
    .method(Method.GET)
    .findRoutes()
    .stream()
    .filter(route -> route.getProduce().equals(MediaType.APPLICATION_JSON))
    .forEach(WebRoute::remove);
```

Path routing rule

The path routing rule matches exchanges whose request targets a specific path or a path that matches against a particular pattern. The path or path pattern of a routing rule must be absolute (ie. start with `/`).

We can for instance define a route to handle all requests to `/bar/foo` as follows:

```
router
  .route()
    .path("/foo/bar")
    .handler(exchange -> {
      ...
    });
```


The route in the preceding example specifies an exact match for the exchange request path, it is also possible to make the route match the path with or without a trailing slash as follows:

```
router
  .route()
    .path("/foo/bar", true)
    .handler(exchange -> {
      ...
    });
```

A path pattern following the [parameterized URIs notation](#) can also be specified to create a routing rule matching multiple paths. This also allows to specify [path parameters](#) that can be retrieved from the [WebExchange](#).

In the following example, the route will match exchanges whose request path is [/book/1](#), [/book/abc...](#) and store the extracted parameter value in path parameter [id](#):

```
router
  .route()
    .path("/book/{id}")
    .handler(exchange -> {
      exchange.request().pathParameters().get("id")...
    });
```

A parameter is matched against a regular expression set to `[^/]*` by default which is why previous route does not match [/book/a/b](#). Parameterized URIs allow to specify the pattern matched by a particular path parameter using `{[<name>][:<pattern>]}` notation, we can then put some constraints on path parameters value. For instance, we can make sure the [id](#) parameter is a number between 1 and 999:

```
router
  .route()
    .path("/book/{id:[1-9][0-9]{0,2}}")
    .handler(exchange -> {
      ...
    });
```

If we just want to match a particular path without extracting path parameters, we can omit the parameter name and simply write:

```
router
  .route()
    .path("/book/{}")
    .handler(exchange -> {
      ...
    });
```

Method routing rule

The method routing rule matches exchanges that have been sent with a particular HTTP method.

In order to handle all [GET](#) exchanges, we can do:

```
router
  .route()
    .method(Method.GET)
    .handler(exchange -> {
      ...
    });
```

Consume routing rule

The consume routing rule matches exchanges whose request body content type matches a particular media range as defined by [RFC 7231 Section 5.3.2](#).

For instance, in order to match all exchanges with a `application/json` request payload, we can do:

```
router
  .route()
    .method(Method.POST)
    .consumes(MediaType.APPLICATION_JSON)
    .handler(exchange -> {
      ...
    });
```

We can also specify a media range to match, for example, all exchanges with a `*/json` request payload:

```
router
  .route()
    .method(Method.POST)
    .consumes("*/json")
    .handler(exchange -> {
      ...
    });
```

The two previous routes are different and as a result they can be both defined in the router, a content negotiation algorithm is used to determine which route should process a particular exchange as defined in [RFC 7231 Section 5.3](#).

Routes are sorted by consumed media ranges as follows:

- quality value is compared first as defined by [RFC7231 Section 5.3.1](#), the default quality value is 1.
- type and subtype wildcards are considered after: `a/b > a/* > */b > */*`
- parameters are considered last, the most precise media range which is the one with the most parameters with values gets the highest priority (eg. `application/json;p1=a;p2=2 > application/json;p1=b > application/json;p1`)

The Web router then selects the first route whose media range matches the request `content-type` header field.

If we consider previous routes, an exchange with a `application/json` request payload will be matched by the first route while an exchange with a `text/json` request will be matched by the second route.

A media range can also be parameterized which allows for interesting setup such as:

```

router
  .route()
    .path("/document")
    .method(Method.POST)
    .consumes("application/json;version=1")
    .handler(exchange -> {
      ...
    });

router
  .route()
    .path("/document")
    .method(Method.POST)
    .consumes("application/json;version=2")
    .handler(exchange -> {
      ...
    });

router
  .route()
    .path("/document")
    .method(Method.POST)
    .consumes("application/json")
    .handler(exchange -> {
      ...
    });

```

In the above example, an exchange with a `application/json;version=1` request payload is matched by the first route, `application/json;version=2` request payload is matched by the second route and any other `application/json` request payload is matched by the third route.

If there is no route matching the content type of a request of an exchange matched by previous routing rules, a (415) unsupported media type error is returned.

As described before, if no route is defined with a consume routing rule, exchanges are matched regardless of the request content type, content negotiation is then eventually delegated to the handler which must be able to process the payload whatever the format.

Produce routing rule

The produce routing rule matches exchanges based on the acceptable media ranges supplied by the client in the `accept` header field of the request as defined by [RFC 7231 Section 5.3.2](#).

A HTTP client (eg. Web browser) typically sends a `accept` header to indicate the server which response media types are acceptable in the response. The Web router determines the best matching route based on the media types produced by the routes matching previous routing rules.

We can for instance define the following routes:

```

router
  .route()
    .path("/doc")
    .produces(MediaType.APPLICATION_JSON)
    .handler(exchange -> {
      ...
    });

router
  .route()
    .path("/doc")
    .produces(MediaType.TEXT_XML)
    .handler(exchange -> {
      ...
    });

```

Now let's consider the following **accept** request header field:

```
accept: application/json, application/xml;q=0.9, */xml;q=0.8
```

This field basically tells the server that the client wants to receive first a **application/json** response payload, if not available a **application/xml** response payload and if not available any ***/xml** response payload.

The content negotiation algorithm is similar as the one described in the [consume routing rule](#), it is simply reversed in the sense that it is the acceptable media ranges defined in the **accept** header field that are sorted and the route producing the media type matching the media range with the highest priority is selected.

Considering previous routes, a request with previous **accept** header field is then matched by the first route.

A request with the following **accept** header field is matched by the second route:

```
accept: application/xml;q=0.9, */xml;q=0.8
```

The exchange is also matched by the second route with the following **accept** header field:

```
accept: application/json;q=0.5, text/xml;q=1.0
```

If there is no route producing a media type that matches any of the acceptable media ranges, then a (406) not acceptable error is returned.

As described before, if no route is defined with a produce routing rule, exchanges are matched regardless of the acceptable media ranges, content negotiation is then eventually delegated to the handler which becomes responsible to return an acceptable response to the client.

Language routing rule

Finally, the language routing rule matches exchanges based on the acceptable languages supplied by client in the `accept-language` header field of the request as defined by [RFC 7231 Section 5.3.5](#).

A HTTP client (eg. Web browser) typically sends a `accept-language` header to indicate the server which languages are acceptable for the response. The Web router determines the best matching route based on the language tags produced by the routes matching previous routing rules.

We can defines the following routes to return a particular resource in English or in French:

```
router
  .route()
    .path("/doc")
    .language("en-US")
    .handler(exchange -> {
      ...
    });

router
  .route()
    .path("/doc")
    .language("fr-FR")
    .handler(exchange -> {
      ...
    });
```

The content negotiation is similar to the one described in the [produce routing rule](#) but using language ranges and language types instead of media ranges and media types. Acceptable language ranges are sorted as follows:

- quality value is compared first as defined by [RFC 7231 Section 5.3.1](#), the default quality value is 1.
- primary and secondary language tags and wildcards are considered after: `fr-FR > fr > *`

The Web router then selects the route whose produced language tag matches the language range with the highest priority.

As for the produce routing rule, if there is no route defined with a language tag that matches any of the acceptable language ranges, then a (406) not acceptable error is returned. However, unlike the produce routing rule, a default route can be defined to handle such unmatched exchanges.

For instance, we can add the following default route to the router:

```
router
  .route()
    .path("/doc")
    .handler(exchange -> {
      ...
    });
```

A request with the following `accept-language` header field is then matched by the default route:

`accept-language: it-IT`

Error routing

An `ErrorRouter` is used to route an `ErrorWebExchange` to the right `ErrorWebExceptionHandler` when an exception is thrown during the normal processing of an exchange, it implements `ExchangeHandler<ErrorExchange<Throwable>>` and it is typically used as error handler in the HTTP server.

Error web exchange

The *web* module API extends the [server exchange API](#) defined in the *http-server* module and defines the server `ErrorWebExceptionHandler` to handle a server `ErrorWebExchange`. These interfaces respectively extends the `ExchangeHandler` and `Exchange` interfaces which are defined in the *http-server* module. An error Web exchange handler is typically attached to one or more error Web routes defined in an `ErrorWebRouter`.

The `ErrorWebExchange` provides a response body encoder which can be used to encode error response body based on the content type specified in the response headers. The usage is exactly the same as for the Web server exchange [response body encoder](#).

```
errorRouter
    .route()
        .error(IllegalArgumentException.class)
        .produces(MediaType.APPLICATION_JSON)
        .handler(errorExchange ->
            errorExchange.response()
                .body()
                .encoder(Message.class)
                .value(new Message("IllegalArgumentException")))
        );
```

Error router

An `ErrorWebRouter` routes an error exchange to an error exchange handler attached to an `ErrorWebRoute` by matching it against a combination of routing rules specified in the route. An error route can combine the following routing rules which are matched in that order: the type of the error, the media ranges and language ranges accepted by the client. For instance, the error router matches an error exchange against the error type routing rule first, then the produce routing rule... Multiples routes can then match a given exchange but only one will be retained to actually process the exchange which is the one matching the highest routing rules.

If a route doesn't define a particular routing rule, the routing rule is simply ignored and matches all exchanges. For instance, if a route doesn't define any error type routing rule, it matches error exchanges regardless of the error.

The following is an example of an error route which matches all error exchanges and as a result handles all types of error:

```

errorRouter
    .route()
        .handler(errorExchange ->
            errorExchange.response()
                .headers(headers ->
                    headers
                        .status(Status.INTERNAL_SERVER_ERROR)
                        .contentType(MediaType.TEXT_PLAIN)
                )
            .body()
            .empty()
        );

```

An `ErrorWebRouter` is exposed in the `web` module and wired to the `http-server` module to override the HTTP server's error handler. It defines error routes to *whitelabel* error handlers for standard `HttpException` as defined by [HTTP base API](#) and producing `application/json` or `text/html` payloads. It can be configured to override these routes or defines others using an `ErrorWebRouterConfigurer` as defined in the [Web Server documentation](#).

The creation, activation, deactivation or removal of routes in an error router is done in the exact same way as for the [Web router](#).

In the following example, we define an error route to handle `SomeCustomException` producing `text/html` response in English for the clients that accept it:

```

errorRouter
    .route()
        .error(SomeCustomException.class)
        .produces(MediaType.TEXT_HTML)
        .language("en-US")
        .handler(errorExchange ->
            ...
        );

```

Error routes can be queried as follows:

```

Set<ErrorWebRoute> errorRoutes = errorRouter
    .route()
        .error(SomeCustomException.class)
        .findRoutes();

```

They can be enabled, disabled or removed as follows:

```
// Disables all SomeCustomException routes
errorRouter
  .route()
    .error(SomeCustomException.class)
    .disable();

// Enables all SomeCustomException routes
errorRouter
  .route()
    .error(SomeCustomException.class)
    .enable();

// remove all SomeCustomException routes
errorRouter
  .route()
    .error(SomeCustomException.class)
    .remove();
```

Individual error routes are enabled, disabled or removed as follows:

```
// Disables all SomeCustomException routes producing 'application/json'
router
  .route()
    .error(SomeCustomException.class)
    .findRoutes()
    .stream()
    .filter(route -> route.getProduce().equals(MediaType.APPLICATION_JSON))
    .forEach(WebRoute::disable);

// Enables all SomeCustomException routes producing 'application/json'
router
  .route()
    .error(SomeCustomException.class)
    .findRoutes()
    .stream()
    .filter(route -> route.getProduce().equals(MediaType.APPLICATION_JSON))
    .forEach(WebRoute::enable);

// Removes all SomeCustomException routes producing 'application/json'
router
  .route()
    .error(SomeCustomException.class)
    .findRoutes()
    .stream()
    .filter(route -> route.getProduce().equals(MediaType.APPLICATION_JSON))
    .forEach(WebRoute::remove);
```

Error type routing rule

The error type routing rule matches error exchanges whose error is of a particular type.

For instance, in order to handle all error exchanges whose error is an instance of `SomeCustomException`, we can do:

```
errorRouter
  .route()
    .method(SomeCustomException.class)
    .handler(exchange -> {
      ...
    });
```


Produce routing rule

The produce routing rule, when applied to an error route behaves exactly the same as for a [Web route](#). It allows to define error handlers that produce responses of different types based on the set of media range accepted by the client.

This is particularly useful to returned specific error responses to a particular client in a particular context. For instance, a backend application might want to receive errors in a parseable format like `application/json` whereas a Web browser might want to receive errors in a human readable format like `text/html`.

Language routing rule

The language routing rule, when applied to an error route behaves exactly the same as for a [Web route](#). It allows to define error handlers that produce responses with different languages based on the set of language range accepted by the client fallbacking to the default route when content negotiation did not give any match.

Static handler

The `StaticHandler` is a specific `WebExchangeHandler<WebExchange>` implementation that can be used to define routes for serving static resources resolved with the [Resource API](#).

For instance, we can create a route to serve files stored in a `web-root` directory as follows:

```
router
  .route()
    .path("/static/{path:. *}") // 1
    .handler(new StaticHandler(new FileResource("web-root/"))) // 2
```

1. The path must be parameterized with a `path` parameter which can include `/`, for the static handler to be able to determine the relative path of the resource in the `web-root` directory
2. The base resource is defined directly as a `FileResource`, although it is also possible to use a `ResourceService` to be more flexible in terms of the kind of resource

The static handler relies on the resource abstraction to resolve resources, as a result, these can be located on the file system, on the class path, on the module path...

The static handler also looks for a welcome page when a directory resource is requested. For instance considering the following `web-root` directory:

```
web-root/
├── index.html
└── snowflake.svg
```

A request to `http://127.0.0.1/static/` would return the `index.html` file.

Web Server

The *web* module composes the *http-server* module and as a result it requires a *NetService* and a *ResourceService*. A set of [media type converters](#) is also required for message payload conversion. All these services are provided by the *boot* module, so one way to create an application with a Web server is to create a Winter module composing *boot* and *web* modules.

```
@io.winterframework.core.annotation.Module
module io.winterframework.example.app_web {
    requires io.winterframework.mod.boot;
    requires io.winterframework.mod.web;
}
```

The resulting *app_web* module, thus created, can then be started as an application as follows:

```
package io.winterframework.example.app_web;

import io.winterframework.core.v1.Application;

public class Main {

    public static void main(String[] args) {
        Application.with(new App_web.Builder()).run();
    }
}
```

The above example starts a Web server using default configuration which is a HTTP/1.x server with a Web router as root handler and an error router as error handler.

404 Not Found

```
io.winterframework.mod.web.internal.RouteNotFoundException
    at io.winterframework.mod.web@1.0.0-SNAPSHOT/io.winterframework.mod.web.internal.HandlerRoutingLink.handle(HandlerRoutingLink.java:98)
    at io.winterframework.mod.web@1.0.0-SNAPSHOT/io.winterframework.mod.web.internal.LanguageRoutingLink.handle(LanguageRoutingLink.java:177)
    at io.winterframework.mod.web@1.0.0-SNAPSHOT/io.winterframework.mod.web.internal.ProducesRoutingLink.handle(ProducesRoutingLink.java:177)
    at io.winterframework.mod.web@1.0.0-SNAPSHOT/io.winterframework.mod.web.internal.ConsumesRoutingLink.handle(ConsumesRoutingLink.java:164)
    at io.winterframework.mod.web@1.0.0-SNAPSHOT/io.winterframework.mod.web.internal.MethodRoutingLink.handle(MethodRoutingLink.java:158)
    at io.winterframework.mod.web@1.0.0-SNAPSHOT/io.winterframework.mod.web.internal.PathPatternRoutingLink.handle(PathPatternRoutingLink.java:153)
    at io.winterframework.mod.web@1.0.0-SNAPSHOT/io.winterframework.mod.web.internal.PathRoutingLink.handle(PathRoutingLink.java:160)
    at io.winterframework.mod.web@1.0.0-SNAPSHOT/io.winterframework.mod.web.internal.GenericWebRouter.handle(GenericWebRouter.java:187)
    at io.winterframework.mod.http.server@1.0.0-SNAPSHOT/io.winterframework.mod.http.server.internal.AbstractExchange.start(AbstractExchange.java:148)
    at io.winterframework.mod.http.server@1.0.0-SNAPSHOT/io.winterframework.mod.http.server.internal.Http1x.Http1xChannelHandler.channelRead(Http1xChannelHandler.java:379)
    at io.netty.transport@4.1.59.Final/io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(AbstractChannelHandlerContext.java:379)
    at io.netty.transport@4.1.59.Final/io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(AbstractChannelHandlerContext.java:365)
    at io.netty.transport@4.1.59.Final/io.netty.channel.AbstractChannelHandlerContext.fireChannelRead(AbstractChannelHandlerContext.java:357)
    at io.netty.codec@4.1.59.Final/io.netty.handler.codec.ByteToMessageDecoder.fireChannelRead(ByteToMessageDecoder.java:324)
    at io.netty.codec@4.1.59.Final/io.netty.handler.codec.ByteToMessageDecoder.channelRead(ByteToMessageDecoder.java:296)
    at io.netty.transport@4.1.59.Final/io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(AbstractChannelHandlerContext.java:379)
    at io.netty.transport@4.1.59.Final/io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(AbstractChannelHandlerContext.java:365)
    at io.netty.transport@4.1.59.Final/io.netty.channel.AbstractChannelHandlerContext.fireChannelRead(AbstractChannelHandlerContext.java:357)
    at io.netty.transport@4.1.59.Final/io.netty.channel.DefaultChannelPipeline$HeadContext.channelRead(DefaultChannelPipeline.java:1410)
    at io.netty.transport@4.1.59.Final/io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(AbstractChannelHandlerContext.java:379)
    at io.netty.transport@4.1.59.Final/io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(AbstractChannelHandlerContext.java:365)
    at io.netty.transport@4.1.59.Final/io.netty.channel.DefaultChannelPipeline.fireChannelRead(DefaultChannelPipeline.java:919)
    at io.netty.transport@4.1.59.Final/io.netty.channel.nio.AbstractNioByteChannel$NioByteUnsafe.read(AbstractNioByteChannel.java:166)
    at io.netty.transport@4.1.59.Final/io.netty.channel.nio.NioEventLoop.processSelectedKey(NioEventLoop.java:719)
    at io.netty.transport@4.1.59.Final/io.netty.channel.nio.NioEventLoop.processSelectedKeysOptimized(NioEventLoop.java:655)
    at io.netty.transport@4.1.59.Final/io.netty.channel.nio.NioEventLoop.processSelectedKeys(NioEventLoop.java:581)
    at io.netty.transport@4.1.59.Final/io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:493)
    at io.netty.common@4.1.59.Final/io.netty.util.concurrent.SingleThreadEventExecutor$4.run(SingleThreadEventExecutor.java:989)
    at io.netty.common@4.1.59.Final/io.netty.util.concurrent.ThreadExecutorMap$2.run(ThreadExecutorMap.java:74)
    at io.netty.common@4.1.59.Final/io.netty.util.concurrent.FastThreadLocalRunnable.run(FastThreadLocalRunnable.java:30)
    at java.base/java.lang.Thread.run(Thread.java:831)
```

This is a whitelabel error page, providing a custom error handler is recommended.

Configuration

The Web server configuration is done in the the *web* module configuration [WebConfiguration](#) which includes the *http-server* module configuration [HttpServerConfiguration](#). As for the *http-server* module, the net service configuration can also be considered to set low level network configuration in the *boot* module.

The following configuration can then be created in the *app_http* module:

Let's create the following configuration in the *app_web* module:

```
package io.winterframework.example.app_web;

import io.winterframework.core.annotation.NestedBean;
import io.winterframework.mod.boot.NetConfiguration;
import io.winterframework.mod.configuration.Configuration;
import io.winterframework.mod.web.WebConfiguration;

@Configuration
public interface App_webConfiguration {

    @NestedBean
    NetConfiguration net();

    @NestedBean
    WebConfiguration web();
}
```

The Web server can then be configured. For instance, we can enable HTTP/2 over cleartext, TLS, HTTP compression... as described in the [http-server module documentation](#).

For instance:

```

package io.winterframework.example.app_web;

import io.winterframework.core.v1.Application;

public class Main {

    public static void main(String[] args) {
        Application.with(new App_web.Builder()
            .setApp_webConfiguration(
                App_webConfigurationLoader.load(configuration -> configuration
                    .web(web -> web
                        .http_server(server -> server
                            .server_port(8081)
                            .h2c_enabled(true)
                            .server_event_loop_group_size(4)
                        )
                    )
                )
            ).run();
    }
}

```

Configuring the Web router

As explained before, a [Web router](#) is used to route a request to the right handler based on a set of rules defined in a route. The *web* module provides a **WebRouter** bean which is wired to the *http-server* module to override the default root handler and handle all incoming requests to the server. Unlike the root handler in the *http-server* module, this bean is not overridable but it can be configured in order to define Web routes for the *app_web* module.

This can be done by defining a **WebRouterConfigurer** bean in the *app_web* module. A Web router configurer is basically a **Consumer<WebRouter>** invoked after the initialization of the Web router and more precisely after the default configuration has been applied.

Using what we've learned from the [Web routing documentation](#), routes can then be defined as follows:

```

package io.winterframework.example.app_web;

import io.winterframework.core.annotation.Bean;
import io.winterframework.mod.base.resource.MediaTypees;
import io.winterframework.mod.web.WebExchange;
import io.winterframework.mod.web.WebRouter;
import io.winterframework.mod.web.WebRouterConfigurer;

@Bean
public class App_webWebRouterConfigurer implements WebRouterConfigurer<WebExchange> {

    @Override
    public void accept(WebRouter<WebExchange> router) {
        router
            .route()
                .path("/hello")
                .produces(MediaTypees.TEXT_PLAIN)
                .language("en-US")
                .handler(exchange -> exchange
                    .response()
                        .body()
                        .encoder(String.class)
                        .value("Hello!")
                    )
            .route()
                .path("/hello")
                .produces(MediaTypees.TEXT_PLAIN)
                .language("fr-FR")
                .handler(exchange -> exchange
                    .response()
                        .body()
                        .encoder(String.class)
                        .value("Bonjour!")
                    );
    }
}

```

Now we can test the application:

```

$ curl -i http://localhost:8080/
HTTP/1.1 404 Not Found
content-length: 0

```

```

$ curl -i http://localhost:8080/hello
HTTP/1.1 200 OK
content-type: text/plain
content-length: 6

```

Hello!

```

$ curl -i -H 'accept-language: fr' http://localhost:8080/hello
HTTP/1.1 200 OK
content-type: text/plain
content-length: 8

```

Bonjour!

```

$ curl -i -H 'accept: application/json' http://localhost:8080/hello
HTTP/1.1 406 Not Acceptable
content-type: application/json
content-length: 81

```

```

{"status":"406","path":"/hello","error":"Not Acceptable","accept":["text/plain"]}

```

Configuring the error router

The *web* module also provides an error router bean wired to the *http-server* module to override the default error handler and handle all errors thrown when processing an exchange. As a reminder, an [error router](#) is used to route an error exchange to the right error handler based on a set of rules defined in an error route.

As for the Web router, the error router can't be overridden, it is rather configured by defining an *ErrorWebRouterConfigurer* bean invoked after the initialization of the error router and more precisely after the default configuration has been applied which means default error routes remain although they can be overridden in the configurer.

The error router implemented in the *web* module provides default error routes for handling base *HttpException*, whitelabel error pages are returned in particular when a client request *text/html* responses. More generally, *Throwable* errors are also handled by default, returning a generic (500) internal server error.

Now let's assume, we have created a route which might throw a particular custom exception for which we want to return a particular response. For instance, we might have defined the following route in the *App_webWebRouterConfigurer*:

```
@Bean
public class App_webWebRouterConfigurer implements WebRouterConfigurer<WebExchange> {

    @Override
    public void accept(WebRouter<WebExchange> router) {
        router
            ...
            .route()
                .path("custom_exception")
                .handler(exchange -> {
                    throw new SomeCustomException();
                });
    }
}
```

Note that an exchange handler is defined to throw checked *HttpException* only which actually makes sense since a Http error is eventually what will be returned to the client. In our example, the *SomeCustomException* is then either unchecked or extends *HttpException*.

Now using what we've learned from the [error routing documentation](#), we can define an error route to handle that particular exception as follows:

```

package io.winterframework.example.app_web;

import io.winterframework.core.annotation.Bean;
import io.winterframework.mod.base.resource.MediaTypees;
import io.winterframework.mod.http.base.Status;
import io.winterframework.mod.web.ErrorWebRouter;
import io.winterframework.mod.web.ErrorWebRouterConfigurer;

@Bean
public class App_webErrorWebRouterConfigurer implements ErrorWebRouterConfigurer {

    @Override
    public void accept(ErrorWebRouter errorRouter) {
        errorRouter
            .route()
                .error(SomeCustomException.class)
                .handler(errorExchange -> errorExchange
                    .response()
                        .headers(headers -> headers
                            .status(Status.BAD_REQUEST)
                            .contentType(MediaTypees.TEXT_PLAIN)
                        )
                    .body()
                    .encoder()
                    .value("A custom exception was raised")
                );
    }
}

```

Now if we hit `http://localhost:8080/custom_exception` we should receive a custom error:

```

$ curl -i http://localhost:8080/custom_exception
HTTP/1.1 400 Bad Request
content-type: text/plain
content-length: 29

```

A custom exception was raised

WebJars

The Web server can also be configured to automatically serve static resources from [WebJars dependencies](#) using a version agnostic path: `/webjars/{webjar_module}/{path:. *}` where `{webjar_module}` corresponds to the *modularized* name of the WebJar minus `org.webjars`. For example the location of the Swagger UI WebJar would be `/webjars/swagger.ui/`.

This feature can be activated with the following configuration:


```

package io.winterframework.example.app_web;

import io.winterframework.core.v1.Application;

public class Main {

    public static void main(String[] args) {
        Application.with(new App_web.Builder()
            .setApp_webConfiguration(
                App_webConfigurationLoader.load(configuration -> configuration
                    .web(web -> web
                        .enable_webjars(true)
                    )
                )
            )
        ).run();
    }
}

```

Then we can declare WebJars dependencies such as the Swagger UI in the build descriptor:

```

<project>
  <dependencies>
    <dependency>
      <groupId>org.webjars</groupId>
      <artifactId>swagger-ui</artifactId>
    </dependency>
  </dependencies>
</project>

```

The Swagger UI should now be accessible at <http://localhost:8080/webjars/swagger-ui/>.

Sadly WebJars are not modular JARs, they are not even named modules which causes several issues when dependencies are specified on the module path. That's why when an application is run or packaged using [Winter tools](#), such dependencies and WebJars in particular are *modularized*. A WebJar such as `swagger-ui` is modularized into `org.webjars.swagger.ui` module which explains why it is referred by its module name: `swagger.ui` in the WebJars resource path (the `org.webjars` part is omitted since the context is known).

When running a fully modular Winter application, *modularized* WebJars modules must be added explicitly to the JVM using the `--add-modules` option, otherwise they are not resolved when the JVM starts. For instance:

```
$ java --add-modules org.webjars.swagger.ui ...
```

Hopefully, the Winter Maven plugin adds unnamed modules by default when running or packaging an application, so you shouldn't have to worry about it. The following command automatically adds the unnamed modules when running the JVM:

```
$ mvn winter:run
```

This can be disabled in order to manually control which modules should be added:

```
$ mvn winter:run -Dwinter.exec.addUnnamedModules=false -Dwinter.exec.vmOptions="--add-modules org.webjars.swagger.ui"
```

It might also be possible to define the dependency in the module descriptor, unfortunately since WebJars modules are unnamed, they are named against the name of the JAR file which is greatly unstable and can't be trusted, so previous approach is by far the safest. If you need to create a Webjar you should make it a named module with the `Automatic-Module-Name` attribute sets to `org.webjars.{webjar_module}` in the manifest file and with resources located under `META-INF/resources/webjars/{webjar_module}/{webjar_version}/`.

Note that when the application is run with non-modular WebJars specified on the class path, they can be accessed without any particular configuration as part of the UNNAMED module using the same path notation.

OpenAPI specification

The Web server can also be configured to expose [OpenAPI specifications](#) defined in `/META-INF/winter/web/openapi.yml` resource in application modules.

This feature can be activated with the following configuration:

```
package io.winterframework.example.app_web;

import io.winterframework.core.v1.Application;

public class Main {

    public static void main(String[] args) {
        Application.with(new App_web.Builder()
            .setApp_webConfiguration(
                App_webConfigurationLoader.load(configuration -> configuration
                    .web(web -> web
                        .enable_open_api(true)
                    )
                )
            )
        ).run();
    }
}
```

When the server starts, it will scan for OpenAPI specifications files `/META-INF/winter/web/openapi.yml` in the application modules and configure the following routes:

- `/open-api` returning the list of available OpenAPI specifications in `application/json`
- `/open-api/{moduleName}` returning the OpenAPI specifications defined for the specified module name or (404) not found error if there is no OpenAPI specification defined in the module or no module with that name.

If the server is also configured to serve [WebJars](#), previous routes can also be used to display OpenAPI specifications in a [Swagger UI](#) assuming the Swagger UI Webjar dependency is present:

```

<project>
  <dependencies>
    <dependency>
      <groupId>org.webjars</groupId>
      <artifactId>swagger-ui</artifactId>
      <version>3.46.0</version>
    </dependency>
  </dependencies>
</project>

```

OpenAPI specifications are usually automatically generated by the Web Winter compiler plugin for routes defined in a [Web controller](#) but you can provide manual or generated specifications using the tool of your choice, as long as it is not conflicting with the Web compiler plugin.

Web Controller

The [Web router](#) and [Web server](#) documentations describe a *programmatic* way of defining the Web routes of a Web server but the *web* module API also provides a set of annotations for defining Web routes in a more declarative way.

A **Web controller** is a regular module bean annotated with `@WebController` which defines methods annotated with `@WebRoute` describing Web routes. These beans are scanned at compile time by the Web Winter compiler plugin in order to generate a single `WebRouterConfigurer` bean configuring the routes in the Web router.

For instance, we can create a book resource with basic CRUD operations, to do so we must define a `Book` model in a dedicated `*.dto` package, we'll see later why this matters:

```

package io.winterframework.example.app_web.dto;

public class Book {

    private String isbn;
    private String title;
    private String author;
    private int pages;

    // Constructor, getters, setters, hashCode, equals...
}

```

Now we can define a `BookResource` Web controller as follows:

```

package io.winterframework.example.app_web;

import java.util.Set;

import io.winterframework.core.annotation.Bean;
import io.winterframework.example.app_web.dto.Book;
import io.winterframework.mod.base.resource.MediaTypees;
import io.winterframework.mod.http.base.Method;
import io.winterframework.mod.web.annotation.Body;
import io.winterframework.mod.web.annotation.PathParam;
import io.winterframework.mod.web.annotation.WebController;
import io.winterframework.mod.web.annotation.WebRoute;

@Bean // 1
@WebController( path = "/book" ) // 2
public class BookResource {

    @WebRoute( method = Method.POST, consumes = MediaTypees.APPLICATION_JSON ) // 3
    public void create(@Body Book book) { // 4
        ...
    }

    @WebRoute( path = "/{isbn}", method = Method.PUT, consumes = MediaTypees.APPLICATION_JSON )
    public void update(@PathParam String isbn, @Body Book book) {
        ...
    }

    @WebRoute( method = Method.GET, produces = MediaTypees.APPLICATION_JSON )
    public Set<Book> list() {
        ...
    }

    @WebRoute( path = "/{isbn}", method = Method.GET, produces = MediaTypees.APPLICATION_JSON )
    public Book get(@PathParam String isbn) {
        ...
    }

    @WebRoute( path = "/{isbn}", method = Method.DELETE )
    public void delete(@PathParam String isbn) {
        ...
    }
}

```

Implementations details have been omitted for clarity, here is what's important:

1. A Web controller must be a module bean because it will be wired into the generated Web router configurer and used to invoke the right handler method attached to a Web route. Besides it is anyway convenient for implementation, a repository could be wired into the **BookResource** bean for instance.
2. The **@WebController** annotation tells the Web compiler plugin to process the bean as a Web controller. The controller root path can also be specified in the annotation, if not specified it defaults to **/** which is the root path of the Web server.
3. The **@WebRoute** annotation on a method tells the Web compiler plugin to define a route whose handler invokes that method. The set of routing rules (ie. path, method, consume, produce, language) describing the route are specified in the annotation.
4. Parameters and request body are specified as method parameters annotated with **@CookieParam**, **@FormParam**, **@HeaderParam**, **@PathParam**, **@QueryParam** and **@Body** annotations.

Some other contextual objects like the underlying **webExchange** can also be injected in the Web controller method.

Assuming we have provided proper implementations to create, update, list, get and delete a book in a data store, we can compile the module. A new module bean `io.winterframework.example.app_web.WebRouterConfigurer` implementing `WebRouterConfigurer` should have been generated in `target/generated-sources/annotations`. It basically configures the routes corresponding to the Web controller's annotated methods in the Web router. This class uses the APIs described before and it is perfectly readable and debuggable and above all it eliminates the overhead of resolving Web controllers or Web routes at runtime.

Now let's go back to the `Book` DTO, we said earlier that it must be created in a dedicated package, the reason is actually quite simple. Since above routes consume and produce `application/json` payloads, the `application/json` media type converter will be invoked to convert `Book` objects from/to JSON data. This converter uses an `ObjectMapper` object from module `com.fasterxml.jackson.databind` which uses reflection to instantiate the objects and populate them from a parsed JSON tree. Unfortunately or hopefully the Java modular system prevents unauthorized reflective access and as a result the `ObjectMapper` can't access the `Book` class unless we explicitly export the package containing DTOs to module `com.fasterxml.jackson.databind` in the module descriptor as follows:

```
module io.winterframework.example.app_web {
    ...
    exports io.winterframework.example.app_web.dto to com.fasterxml.jackson.databind;
}
```

Using a dedicated package for DTOs allows then to limit and control the access to the module classes.

If you're not familiar with the Java modular system and used to Java 8<, you might find this a bit distressing but if you want to better structure and secure your applications, this is the way.

We can now run the application and test the book resource:

```
$ curl -i http://localhost:8080/book
HTTP/1.1 200 OK
content-type: application/json
content-length: 2
```

```
[]
```

```
$ curl -i -X POST -H 'content-type: application/json' -d '{"isbn":"978-0132143011","title":"Distributed Systems: Concepts and Design","author":"George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair","pages":1080}' http://localhost:8080/book
HTTP/1.1 200 OK
content-length: 0
```

```
$ curl -i http://localhost:8080/book
HTTP/1.1 200 OK
content-type: application/json
content-length: 163
```

```
[{"isbn":"978-0132143011","title":"Distributed Systems: Concepts and Design","author":"George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair","pages":1080}]
```

```
$ curl -i http://localhost:8080/book/978-0132143011
HTTP/1.1 200 OK
content-type: application/json
content-length: 161
```

```
{"isbn":"978-0132143011","title":"Distributed Systems: Concepts and Design","author":"George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair","pages":1080}
```

If you have carefully followed the *web* module documentation, you should have noticed that we have previously created a Web router configurer bean in the *app_web* module which is indeed conflicting with the generated Web router configurer resulting in the following self-explanatory compilation error:

```
[ERROR] /home/jkuhn/Devel/git/frmk/io.winterframework.example.app_web/src/main/java/module-info.java:[4,1] Multiple beans matching socket io.winterframework.mod.web:webRouterConfigurer were found
- io.winterframework.example.app_web:app_webWebRouterConfigurer of type
io.winterframework.example.app_web.App_webWebRouterConfigurer
- io.winterframework.example.app_web:webRouterConfigurer of type
io.winterframework.example.app_web.WebRouterConfigurer
```

Consider specifying an explicit wiring in module `io.winterframework.example.app_web` (eg. `@io.winterframework.core.annotation.Wire(beans="io.winterframework.example.app_web:app_webWebRouterConfigurer", into="io.winterframework.mod.web:webRouterConfigurer")`)

In order to resolve that conflict, you can remove the first router configurer or define an explicit wire in the module definition:

```
@io.winterframework.core.annotation.Module
@io.winterframework.core.annotation.Wire(beans="io.winterframework.example.app_web:webRouterConfigurer", into="io.winterframework.mod.web:webRouterConfigurer")
module io.winterframework.example.app_web {
    ...
}
```

Such situation can also occur when you are composing multiple modules defining Web controller beans and thus exposing multiple Web router configurers in the module. Hopefully it is safe to resolve these conflicts by wiring the Web router configurer of the module composing the *web* module as it aggregates all Web router configurers annotated with `@WebRoutes`. Please look at [Composite Web module documentation](#) for further details.

It is possible to separate the API from the implementation by defining the Web controller and the Web routes in an interface implemented in a module bean. For instance,

```

package io.winterframework.example.app_web;

import java.util.Set;

import io.winterframework.example.app_web.dto.Book;
import io.winterframework.mod.base.resource.MediaTypees;
import io.winterframework.mod.http.base.Method;
import io.winterframework.mod.web.annotation.Body;
import io.winterframework.mod.web.annotation.PathParam;
import io.winterframework.mod.web.annotation.WebController;
import io.winterframework.mod.web.annotation.WebRoute;

@WebController(path = "/book")
public interface BookResource {

    @WebRoute(method = Method.POST, consumes = MediaTypees.APPLICATION_JSON)
    void create(@Body Book book);

    @WebRoute(path =("/{isbn}", method = Method.PUT, consumes = MediaTypees.APPLICATION_JSON)
    void update(@PathParam String isbn, @Body Book book);

    @WebRoute(method = Method.GET, produces = MediaTypees.APPLICATION_JSON)
    Set<Book> list();

    @WebRoute(path =("/{isbn}", method = Method.GET, produces = MediaTypees.APPLICATION_JSON)
    Book get(@PathParam String isbn);

    @WebRoute(path =("/{isbn}", method = Method.DELETE)
    void delete(@PathParam String isbn);
}

```

This provides better modularity allowing to define the API in a dedicated module which can later be used to implement various server and/or client implementations in different modules. Another advantage is that it allows to split a Web controller interface into multiple interfaces.

Generics are also supported, we can for instance create a generic **CRUD<T>** interface as follows:

```

package io.winterframework.example.app_web;

import java.util.Set;

import io.winterframework.mod.base.resource.MediaTypees;
import io.winterframework.mod.http.base.Method;
import io.winterframework.mod.web.annotation.Body;
import io.winterframework.mod.web.annotation.PathParam;
import io.winterframework.mod.web.annotation.WebRoute;

public interface CRUD<T> {

    @WebRoute(method = Method.POST, consumes = MediaTypees.APPLICATION_JSON)
    void create(@Body T resource);

    @WebRoute(path =("/{id}", method = Method.PUT, consumes = MediaTypees.APPLICATION_JSON)
    void update(@PathParam String id, @Body T resource);

    @WebRoute(method = Method.GET, produces = MediaTypees.APPLICATION_JSON)
    Set<T> list();

    @WebRoute(path =("/{id}", method = Method.GET, produces = MediaTypees.APPLICATION_JSON)
    T get(@PathParam String id);

    @WebRoute(path =("/{id}", method = Method.DELETE)
    void delete(@PathParam String id);
}

```

And then create specific resources sharing the same interface:

```

package io.winterframework.example.app_web;

import java.util.Set;

import io.winterframework.example.app_web.dto.Book;
import io.winterframework.mod.web.annotation.WebController;

@WebController(path = "/book")
public interface BookResource extends CRUD<Book> {

    void create(Book book);

    void update(String id, Book book);

    Set<Book> list();

    Book get(String id);

    void delete(String id);
}

```

The book resource as we defined it doesn't seem very reactive, this statement is both true and untrue: the API and the Web server are fully reactive, as a result Web routes declared in the book resource Web controller are configured using a reactive API in the Web router configurer, nonetheless the methods in the Web controller are not reactive.

Luckily, we can transform previous interfaces to make them fully reactive:


```

package io.winterframework.example.app_web;

import io.winterframework.mod.base.resource.MediaTypees;
import io.winterframework.mod.http.base.Method;
import io.winterframework.mod.web.annotation.Body;
import io.winterframework.mod.web.annotation.PathParam;
import io.winterframework.mod.web.annotation.WebRoute;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

public interface CRUD<T> {

    @WebRoute(method = Method.POST, consumes = MediaTypees.APPLICATION_JSON)
    Mono<Void> create(@Body Mono<T> resource);

    @WebRoute(path =("/{id}", method = Method.PUT, consumes = MediaTypees.APPLICATION_JSON)
    Mono<Void> update(@PathParam String id, @Body Mono<T> resource);

    @WebRoute(method = Method.GET, produces = MediaTypees.APPLICATION_JSON)
    Flux<T> list();

    @WebRoute(path =("/{id}", method = Method.GET, produces = MediaTypees.APPLICATION_JSON)
    Mono<T> get(@PathParam String id);

    @WebRoute(path =("/{id}", method = Method.DELETE)
    Mono<Void> delete(@PathParam String id);
}

```

There is one remaining thing to do to make the book resource a proper REST resource. When creating a book we must return a 201 Created HTTP code with a **location** header as defined by [RFC7231 Section 7.1.2](#). This can be done by injecting the **WebExchange** in the **create()** method:

```

public interface CRUD<T> {

    @WebRoute(method = Method.POST, consumes = MediaTypees.APPLICATION_JSON, produces =
MediaTypees.APPLICATION_JSON)
    Mono<Void> create(@Body Mono<T> resource, WebExchange exchange);
    ...
}

```

We can then do the following in the book resource implementation class:

```

package io.winterframework.example.app_web;

import io.winterframework.core.annotation.Bean;
import io.winterframework.example.app_web.dto.Book;
import io.winterframework.mod.http.base.Status;
import io.winterframework.mod.http.base.header.Headers;
import io.winterframework.mod.web.WebExchange;
import reactor.core.publisher.Mono;

@Bean
public class BookResourceImpl implements BookResource {

    @Override
    public Mono<Void> create(Mono<Book> book, WebExchange exchange) {
        ...
        exchange.response().headers(headers -> headers
            .status(Status.CREATED)
            .add(Headers.NAME_LOCATION,
exchange.request().getPathBuilder().segment(b.getIsbn()).buildPath())
        );
        ...
    }
    ...
}

```

Now if we run the application and create a book resource we should get the following:

```

$ curl -i -X POST -H 'content-type: application/json' -d '{"isbn":"978-0132143011","title":"Distributed Systems: Concepts and Design","author":"George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair","pages":1080}' http://localhost:8080/book
HTTP/1.1 201 Created
content-type: application/json
location: /book/978-0132143012
content-length: 0

```

Web route

So far, we have described a concrete Web controller use case which should already give a good idea of what can be done with the *web* module. Now, let's examine in details how a Web route is declared in a Web controller.

A Web route or HTTP endpoint or REST endpoint... in short an HTTP request/response exchange is essentially defined by:

- an input, basically an HTTP request characterized by the following components: path, method, query parameters, headers, cookies, path parameters, request body
- a normal output, basically a successful HTTP response and more especially: status (2xx or 3xx), headers, response body
- a set of error outputs, basically unsuccessful HTTP responses and more especially: status (4xx or 5xx), headers, response body

Web routes are defined as methods in a Web controller which match this definition: the Web route input is defined in the parameters of the method, the Web route normal output is defined by the return type of the method and finally the exceptions thrown by the method define the Web route error outputs.

It then remains to bind the Web route semantic to the method, this is done within various annotations on the method and its parameters.

Routing rules

Routing rules, as defined in the [Web router documentation](#), are specified in a single `@WebRoute` annotation on a Web controller method. It allows to define the paths, the methods, the consumed media ranges, the produced media types and the produced languages of the Web routes that route a matching request to the handler implemented in the method.

For instance, we can define multiple paths and/or multiple produced media types in order to expose a resource at different locations in different formats:

```
@WebRoute( path = { "/book/current", "/book/v1" }, produces = { MediaType.APPLICATION_JSON,
    MediaType.APPLICATION_XML } )
Flux<T> list();
```

Note that this exactly corresponds to the *web* module API

The `matchTrailingSlash` parameter can be used to indicate that the defined path should be matched taking the trailing slash into account or not.

Parameter bindings

As stated above, a `@WebRoute` annotated method must be bound to the Web exchange. In particular, method parameters are bound to the various elements of the request using `*Param` annotations defined in the *web* module API.

Such parameters can be of any type, as long as the parameter converter plugged into the *web* module can convert it, otherwise a `ConverterException` is thrown. The default parameter converter provided in the *boot* module is able to convert primitive and common types including arrays and collections. Please refer to the [HTTP server documentation](#) to learn how to extend the parameter converter to convert custom types.

In the following example, the value or values of query parameter `isbn`s is converted to an array of strings:

```
@WebRoute( path = { "/book/byIsbn" }, produces = { MediaType.APPLICATION_JSON } )
Flux<T> getBooksByIsbn(@QueryParam String[] isbn);
```

If the above route is queried with `/book/byIsbn?isbn=978-0132143011,978-0132143012,978-0132143013&isbn=978-0132143014` the `isbn`s parameter is then: `["978-0132143011", "978-0132143012", "978-0132143013", "978-0132143014"]`.

A parameter defined like this is required by default and a `MissingRequiredParameterException` is thrown if one or more parameters are missing from the request but it can be declared as optional by defining it as an `Optional<T>`:

In the following example, query parameter `limit` is optional and no exception will be thrown if it is missing from the request:

```
@WebRoute( path = { "/book" }, produces = { MediaType.APPLICATION_JSON } )
Flux<T> getBooks(@QueryParam Optional<Integer> limit);
```

Query parameter

Query parameters are declared using the `@QueryParam` annotation as follows:

```
@WebRoute( path = { "/book/byIsbn" }, produces = { MediaType.APPLICATION_JSON } )  
Flux<T> getBooksByIsbn(@QueryParam String[] isbnns);
```

Note that the name of the method parameter actually specifies the name of the query parameter.

This contrasts with other RESTful API, such as JAX-RS, which requires to specify the parameter name, again, in the annotation. Since the Web Winter compiler plugin works at compile time, it has access to actual method parameter names defined in the source.

Path parameter

Path parameters are declared using the `@PathParam` annotation as follows:

```
@WebRoute(path =("/{id}", method = Method.GET, produces = MediaType.APPLICATION_JSON)  
Mono<T> get(@PathParam String id);
```

Note that the name of the method parameter must match the name of the path parameter of the route path defined in the `@WebRoute` annotation.

Cookie parameter

It is possible to bind cookie values as well using the `@CookieParam` annotation as follows:

```
@WebRoute(method = Method.POST, consumes = MediaType.APPLICATION_JSON)  
Mono<Void> create(@CookieParam String book_store, @Body Mono<T> book, WebExchange exchange);
```

In previous example, the route must be queried with a `book_store` cookie:

```
$ curl -i -X POST -H 'cookie: book_store=store1' -H 'content-type: application/json' -d '...'  
http://localhost:8080/book  
...
```

Header parameter

Header field can also be bound using the `@HeaderParam` annotation as follows:

```
@WebRoute(method = Method.GET, produces = MediaType.APPLICATION_JSON)  
Flux<T> list(@HeaderParam Optional<Format> format);
```

In previous example, the `Format` type is an enumeration indicating how book references must be returned (eg. `SHORT`, `FULL`...), a `format` header may or may not be added to the request since it is declared as optional:

```
$ curl -i -H 'format: SHORT' http://localhost:8080/book  
...
```

Form parameter

Form parameters are bound using the `@FormParam` annotation as follows:

```
@WebRoute(method = Method.POST, consumes = MediaType.APPLICATION_X_WWW_FORM_URLENCODED)
Mono<Void> createAuthor(
    @FormParam String forename,
    @FormParam Optional<String> middlename,
    @FormParam String surname,
    @FormParam LocalDate birthdate,
    @FormParam Optional<LocalDate> deathdate,
    @FormParam String nationality);
```

Form parameters are sent in a request body using `application/x-www-form-urlencoded` format as defined by [living standard](#). They can be sent using a HTML form submitted to the server resulting in the following request body:

```
forename=Leslie,middlename=B.,surname=Lamport,birthdate=19410207,nationality=US
```

Previous route can then be queried as follows:

```
$ curl -i -X POST -H 'content-type:application/x-www-form-urlencoded' -d
'forename=Leslie,middlename=B.,surname=Lamport,birthdate=19410207,nationality=US'
http://localhost:8080/author
```

Form parameters results from the parsing of the request body and as such, `@FormParam` annotations can't be used together with `@Body` on route method parameters.

Contextual parameters

A contextual parameter is directly related to the context into which an exchange is processed in the route method, it can be injected in the route method by specifying a method parameter of a supported contextual parameter type.

Exchange

Currently the only supported contextual parameter is the exchange which can be provided by specifying a method parameter of a type assignable from `WebExchange`.

```
@WebRoute(method = Method.POST, consumes = MediaType.APPLICATION_JSON)
Mono<Void> create(@Body Mono<T> book, WebExchange exchange);
```

The exchange gives a full access to the underlying request and response. Although it allows to manipulate the request and response bodies, this might conflict with the generated Web route and as a result the exchange should only be used to specify a specific response status, response cookies or headers...

Request body

The request body can be bound to a route method parameter using the `@Body` annotation. Request body is automatically converted based on the media type declared in the `content-type` header field of the request as described in the [Web server exchange documentation](#). The body parameter method can then be of any type as long as there is a media type converter for the media type specified in the request that can convert it.

In the following example, the request body is bound to parameter `book` of type `Book`, it is then converted from `application/json` into a `Book` instance:

```
@WebRoute(method = Method.POST, consumes = MediaType.APPLICATION_JSON)
void create(@Body Book book);
```

Unlike parameters, the request body can be provided in a non-blocking/reactive way, the previous example can then be rewritten using a `Mono<T>`, a `Flux<T>` or more broadly a `Publisher<T>` as body parameter type as follows:

```
@WebRoute(method = Method.POST, consumes = MediaType.APPLICATION_JSON)
Mono<Void> create(@Body Mono<Book> book);
```

A stream of objects can be processed when the media type converter supports it. For instance, the `application/x-ndjson` converter can emit converted objects each time a new line is encountered, this allows to process content without having to wait for the entire message resulting in better response time and reduced memory consumption.

```
@WebRoute(method = Method.POST, consumes = MediaType.APPLICATION_X_NDJSON)
Mono<Void> create(@Body Flux<Book> book);
```

The `application/json` also supports such streaming capability by emitting converted objects while parsing a JSON array.

The `@Body` annotation can't be used together with the `@FormParam` annotation on route method parameters because the request body can only be consumed once.

Multipart form data

Multipart form data request body can be bound by defining a body parameter of type `Mono<WebPart>` if one part is expected, `Flux<WebPart>` if multiple parts are expected or more broadly of type `Publisher<WebPart>`.

We can then rewrite the example described in [Web server exchange documentation](#) as follows:

```
@WebRoute( path = "/bulk", method = Method.POST, consumes = MediaType.MULTIPART_FORM_DATA)
Flux<Result> createBulk(@Body Flux<WebPart> parts) {
    return parts
        .flatMap(part -> part.decoder(Book.class).one())
        .map(book -> storeBook(book));
}
```

It is not possible to bind particular parts to a route method parameter. This design choice has been motivated by performance and resource consumption considerations. Indeed, this would require to consume and store the entire request body in memory before invoking the method. As a result, multipart data must still be handled *manually* using the *web* module API.

Response body

The response body is specified by the return type of the route method.

```
@WebRoute(path =("/{id}", method = Method.GET, produces = MediaType.APPLICATION_JSON)
Book get(@PathParam String id);
```

As for the request body, the response body can be reactive if specified as a `Mono<T>`, a `Flux<T>` or more broadly as a `Publisher<T>`:

```
@WebRoute(path =("/{id}", method = Method.GET, produces = MediaType.APPLICATION_JSON)
Mono<Book> get(@PathParam String id);
```

Depending on the media type converter, partial responses can be sent to the client as soon as they are complete. For instance a stream of responses can be sent to a client as follows:

```
@WebRoute(path = "/", method = Method.GET, produces = MediaType.APPLICATION_X_NDJSON)
Stream<Book> list();
```

In the preceding example, as soon as a book is retrieved from a data store it can be sent to the client which can then process responses as soon as possible reducing the latency and resource consumption on both client and server. The response content type is `application/x-ndjson`, so each book is encoded in JSON before a newline delimiter to let the client detects partial responses as defined by [the ndjon format](#).

Server-sent events

Server-sent events can be streamed in the response body when declared together with a server-sent event factory route method parameter. A server-sent event factory can be bound to a route method parameter using the `@SseEventFactory` annotation.

In the following example, we declare a basic server-sent events Web route producing events with a `String` message:

```
@WebRoute(path = "/event", method = Method.GET)
Publisher<WebResponseBody.SseEncoder.Event<String>> getBookEvents(@SseEventFactory
WebResponseBody.SseEncoder.EventFactory<String> events);
```

Server-sent event return type can be any of `Mono<WebResponseBody.SseEncoder.Event<T>>` if only one event is expected, `Flux<WebResponseBody.SseEncoder.Event<T>>` if multiple events are expected and more broadly `Publisher<WebResponseBody.SseEncoder.Event<T>>`.

By default, the media type of a server-sent event message is `text/plain` but it can be encoded using a specific media type converter as well by specifying a media type in the `@SseEventFactory` annotation.

We can rewrite previous [SSE example](#) with a message of a custom type as follows:

```
@WebRoute(path = "/event", method = Method.GET)
public Publisher<WebResponseBody.SseEncoder.Event<BookEvent>>
getBookEvents(@SseEventFactory(MediaType.APPLICATION_JSON)
WebResponseBody.SseEncoder.EventFactory<BookEvent> events) {
    return Flux.interval(Duration.ofSeconds(1))
        .map(seq -> events.create(
            event -> event
                .id(Long.toString(seq))
                .event("bookEvent")
                .value(new BookEvent("some book event"))
        ))
    );
}
```

Composite Web module

The Web Winter compiler plugin generates a single Web router configurer bean aggregating all route definitions specified in Web controllers beans in the module. When the module composes the *web* module, this bean is then plugged in the *web* module to configure the Web server router.

Now if the module doesn't compose the *web* module, the Web router configurer bean is simply exposed by the module waiting for the module to be composed within other modules until a top module eventually composes the *web* module.

This raises two issues:

- First if multiple modules exposing web router configurers are composed together with the *web* module, we'll end up with dependency injection conflicts since multiple web router configurer beans can be wired to the *web* module. Selecting one of them with a `@Wire` annotation doesn't really solve the problem since we expect all routes to be configured in the Web server router.
- Then if such module is composed in another module defining other Web controllers, we still need to expose one Web router configurer providing all route definitions to a top module composing the *web* module.

In order to solve these issues, the Web Winter compiler plugin aggregates all Web router configurer beans annotated with `@WebRoutes` into the generated Web router configurer of the module so that it can be used to configure all defined routes. This includes Web router configurer exposed in component modules as well as user defined Web router configurer beans within the module.

A generated Web router configurer is always annotated with a `@WebRoutes` annotation specifying the Web routes it configures. For instance, the Web router configurer generated for the module defining the book Web controller looks like:


```

@WebRoutes({
    @WebRoute(path = { "/book/{id}" }, method = { Method.GET }, produces = { "application/json" }),
    @WebRoute(path = { "/book" }, method = { Method.POST }, consumes = { "application/json" }),
    @WebRoute(path = { "/book/{id}" }, method = { Method.PUT }, consumes = { "application/json" }),
    @WebRoute(path = { "/book" }, method = { Method.GET }, produces = { "application/json" }),
    @WebRoute(path = { "/book/{id}" }, method = { Method.DELETE })
})
@Bean
public final class WebRouterConfigurer implements
io.winterframework.mod.web.WebRouterConfigurer<WebExchange> {
    ...
}

```

This information is used by the compiler plugin to statically check that there is no conflicting routes when generating the Web router configurer.

Now let's imagine we have created a modular Web application with a *book* module defining the book Web controller, an *admin* module defining some admin Web controllers and a top *app* module composing these modules together with the *web* module.

The module descriptors for each of these modules should look like:

```

@io.winterframework.core.annotation.Module( excludes = { "io.winterframework.mod.web" } )
module io.winterframework.example.web_modular.admin {
    requires io.winterframework.core;
    requires io.winterframework.mod.web;

    exports io.winterframework.example.web_modular.admin to
io.winterframework.example.web_modular.app;
}

@io.winterframework.core.annotation.Module( excludes = { "io.winterframework.mod.web" } )
module io.winterframework.example.web_modular.book {
    requires io.winterframework.core;
    requires io.winterframework.mod.web;

    exports io.winterframework.example.web_modular.book to
io.winterframework.example.web_modular.app;
    exports io.winterframework.example.web_modular.book.dto to com.fasterxml.jackson.databind;
}

@io.winterframework.core.annotation.Module
module io.winterframework.example.web_modular.app {
    requires io.winterframework.mod.boot;
    requires io.winterframework.mod.web;

    requires io.winterframework.example.web_modular.admin;
    requires io.winterframework.example.web_modular.book;
}

```

The first thing to notice is that the *web* module is excluded from *admin* and *book* modules since we don't want to start a Web server in these modules, we only need the Web API to define Web controllers and generate Web router configurer beans. As a consequence, the *boot* module which provides converters and net service required to create and start the *web* module is also not required but the *io.winterframework.core* module is still required. Finally we must export packages containing the generated module classes to the *app* module so it can compose them.

The *admin* and *book* modules should compile just fine resulting in two Web router configurer beans being generated and exposed in each module. But the compilation of *app* module should raise some dependency injection errors since multiple Web router configurer beans exist whereas only one can be wired to the *web* module. There are actually three Web configurer beans, how so? There are those exposed by the *admin* and *book* modules and one Web router configurer bean generated in the *app* module and aggregating the previous two. In order to solve the conflict, we should then define the following explicit wire in the *app* module:

```
@io.winterframework.core.annotation.Module
@io.winterframework.core.annotation.Wire(beans="io.winterframework.example.web_modular.app:webRouterConfigurer", into="io.winterframework.mod.web:webRouterConfigurer")
module io.winterframework.example.web_modular.app {
    ...
}
```

One could rightfully argue that this explicit wiring is useless and cumbersome, but it is consistent with the IoC/DI core framework principles. Keeping things simple and explicit limits possible side effects induced by the fact that what's happening with *automatic* conflict resolution is often specific and might not be obvious. This is all the more true when such behavior is manually overridden.

The same principles applies if multiple modules like *admin* or *book* are cascaded into one another: the Web router configurer beans at a given level are aggregated in the Web router configurer bean in the next level.

Automatic OpenAPI specifications

Besides facilitating the development of REST and Web resources in general, Web controllers also simplify documentation. The Web Winter compiler plugin can be setup to generate [Open API](#) specifications from the Web controller classes defined in a module and their JavaDoc comments.

Writing JavaDoc comments is something natural when developing in the Java language, with this approach, a REST API can be documented just as you document a Java class or method, documentation is written once and can be used in both Java and other languages and technologies using the generated Open API specification.

In order to activate this feature the `winter.web.generateOpenApiDefinition` annotation processor option must be enabled when compiling a Web module. This can be done on the command line: `java -Awinter.web.generateOpenApiDefinition=true ...` or in the Maven compiler plugin configuration in the build descriptor:

```

<project>
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <configuration>
            <compilerArgs combine.children="append">
              <arg>-Awinter.web.generateOpenApiDefinition=true</arg>
            </compilerArgs>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>

```

The compiler then generates an Open API specification in `META-INF/winter/web/openapi.yml` for any module defining one or more Web controllers.

The previous [book resource](#) could then be documented as follows:

```

/**
 * The book resource.
 */
@Bean
@WebController(path = "/book")
public class BookResource {

    /**
     * Creates a book resource.
     *
     * @param book a book
     * @param exchange the web exchange
     *
     * @return the book resource has been successfully created
     * @throws BadRequestException A book with the same ISBN reference already exist
     */
    @WebRoute(method = Method.POST, consumes = MediaType.APPLICATION_JSON)
    public Mono<Void> create(@Body Mono<Book> book, WebExchange exchange) throws BadRequestException
    { ... }

    /**
     * Updates a book resource.
     *
     * @param isbn the reference of the book resource to update
     * @param book the updated book resource
     *
     * @return the book resource has been successfully updated
     * @throws NotFoundException if the specified reference does not exist
     */
    @WebRoute(path =("/{isbn}", method = Method.PUT, consumes = MediaType.APPLICATION_JSON)
    public Mono<Void> update(@PathParam String isbn, @Body Mono<Book> book) throws NotFoundException
    { ... }

    /**
     * Returns the list of book resources.
     *
     * @return a list of book resources
     */
    @WebRoute(method = Method.GET, produces = MediaType.APPLICATION_JSON)
    public Flux<Book> list();

    /**
     * Returns the book resource identified by the specified ISBN.
     *
     * @param isbn an ISBN
     *
     * @return the requested book resource
     * @throws NotFoundException if the specified reference does not exist
     */
    @WebRoute(path =("/{isbn}", method = Method.GET, produces = MediaType.APPLICATION_JSON)
    public Mono<Book> get(@PathParam String isbn) throws NotFoundException { ... }

    /**
     * Deletes the book resource identified by the specified ISBN.
     *
     * @param isbn an ISBN
     *
     * @return the book resource has been successfully deleted
     * @throws NotFoundException if the specified reference does not exist
     */
    @WebRoute(path =("/{isbn}", method = Method.DELETE)
    public Mono<Void> delete(@PathParam String isbn) { ... }
}

```

Note that just like the `javadoc` tool, the Web compiler plugin takes inheritance into account when resolving JavaDoc comments and as a result, it is possible to define JavaDoc comments in an interface and enrich or override them in the implementation classes.

By default, the normal HTTP status code responded by a route is assumed to be `200` but it is possible to specify a custom status code using the `@winter.web.status` tag. For instance the book creation route which actually responds with a `201` status should be documented as follows:

```
public class BookResource {

    /**
     * Creates a book resource.
     *
     * @param book a book
     * @param exchange the web exchange
     *
     * @return {@code @winter.web.status 201} the book resource has been successfully created
     * @throws BadRequestException A book with the same ISBN reference already exist
     */
    @WebRoute(method = Method.POST, consumes = MediaType.APPLICATION_JSON)
    public Mono<Void> create(@Body Mono<Book> book, WebExchange exchange) throws BadRequestException
    { ... }

    ...
}
```

Multiple `@return` statements can be specified if multiple response statuses are expected, however this might raise issues during the generation of the JavaDoc, you can bypass this by disabling the linter with `-Xdoclint:none` option.

This tag can also be used to specify error status code in `@throws` statements, but this is usually not necessary since the Web compiler plugin automatically detects status code for regular `HttpException` such as `BadRequestException` (400) or `NotFoundException` (404).

The Web compiler plugin generates, per module, one Open API specification and one Web router configurator bean aggregating all routes from all Web controllers. As a result the general API documentation corresponds to the general documentation of the module and defined in the module descriptor JavaDoc comment.

For instance, we can describe the API exposed by the *book* module in the module descriptor including the API version which should normally match the module version:

```

/**
 * This is a sample Book API which demonstrates Winter Web module capabilities.
 *
 * @author <a href="mailto:jeremy.kuhn@winterframework.io">Jeremy Kuhn</a>
 *
 * @version 1.2.3
 */
@io.winterframework.core.annotation.Module( excludes = { "io.winterframework.mod.web" } )
module io.winterframework.example.web_modular.book {
    requires io.winterframework.core;
    requires io.winterframework.mod.web;

    exports io.winterframework.example.web_modular.book to
io.winterframework.example.web_modular.app;
    exports io.winterframework.example.web_modular.book.dto to com.fasterxml.jackson.databind;
}

```

These specifications can also be exposed in the Web server as described in the [Web server documentation](#).

If we build and run the [modular book application](#) and access <http://localhost:8080/open-api> in a Web browser we should see a Swagger UI loaded with the Open API specifications of the *admin* and *book* modules:

The screenshot displays the Swagger UI for the API `io.winterframework.example.web_modular.book` at version `1.2.3` using the `OAS3` specification. The UI includes a header with the Swagger logo and a dropdown menu to select a definition. The main content area shows the API title and a description: "This is a sample Book API which demonstrates Winter Web module capabilities." Below this, there is a link to "Contact Jeremy Kuhn". The API endpoints are listed in a table-like format with colored headers: **POST** `/book` (Creates a book resource), **GET** `/book` (Returns the list of book resources), **GET** `/book/{isbn}` (Returns the book resource identified by the specified ISBN), **PUT** `/book/{isbn}` (Updates a book resource), and **DELETE** `/book/{isbn}` (Deletes the book resource identified by the specified ISBN). At the bottom, there is a section for "Schemas" with a right arrow.

It is also possible to target a single specification by specifying the module name in the URI, for instance http://localhost:8080/open-api/io.winterframework.example.web_modular.book:

io.winterframework.example.web_modular.book 1.2.3 OAS3

/open-api/io.winterframework.example.web_modular.book

This is a sample Book API which demonstrates Winter Web module capabilities.

[Contact Jeremy Kuhn](#)

bookResource The book resource.

POST **/book** Creates a book resource.

GET **/book** Returns the list of book resources.

GET **/book/{isbn}** Returns the book resource identified by the specified ISBN.

PUT **/book/{isbn}** Updates a book resource.

DELETE **/book/{isbn}** Deletes the book resource identified by the specified ISBN.

Schemas

Finally, Open API specifications formatted in [YAML](#) can be retrieved as follows:

```
$ curl http://localhost:8080/open-api/io.winterframework.example.web_modular.admin
```

```
openapi: 3.0.3
```

```
info:
```

```
  title: 'io.winterframework.example.web_modular.admin'
```

```
  version: ''
```

```
...
```

6

Winter Maven Plugin

The Winter Maven Plugin is used to run, package and distribute modular applications and Winter applications in particular. It relies on a set of Java tools to build native runtime or application images as well as Docker or OCI images for modular Java projects.

Usage

The Winter Maven plugin can be used to run a modular application project or build an image for a modular project. There are three types of images that can be build using the plugin:

- **runtime image** is a custom Java runtime containing a set of modules and their dependencies.
- **application image** is a native self-contained Java application including all the necessary dependencies to run the application without the need of a Java runtime.
- **container image** is a Docker or CLI container image that can be packaged as a TAR archive or directly deployed on a Docker daemon or container registry.

Run a module application project

The `winter:run` goal is used to execute the modular application defined in the project from the command line.

```
$ mvn winter:run
```

The application is first *modularized* which means that any non-modular dependency is modularized by generating an appropriate module descriptor using the `jdeps` tool in order for the application to be run with a module path and not a class path (and certainly not both).

The application is executed in a forked process, application arguments can be passed on the command line:


```
$ mvn winter:run -Dwinter.run.arguments='--some.configuration="hello\''
```

Actual arguments are determined by splitting the parameter value around spaces. There are several options to declare an argument which contains spaces:

- it can be escaped: `Hello\ World`
- it can be quoted: `"Hello World"` or `'Hello World'`

Since quotes or double quotes are used as delimiters, they might need to be escaped as well to declare an argument that contains some: `I\'m\ happy`, `"I'm happy"`, `'I\'m happy'`.

In order to debug the application, we need to specify the appropriate options to the JVM:

```
$ mvn winter:run -Dwinter.exec.vmOptions="-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=8000"
```

By default the plugin will detect the main class of the application, but it is also possible to specify it explicitly in case multiple main classes exist in the project module.

```
$ mvn winter:run -Dwinter.exec.mainClass=io.winterframework.example.Main
```

A pidfile is created when the application is started under `${project.build.directory}/maven-winter` directory, it indicates the pid of the process running the application. If the build exits while the application is still running or if the pidfile was not properly removed after the application has exited, it might be necessary to manually kill the process and/or remove the pidfile.

Start and stop the application for integration testing

The `winter:start` and `winter:stop` goals are used together to start and stop the application while not blocking the Maven build process which can then execute other goals targeting the running application such as integration tests.

They are bound to the `pre-integration-test` and `pre-integration-test` phases respectively:

```

<project>
  <build>
    <plugins>
      <plugin>
        <groupId>io.winterframework.tool</groupId>
        <artifactId>winter-maven-plugin</artifactId>
        <executions>
          <execution>
            <id>start</id>
            <phase>pre-integration-test</phase>
            <goals>
              <goal>start</goal>
            </goals>
          </execution>
          <execution>
            <id>stop</id>
            <phase>post-integration-test</phase>
            <goals>
              <goal>stop</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>

```

Build a runtime image

A runtime image is a custom Java runtime distribution containing specific modules and their dependencies. Such image is used as a base for generating application image but it can also be distributed as a lightweight Java runtime.

The `winter:build-runtime` goal uses `jlink` tool to assemble the project module and its dependencies.

```

<project>
  <build>
    <plugins>
      <plugin>
        <groupId>io.winterframework.tool</groupId>
        <artifactId>winter-maven-plugin</artifactId>
        <executions>
          <execution>
            <id>build-project-runtime</id>
            <phase>package</phase>
            <goals>
              <goal>build-runtime</goal>
            </goals>
            <configuration>
              <vm>server</vm>
              <addModules>jdk.jdwp.agent,jdk.crypto.ec</addModules>
              <vmOptions>-Xms2G -Xmx2G -XX:+UseNUMA -XX:+UseParallelGC</vmOptions>
              <formats>
                <format>zip</format>
                <format>tar.gz</format>
                <format>tar.bz2</format>
              </formats>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>

```

By default, the project module and its dependencies are included in the resulting image, this include JDK's modules such as `java.base`, in the previous example we've also explicitly added the `jdk.jdwp.agent` to support remote debugging and `jdk.crypto.ec` to support TLS communications.

The resulting image is packaged to the formats defined in the configuration and attached, by default, to the Maven project.

Build an application image

An application image is built using the `winter:build-app` goal which basically generates a runtime image and uses `jpackage` tool to generate a native platform-specific application package.

```

<project>
  <build>
    <plugins>
      <plugin>
        <groupId>io.winterframework.tool</groupId>
        <artifactId>winter-maven-plugin</artifactId>
        <executions>
          <execution>
            <id>build-application</id>
            <phase>package</phase>
            <goals>
              <goal>build-app</goal>
            </goals>
            <configuration>
              <vm>server</vm>
              <addModules>jdk.jdwp.agent,jdk.crypto.ec</addModules>
              <vmOptions>-Xms2G -Xmx2G -XX:+UseNUMA -XX:+UseParallelGC</vmOptions>
              <formats>
                <format>zip</format>
                <format>deb</format>
              </formats>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>

```

The `winter:build-app` goal is very similar to the `winter:build-runtime` goal except that the resulting image provides an application launcher and it can be packaged in a platform-specific format. For instance, we can generate a `.deb` on a Linux platform or a `.exe` or `.msi` on a Windows platform or a `.dmg` on a MacOS platform. The resulting package can be installed on these platforms in a standard way.

This goal uses `jpackage` tool which is an incubating feature in JDK<16, if you intend to build an application image with an old JDK, you'll need to explicitly add the `jdk.incubator.jpackage` module in `MAVEN_OPTS`:

```
$ export MAVEN_OPTS="--add-modules jdk.incubator.jpackage"
```

Build a container image tarball

A container image can be built in a TAR archive using the `winter:build-image-tar` goal which basically build an application package and package it in a container image.

```

<project>
  <build>
    <plugins>
      <plugin>
        <groupId>io.winterframework.tool</groupId>
        <artifactId>winter-maven-plugin</artifactId>
        <executions>
          <execution>
            <id>build-image-tar</id>
            <phase>package</phase>
            <goals>
              <goal>build-image-tar</goal>
            </goals>
            <configuration>
              <vm>server</vm>
              <addModules>jdk.jdwp.agent,jdk.crypto.ec</addModules>
              <vmOptions>-Xms2G -Xmx2G -XX:+UseNUMA -XX:+UseParallelGC</vmOptions>
              <repository>example</repository>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>

```

The resulting image reference is defined by `${registry}/${repository}/${name}:${project.version}`, the registry and the repository are optional and the name default to the project artifact id.

The resulting image can then be loaded in a docker daemon:

```
$ docker load --input target/example-1.0.0-SNAPSHOT-container_linux_amd64.tar
```

As for `build-app` goal, this goal uses `jpackage` tool so if you intend to use a JDK<16 you'll need to explicitly add the `jdk.incubator.jpackage` module in `MAVEN_OPTS`:

```
$ export MAVEN_OPTS="--add-modules jdk.incubator.jpackage"
```

Build and deploy a container image to a Docker daemon

The `winter:build-image-docker` goal is used to build a container image and deploy it to a Docker daemon using the Docker CLI.

```

<project>
  <build>
    <plugins>
      <plugin>
        <groupId>io.winterframework.tool</groupId>
        <artifactId>winter-maven-plugin</artifactId>
        <executions>
          <execution>
            <id>build-image-docker</id>
            <phase>package</phase>
            <goals>
              <goal>build-image-docker</goal>
            </goals>
            <configuration>
              <vm>server</vm>
              <addModules>jdk.jdwp.agent,jdk.crypto.ec</addModules>
              <vmOptions>-Xms2G -Xmx2G -XX:+UseNUMA -XX:+UseParallelGC</vmOptions>
              <repository>example</repository>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>

```

By default the `docker` command is used but it is possible to specify the path to the Docker CLI in the `winter.container.docker.executable` parameter.

As for `build-app` goal, this goal uses `jpackage` tool so if you intend to use a JDK<16 you'll need to explicitly add the `jdk.incubator.jpackage` module in `MAVEN_OPTS`:

```
$ export MAVEN_OPTS="--add-modules jdk.incubator.jpackage"
```

Build and deploy a container image to a remote repository

The `winter:build-image` goal builds a container image and deploy it to a remote repository.

```

<project>
  <build>
    <plugins>
      <plugin>
        <groupId>io.winterframework.tool</groupId>
        <artifactId>winter-maven-plugin</artifactId>
        <executions>
          <execution>
            <id>build-image-docker</id>
            <phase>package</phase>
            <goals>
              <goal>build-image-docker</goal>
            </goals>
            <configuration>
              <vm>server</vm>
              <addModules>jdk.jdwp.agent,jdk.crypto.ec</addModules>
              <vmOptions>-Xms2G -Xmx2G -XX:+UseNUMA -XX:+UseParallelGC</vmOptions>
              <registryUsername>user</registryUsername>
              <registryPassword>password</registryPassword>
              <registry>gcr.io</registry>
              <repository>example</repository>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>

```

By default the registry points to the Docker hub registry-1.docker.io but another registry can be specified, gcr.io in our example.

As for [build-app](#) goal, this goal uses [jpackage](#) tool so if you intend to use a JDK<16 you'll need to explicitly add the [jdk.incubator.jpackage](#) module in [MAVEN_OPTS](#):

```
$ export MAVEN_OPTS="--add-modules jdk.incubator.jpackage"
```

Goals

Overview

- [winter:build-app](#) Builds the project application package.
- [winter:build-image](#) Builds a container image and publishes it to a registry.
- [winter:build-image-docker](#) Builds a Docker container image to a local Docker daemon.
- [winter:build-image-tar](#) Builds a container image to a TAR archive that can be later loaded into Docker:
- [winter:build-runtime](#) Builds the project runtime image.
- [winter:help](#) Display help information on winter-maven-plugin.
- [winter:run](#) Runs the project application.
- [winter:start](#) Starts the project application without blocking the Maven build.
- [winter:stop](#) Stops the project application that has been previously started using the start goal.

winter:build-app

Full name:

io.winterframework.tool:winter-maven-plugin:1.0.0:build-app

Description:

Builds the project application package.

A project application package is a native self-contained Java application including all the necessary dependencies. It can be used to distribute a complete application.

Attributes:

- Requires a Maven project to be executed.
- Requires dependency resolution of artifacts in scope: compile+runtime.
- Requires dependency collection of artifacts in scope: compile+runtime.
- Since version: 1.0.
- Binds by default to the lifecycle phase: package.

Required parameters

Name	Type	Default
attach	boolean	Attach the resulting image archives to the project to install them in the local Maven repository and deploy them to remote repositories. <ul style="list-style-type: none">• <i>User property</i> : winter.image.attach• <i>Default</i> : true
formats	Set	A list of archive formats to generate (eg. zip, tar.gz...) <ul style="list-style-type: none">• <i>Default</i> : zip

Optional parameters

Name	Type	Description
addModules	String	The modules to add to the resulting image <ul style="list-style-type: none"> <i>User property</i> : winter.image.addModules
addOptions	String	The options to prepend before any other options when invoking the JVM in the resulting image. <ul style="list-style-type: none"> <i>User property</i> : winter.image.addOptions
automaticLaunchers	boolean	Enables the automatic generation of launchers based on the main classes extracted from the application module. If enabled, a launcher is generated for all main classes other than the main launcher. <ul style="list-style-type: none"> <i>User property</i> : winter.app.automaticLaunchers <i>Default</i> : false
bindServices	boolean	Link in service provider modules and their dependencies. <ul style="list-style-type: none"> <i>User property</i> : winter.image.bindServices <i>Default</i> : false
compress	String	The compress level of the resulting image 0=No compression, 1=constant string sharing, 2=ZIP. <ul style="list-style-type: none"> <i>User property</i> : winter.image.compress
configurationDirectory	File	A directory containing user-editable configuration files that will be copied to the resulting image. <ul style="list-style-type: none"> <i>User property</i> : winter.image.configurationDirectory <i>Default</i> : \${project.basedir}/src/main/conf/
copyright	String	The application copyright. <ul style="list-style-type: none"> <i>User property</i> : winter.app.copyright
description	String	The description of the application. <ul style="list-style-type: none"> <i>User property</i> : winter.app.description <i>Default</i> : \${project.description}
excludeArtifactIds	String	Comma separated list of Artifact names to exclude. <ul style="list-style-type: none"> <i>User property</i> : winter.image.excludeArtifactIds
excludeClassifiers	String	Comma Separated list of Classifiers to

		<p>exclude. Empty String indicates don't exclude anything (default).</p> <ul style="list-style-type: none"> • <i>User property</i> : excludeClassifiers
excludeGroupIds	String	<p>Comma separated list of GroupId Names to exclude.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.excludeGroupIds
excludeScope	String	<p>Scope to exclude. An Empty string indicates no scopes (default).</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.excludeScope
ignoreSigningInformation	boolean	<p>Suppress a fatal error when signed modular JARs are linked in the image.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.ignoreSigningInformation • <i>Default</i> : false
includeArtifactIds	String	<p>Comma separated list of Artifact names to include. Empty String indicates include everything (default).</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.includeArtifactIds
includeClassifiers	String	<p>Comma Separated list of Classifiers to include. Empty String indicates include everything (default).</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.includeClassifiers
includeGroupIds	String	<p>Comma separated list of GroupIds to include. Empty String indicates include everything (default).</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.includeGroupIds
includeScope	String	<p>Scope to include. An Empty string indicates all scopes (default). The scopes being interpreted are the scopes as Maven sees them, not as specified in the pom. In summary:</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.includeScope
installDirectory	String	<p>Absolute path of the installation directory of the application on OS X or Linux. Relative sub-path of the installation location of the application such as 'Program Files' or 'AppData' on Windows.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.app.installDirectory

jmodsOverrideDirectory	File	<p>A directory containing module descriptors to use to modularize unnamed dependenc modules and which override the ones that are otherwise generated.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.jmodsOverrideDirectory • <i>Default</i> : \${project.basedir}/src/jmods/
launchers	List	A list of extra launchers to include in the resulting application.
legalDirectory	File	<p>A directory containing legal notices that will be copied to the resulting image.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.legalDirectory • <i>Default</i> : \${project.basedir}/src/main/legal/
licenseFile	File	<p>The path to the application license file.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.app.licenseFile • <i>Default</i> : \${project.basedir}/LICENSE
linuxConfiguration	LinuxConfiguration	Linux specific configuration.
macOSConfiguration	MacOSConfiguration	MacOS specific configuration.
manDirectory	File	<p>A directory containing man pages that will be copied to the resulting image.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.manDirectory • <i>Default</i> : \${project.basedir}/src/main/man/
overWritelfNewer	boolean	<p>Overwrite dependencies that don't exist or are older than the source.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.overWritelfNewer • <i>Default</i> : true
projectMainClass	String	<p>The main class in the project module to use when building the project JMOD package.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.runtime.projectMainClass
resolveProjectMainClass	boolean	<p>Resolve the project main class when not specified explicitly.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.runtime.projectMainClass • <i>Default</i> : false

resourceDirectory	File	The path to resources that override resulting package resources. <ul style="list-style-type: none">• <i>User property</i> : winter.app.resourceDirectory
skip	boolean	Skips the generation of the application. <ul style="list-style-type: none">• <i>User property</i> : winter.app.skip
stripDebug	boolean	Strip debug information from the resulting image. <ul style="list-style-type: none">• <i>User property</i> : winter.image.stripDebug• <i>Default</i> : true
stripNativeCommands	boolean	Strip native command (eg. java...) from the resulting image. <ul style="list-style-type: none">• <i>User property</i> : winter.image.stripNativeCommands• <i>Default</i> : true
vendor	String	The application vendor. <ul style="list-style-type: none">• <i>User property</i> : winter.app.vendor• <i>Default</i> : \${project.organization.name}
verbose	boolean	Enables verbose logging. <ul style="list-style-type: none">• <i>User property</i> : winter.verbose• <i>Default</i> : false
vm	String	Select the HotSpot VM in the output image defined as: 'client' / 'server' / 'minimal' / 'all' <ul style="list-style-type: none">• <i>User property</i> : winter.image.vm
windowsConfiguration	WindowsConfiguration	Windows specific configuration.



Parameter details

<addModules>

The modules to add to the resulting image.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.addModules

<addOptions>

The options to prepend before any other options when invoking the JVM in the resulting image.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.addOptions

<attach>

Attach the resulting image archives to the project to install them in the local Maven repository and deploy them to remote repositories.

- **Type:** boolean
- **Required:** yes
- **User property:** winter.image.attach
- **Default:** true

<automaticLaunchers>

Enables the automatic generation of launchers based on the main classes extracted from the application module. If enabled, a launcher is generated for all main classes other than the main launcher.

- **Type:** boolean
- **Required:** no
- **User property:** winter.app.automaticLaunchers
- **Default:** false

<bindServices>

Link in service provider modules and their dependencies.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.bindServices
- **Default:** false

<compress>

The compress level of the resulting image: 0=No compression, 1=constant string sharing, 2=ZIP.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.compress

<configurationDirectory>

A directory containing user-editable configuration files that will be copied to the resulting image.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.configurationDirectory
- **Default:** \${project.basedir}/src/main/conf/

<copyright>

The application copyright.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.app.copyright

<description>

The description of the application.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.app.description
- **Default:** \${project.description}

<excludeArtifactIds>

Comma separated list of Artifact names to exclude.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.excludeArtifactIds

<excludeClassifiers>

Comma Separated list of Classifiers to exclude. Empty String indicates don't exclude anything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** excludeClassifiers

<excludeGroupIds>

Comma separated list of GroupId Names to exclude.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.excludeGroupIds

<excludeScope>

Scope to exclude. An Empty string indicates no scopes (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.excludeScope

<formats>

A list of archive formats to generate (eg. zip, tar.gz...)

- **Type:** java.util.Set
- **Required:** yes
- **Default:** zip

<ignoreSigningInformation>

Suppress a fatal error when signed modular JARs are linked in the image.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.ignoreSigningInformation
- **Default:** false

<includeArtifactIds>

Comma separated list of Artifact names to include. Empty String indicates include everything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeArtifactIds

<includeClassifiers>

Comma Separated list of Classifiers to include. Empty String indicates include everything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeClassifiers

<includeGroupIds>

Comma separated list of GroupIds to include. Empty String indicates include everything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeGroupIds

<includeScope>

Scope to include. An Empty string indicates all scopes (default). The scopes being interpreted are the scopes as Maven sees them, not as specified in the pom. In summary:

- runtime scope gives runtime and compile dependencies,
- compile scope gives compile, provided, and system dependencies,
- test (default) scope gives all dependencies,
- provided scope just gives provided dependencies,
- system scope just gives system dependencies.
- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeScope

<installDirectory>

Absolute path of the installation directory of the application on OS X or Linux. Relative sub-path of the installation location of the application such as 'Program Files' or 'AppData' on Windows.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.app.installDirectory

<jmodsOverrideDirectory>

A directory containing module descriptors to use to modularize unnamed dependency modules and which override the ones that are otherwise generated.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.jmodsOverrideDirectory
- **Default:** \${project.basedir}/src/jmods/

<launchers>

A list of extra launchers to include in the resulting application.

- **Type:** java.util.List
- **Required:** no

<legalDirectory>

A directory containing legal notices that will be copied to the resulting image.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.legalDirectory
- **Default:** \${project.basedir}/src/main/legal/

<licenseFile>

The path to the application license file.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.app.licenseFile
- **Default:** \${project.basedir}/LICENSE

<linuxConfiguration>

Linux specific configuration.

- **Type:**
io.winterframework.tool.maven.internal.task.CreateProjectApplicationTask\$LinuxConfigurati
- **Required:** no

<macOSConfiguration>

MacOS specific configuration.

- **Type:**
io.winterframework.tool.maven.internal.task.CreateProjectApplicationTask\$MacOSConfigura
- **Required:** no

<manDirectory>

A directory containing man pages that will be copied to the resulting image.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.manDirectory
- **Default:** \${project.basedir}/src/main/man/

<overWriteIfNewer>

Overwrite dependencies that don't exist or are older than the source.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.overWriteIfNewer
- **Default:** true

<projectMainClass>

The main class in the project module to use when building the project JMOD package.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.runtime.projectMainClass

<resolveProjectMainClass>

Resolve the project main class when not specified explicitly.

- **Type:** boolean
- **Required:** no
- **User property:** winter.runtime.projectMainClass
- **Default:** false

<resourceDirectory>

The path to resources that override resulting package resources.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.app.resourceDirectory

<skip>

Skips the generation of the application.

- **Type:** boolean
- **Required:** no
- **User property:** winter.app.skip

<stripDebug>

Strip debug information from the resulting image.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.stripDebug
- **Default:** true

<stripNativeCommands>

Strip native command (eg. java...) from the resulting image.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.stripNativeCommands
- **Default:** true

<vendor>

The application vendor.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.app.vendor
- **Default:** \${project.organization.name}

<verbose>

Enables verbose logging.

- **Type:** boolean
- **Required:** no
- **User property:** winter.verbose
- **Default:** false

<vm>

Select the HotSpot VM in the output image defined as: 'client' / 'server' / 'minimal' / 'all'

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.vm

<windowsConfiguration>

Windows specific configuration.

- **Type:**
io.winterframework.tool.maven.internal.task.CreateProjectApplicationTask\$WindowsConfigu
- **Required:** no

winter:build-image

Full name:

io.winterframework.tool:winter-maven-plugin:1.0.0:build-image

Description:

Builds a container image and publishes it to a registry.

Attributes:

- Requires a Maven project to be executed.
- Requires dependency resolution of artifacts in scope: compile+runtime.
- Requires dependency collection of artifacts in scope: compile+runtime.
- Since version: 1.0.
- Binds by default to the lifecycle phase: install.

Required parameters

Name	Type	Default
attach	boolean	Attach the resulting image archives to the project to install them in the local Maven repository and deploy them to remote repositories. <ul style="list-style-type: none">• <i>User property</i> : winter.image.attach• <i>Default</i> : true
executable	String	The executable in the application image to use as image entry point. The specified name should correspond to a declared application image launchers or the project artifact id if no launcher was specified. <ul style="list-style-type: none">• <i>User property</i> : winter.app.executable• <i>Default</i> : \${project.artifactId}
formats	Set	A list of archive formats to generate (eg. zip, tar.gz...) <ul style="list-style-type: none">• <i>Default</i> : zip
from	String	The base container image. <ul style="list-style-type: none">• <i>User property</i> : winter.container.from• <i>Default</i> : debian:buster-slim

Optional parameters

Name	Type	Description
addModules	String	The modules to add to the resulting image. <ul style="list-style-type: none"> <i>User property</i> : winter.image.addMod
addOptions	String	The options to prepend before any other o invoking the JVM in the resulting image. <ul style="list-style-type: none"> <i>User property</i> : winter.image.addOpti
automaticLaunchers	boolean	Enables the automatic generation of launc on the main classes extracted from the ap module. If enabled, a launcher is generate main classes other than the main launcher. <ul style="list-style-type: none"> <i>User property</i> : winter.app.automaticl <i>Default</i> : false
bindServices	boolean	Link in service provider modules and their dependencies. <ul style="list-style-type: none"> <i>User property</i> : winter.image.bindServ <i>Default</i> : false
compress	String	The compress level of the resulting image compression, 1=constant string sharing, 2 <ul style="list-style-type: none"> <i>User property</i> : winter.image.compress
configurationDirectory	File	A directory containing user-editable config that will be copied to the resulting image. <ul style="list-style-type: none"> <i>User property</i> : winter.image.configurationDirectory <i>Default</i> : \${project.basedir}/src/main,
copyright	String	The application copyright. <ul style="list-style-type: none"> <i>User property</i> : winter.app.copyright
description	String	The description of the application. <ul style="list-style-type: none"> <i>User property</i> : winter.app.description <i>Default</i> : \${project.description}
environment	Map	The container's environment variables.
excludeArtifactIds	String	Comma separated list of Artifact names to <ul style="list-style-type: none"> <i>User property</i> : winter.image.exclude,
excludeClassifiers	String	Comma Separated list of Classifiers to exc String indicates don't exclude anything (de <ul style="list-style-type: none"> <i>User property</i> : excludeClassifiers
excludeGroupIds	String	Comma separated list of GroupId Names t <ul style="list-style-type: none"> <i>User property</i> : winter.image.exclude,

excludeScope	String	Scope to exclude. An Empty string indicates (default). <ul style="list-style-type: none"> • <i>User property</i> : winter.image.excludeScope
ignoreSigningInformation	boolean	Suppress a fatal error when signed module is linked in the image. <ul style="list-style-type: none"> • <i>User property</i> : winter.image.ignoreSigningInformation • <i>Default</i> : false
imageFormat	ImageFormat	The format of the container image. <ul style="list-style-type: none"> • <i>User property</i> : winter.container.imageFormat • <i>Default</i> : Docker
includeArtifactIds	String	Comma separated list of Artifact names to include. Empty String indicates include everything. <ul style="list-style-type: none"> • <i>User property</i> : winter.image.includeArtifactIds
includeClassifiers	String	Comma Separated list of Classifiers to include. Empty String indicates include everything (default). <ul style="list-style-type: none"> • <i>User property</i> : winter.image.includeClassifiers
includeGroupIds	String	Comma separated list of GroupIds to include. Empty String indicates include everything (default). <ul style="list-style-type: none"> • <i>User property</i> : winter.image.includeGroupIds
includeScope	String	Scope to include. An Empty string indicates (default). The scopes being interpreted are as Maven sees them, not as specified in the summary: <ul style="list-style-type: none"> • <i>User property</i> : winter.image.includeScope
installDirectory	String	Absolute path of the installation directory of the application on OS X or Linux. Relative sub-installation location of the application such as 'Files' or 'AppData' on Windows. <ul style="list-style-type: none"> • <i>User property</i> : winter.app.installDirectory
jmodsOverrideDirectory	File	A directory containing module descriptors to modularize unnamed dependency module and override the ones that are otherwise generated. <ul style="list-style-type: none"> • <i>User property</i> : winter.image.jmodsOverrideDirectory • <i>Default</i> : \${project.basedir}/src/jmodsOverride
labels	Map	The labels to apply to the container image
launchers	List	A list of extra launchers to include in the native application.
legalDirectory	File	A directory containing legal notices that will be included in the application.

		<p>to the resulting image.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.legalDir • <i>Default</i> : \${project.basedir}/src/main,
licenseFile	File	<p>The path to the application license file.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.app.licenseFile • <i>Default</i> : \${project.basedir}/LICENSE
linuxConfiguration	LinuxConfiguration	Linux specific configuration.
macOSConfiguration	MacOSConfiguration	MacOS specific configuration.
manDirectory	File	<p>A directory containing man pages that will be included in the resulting image.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.manDir • <i>Default</i> : \${project.basedir}/src/main,
overWriteIfNewer	boolean	<p>Overwrite dependencies that don't exist or are older than the source.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.overWriteIfNewer • <i>Default</i> : true
ports	Set	<p>The ports exposed by the container at runtime. Format: port_number ['/' udp/tcp]</p>
projectMainClass	String	<p>The main class in the project module to use when building the project JMOD package.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.runtime.projectMainClass
registry	String	<p>The registry part of the target image reference. Format: registry/repository/name:tag</p> <p>as:</p> <p>`\${registry}/\${repository}/\${name}:\${tag}`</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.container.registry
registryPassword	String	<p>The password to use to authenticate to the registry.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.container.registryPassword
registryUsername	String	<p>The user name to use to authenticate to the registry.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.container.registryUsername
repository	String	<p>The repository part of the target image reference. Format: registry/repository/name:tag</p> <p>defined as:</p> <p>`\${registry}/\${repository}/\${name}:\${tag}`</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.container.repository
resolveProjectMainClass	boolean	<p>Resolve the project main class when not specified explicitly.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.runtime.projectMainClass • <i>Default</i> : false

resourceDirectory	File	The path to resources that override resulti resources. • <i>User property</i> : winter.app.resourceDi
skip	boolean	Skips the generation of the application. • <i>User property</i> : winter.app.skip
stripDebug	boolean	Strip debug information from the resulting • <i>User property</i> : winter.image.stripDeb • <i>Default</i> : true
stripNativeCommands	boolean	Strip native command (eg. java...) from th image. • <i>User property</i> : winter.image.stripNativeCommands • <i>Default</i> : true
user	String	The user and group used to run the contain as: user / uid [':' group / gid]
vendor	String	The application vendor. • <i>User property</i> : winter.app.vendor • <i>Default</i> : \${project.organization.name}
verbose	boolean	Enables verbose logging. • <i>User property</i> : winter.verbose • <i>Default</i> : false
vm	String	Select the HotSpot VM in the output image 'client' / 'server' / 'minimal' / 'all' • <i>User property</i> : winter.image.vm
volumes	Set	The container's mount points.
windowsConfiguration	WindowsConfiguration	Windows specific configuration.



Parameter details

<addModules>

The modules to add to the resulting image.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.addModules

<addOptions>

The options to prepend before any other options when invoking the JVM in the resulting image.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.addOptions

<attach>

Attach the resulting image archives to the project to install them in the local Maven repository and deploy them to remote repositories.

- **Type:** boolean
- **Required:** yes
- **User property:** winter.image.attach
- **Default:** true

<automaticLaunchers>

Enables the automatic generation of launchers based on the main classes extracted from the application module. If enabled, a launcher is generated for all main classes other than the main launcher.

- **Type:** boolean
- **Required:** no
- **User property:** winter.app.automaticLaunchers
- **Default:** false

<bindServices>

Link in service provider modules and their dependencies.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.bindServices
- **Default:** false

<compress>

The compress level of the resulting image: 0=No compression, 1=constant string sharing, 2=ZIP.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.compress

<configurationDirectory>

A directory containing user-editable configuration files that will be copied to the resulting image.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.configurationDirectory
- **Default:** \${project.basedir}/src/main/conf/

<copyright>

The application copyright.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.app.copyright

<description>

The description of the application.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.app.description
- **Default:** \${project.description}

<environment>

The container's environment variables.

- **Type:** java.util.Map
- **Required:** no

<excludeArtifactIds>

Comma separated list of Artifact names to exclude.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.excludeArtifactIds

<excludeClassifiers>

Comma Separated list of Classifiers to exclude. Empty String indicates don't exclude anything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** excludeClassifiers

<excludeGroupIds>

Comma separated list of GroupId Names to exclude.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.excludeGroupIds

<excludeScope>

Scope to exclude. An Empty string indicates no scopes (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.excludeScope

<executable>

The executable in the application image to use as image entry point. The specified name should correspond to a declared application image launchers or the project artifact id if no launcher was specified.

- **Type:** java.lang.String
- **Required:** yes
- **User property:** winter.app.executable
- **Default:** \${project.artifactId}

<formats>

A list of archive formats to generate (eg. zip, tar.gz...)

- **Type:** java.util.Set
- **Required:** yes
- **Default:** zip

<from>

The base container image.

- **Type:** java.lang.String
- **Required:** yes
- **User property:** winter.container.from
- **Default:** debian:buster-slim

<ignoreSigningInformation>

Suppress a fatal error when signed modular JARs are linked in the image.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.ignoreSigningInformation
- **Default:** false

<imageFormat>

The format of the container image.

- **Type:** com.google.cloud.tools.jib.api.buildplan.ImageFormat
- **Required:** no
- **User property:** winter.container.imageFormat
- **Default:** Docker

<includeArtifactIds>

Comma separated list of Artifact names to include. Empty String indicates include everything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeArtifactIds

<includeClassifiers>

Comma Separated list of Classifiers to include. Empty String indicates include everything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeClassifiers

<includeGroupIds>

Comma separated list of GroupIds to include. Empty String indicates include everything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeGroupIds

<includeScope>

Scope to include. An Empty string indicates all scopes (default). The scopes being interpreted are the scopes as Maven sees them, not as specified in the pom. In summary:

- runtime scope gives runtime and compile dependencies,
- compile scope gives compile, provided, and system dependencies,
- test (default) scope gives all dependencies,
- provided scope just gives provided dependencies,
- system scope just gives system dependencies.
- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeScope

<installDirectory>

Absolute path of the installation directory of the application on OS X or Linux. Relative sub-path of the installation location of the application such as 'Program Files' or 'AppData' on Windows.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.app.installDirectory

<jmodsOverrideDirectory>

A directory containing module descriptors to use to modularize unnamed dependency modules and which override the ones that are otherwise generated.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.jmodsOverrideDirectory
- **Default:** \${project.basedir}/src/jmods/

<labels>

The labels to apply to the container image.

- **Type:** java.util.Map
- **Required:** no

<launchers>

A list of extra launchers to include in the resulting application.

- **Type:** java.util.List
- **Required:** no

<legalDirectory>

A directory containing legal notices that will be copied to the resulting image.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.legalDirectory
- **Default:** \${project.basedir}/src/main/legal/

<licenseFile>

The path to the application license file.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.app.licenseFile
- **Default:** \${project.basedir}/LICENSE

<linuxConfiguration>

Linux specific configuration.

- **Type:**
io.winterframework.tool.maven.internal.task.CreateProjectApplicationTask\$LinuxConfigurati
- **Required:** no

<macOSConfiguration>

MacOS specific configuration.

- **Type:**
io.winterframework.tool.maven.internal.task.CreateProjectApplicationTask\$MacOSConfigura
- **Required:** no

<manDirectory>

A directory containing man pages that will be copied to the resulting image.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.manDirectory
- **Default:** \${project.basedir}/src/main/man/

<overWriteIfNewer>

Overwrite dependencies that don't exist or are older than the source.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.overWriteIfNewer
- **Default:** true

<ports>

The ports exposed by the container at runtime defined as: port_number ['/' udp/tcp]

- **Type:** java.util.Set
- **Required:** no

<projectMainClass>

The main class in the project module to use when building the project JMOD package.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.runtime.projectMainClass

<registry>

The registry part of the target image reference defined as:
\${registry}/\${repository}/\${name}:\${project.version}

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.container.registry

<registryPassword>

The password to use to authenticate to the registry.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.container.registry.password

<registryUsername>

The user name to use to authenticate to the registry.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.container.registry.username

<repository>

The repository part of the target image reference defined as:
`${registry}/${repository}/${name}:${project.version}`

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.container.repository

<resolveProjectMainClass>

Resolve the project main class when not specified explicitly.

- **Type:** boolean
- **Required:** no
- **User property:** winter.runtime.projectMainClass
- **Default:** false

<resourceDirectory>

The path to resources that override resulting package resources.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.app.resourceDirectory

<skip>

Skips the generation of the application.

- **Type:** boolean
- **Required:** no
- **User property:** winter.app.skip

<stripDebug>

Strip debug information from the resulting image.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.stripDebug
- **Default:** true

<stripNativeCommands>

Strip native command (eg. java...) from the resulting image.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.stripNativeCommands

- **Default:** true

<user>

The user and group used to run the container defined as: user / uid [':' group / gid]

- **Type:** java.lang.String
- **Required:** no

<vendor>

The application vendor.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.app.vendor
- **Default:** \${project.organization.name}

<verbose>

Enables verbose logging.

- **Type:** boolean
- **Required:** no
- **User property:** winter.verbose
- **Default:** false

<vm>

Select the HotSpot VM in the output image defined as: 'client' / 'server' / 'minimal' / 'all'

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.vm

<volumes>

The container's mount points.

- **Type:** java.util.Set
- **Required:** no

<windowsConfiguration>

Windows specific configuration.

- **Type:**
io.winterframework.tool.maven.internal.task.CreateProjectApplicationTask\$WindowsConfigu
- **Required:** no

winter:build-image-docker

Full name:

io.winterframework.tool:winter-maven-plugin:1.0.0:build-image-docker

Description:

Builds a Docker container image to a local Docker daemon.

Attributes:

- Requires a Maven project to be executed.
- Requires dependency resolution of artifacts in scope: compile+runtime.
- Requires dependency collection of artifacts in scope: compile+runtime.
- Since version: 1.0.
- Binds by default to the lifecycle phase: install.

Required parameters

Name	Type	Default
attach	boolean	Attach the resulting image archives to the project to install them in the local Maven repository and deploy them to remote repositories. <ul style="list-style-type: none">• <i>User property</i> : winter.image.attach• <i>Default</i> : true
executable	String	The executable in the application image to use as image entry point. The specified name should correspond to a declared application image launchers or the project artifact id if no launcher was specified. <ul style="list-style-type: none">• <i>User property</i> : winter.app.executable• <i>Default</i> : \${project.artifactId}
formats	Set	A list of archive formats to generate (eg. zip, tar.gz...) <ul style="list-style-type: none">• <i>Default</i> : zip
from	String	The base container image. <ul style="list-style-type: none">• <i>User property</i> : winter.container.from• <i>Default</i> : debian:buster-slim

Optional parameters

Name	Type	Description
addModules	String	The modules to add to the resulting image. <ul style="list-style-type: none"> <i>User property</i> : winter.image.addMod
addOptions	String	The options to prepend before any other o invoking the JVM in the resulting image. <ul style="list-style-type: none"> <i>User property</i> : winter.image.addOpti
automaticLaunchers	boolean	Enables the automatic generation of launc on the main classes extracted from the ap module. If enabled, a launcher is generate main classes other than the main launcher. <ul style="list-style-type: none"> <i>User property</i> : winter.app.automaticl <i>Default</i> : false
bindServices	boolean	Link in service provider modules and their dependencies. <ul style="list-style-type: none"> <i>User property</i> : winter.image.bindServ <i>Default</i> : false
compress	String	The compress level of the resulting image compression, 1=constant string sharing, 2 <ul style="list-style-type: none"> <i>User property</i> : winter.image.compress
configurationDirectory	File	A directory containing user-editable config that will be copied to the resulting image. <ul style="list-style-type: none"> <i>User property</i> : winter.image.configurationDirectory <i>Default</i> : \${project.basedir}/src/main,
copyright	String	The application copyright. <ul style="list-style-type: none"> <i>User property</i> : winter.app.copyright
description	String	The description of the application. <ul style="list-style-type: none"> <i>User property</i> : winter.app.description <i>Default</i> : \${project.description}
dockerEnvironment	Map	The Docker environment variables used by CLI executable.
dockerExecutable	File	The path to the Docker CLI executable use image in the Docker daemon. <ul style="list-style-type: none"> <i>User property</i> : winter.container.docke
environment	Map	The container's environment variables.
excludeArtifactIds	String	Comma separated list of Artifact names to <ul style="list-style-type: none"> <i>User property</i> : winter.image.exclude,

excludeClassifiers	String	Comma Separated list of Classifiers to exclude. An Empty String indicates don't exclude anything (default). <ul style="list-style-type: none"> <i>User property</i> : excludeClassifiers
excludeGroupIds	String	Comma separated list of GroupId Names to exclude. An Empty String indicates include everything (default). <ul style="list-style-type: none"> <i>User property</i> : winter.image.excludeGroupIds
excludeScope	String	Scope to exclude. An Empty string indicates include everything (default). <ul style="list-style-type: none"> <i>User property</i> : winter.image.excludeScope
ignoreSigningInformation	boolean	Suppress a fatal error when signed modules are not found or linked in the image. <ul style="list-style-type: none"> <i>User property</i> : winter.image.ignoreSigningInformation <i>Default</i> : false
imageFormat	ImageFormat	The format of the container image. <ul style="list-style-type: none"> <i>User property</i> : winter.container.imageFormat <i>Default</i> : Docker
includeArtifactIds	String	Comma separated list of Artifact names to include. An Empty String indicates include everything (default). <ul style="list-style-type: none"> <i>User property</i> : winter.image.includeArtifactIds
includeClassifiers	String	Comma Separated list of Classifiers to include. An Empty String indicates include everything (default). <ul style="list-style-type: none"> <i>User property</i> : winter.image.includeClassifiers
includeGroupIds	String	Comma separated list of GroupIds to include. An Empty String indicates include everything (default). <ul style="list-style-type: none"> <i>User property</i> : winter.image.includeGroupIds
includeScope	String	Scope to include. An Empty string indicates include everything (default). The scopes being interpreted are as Maven sees them, not as specified in the pom. For a summary: <ul style="list-style-type: none"> <i>User property</i> : winter.image.includeScope
installDirectory	String	Absolute path of the installation directory for the application on OS X or Linux. Relative sub-installation location of the application such as 'Files' or 'AppData' on Windows. <ul style="list-style-type: none"> <i>User property</i> : winter.app.installDirectory
jmodsOverrideDirectory	File	A directory containing module descriptors to override the ones that are otherwise generated. This is useful to modularize unnamed dependency module override the ones that are otherwise generated. <ul style="list-style-type: none"> <i>User property</i> : winter.image.jmodsOverrideDirectory <i>Default</i> : \${project.basedir}/src/jmods

labels	Map	The labels to apply to the container image
launchers	List	A list of extra launchers to include in the resulting application.
legalDirectory	File	A directory containing legal notices that will be added to the resulting image. <ul style="list-style-type: none"> • <i>User property</i> : winter.image.legalDirectory • <i>Default</i> : \${project.basedir}/src/main/resources/legal
licenseFile	File	The path to the application license file. <ul style="list-style-type: none"> • <i>User property</i> : winter.app.licenseFile • <i>Default</i> : \${project.basedir}/LICENSE
linuxConfiguration	LinuxConfiguration	Linux specific configuration.
macOSConfiguration	MacOSConfiguration	MacOS specific configuration.
manDirectory	File	A directory containing man pages that will be added to the resulting image. <ul style="list-style-type: none"> • <i>User property</i> : winter.image.manDirectory • <i>Default</i> : \${project.basedir}/src/main/resources/man
overwriteIfNewer	boolean	Overwrite dependencies that don't exist or are older than the source. <ul style="list-style-type: none"> • <i>User property</i> : winter.image.overwriteIfNewer • <i>Default</i> : true
ports	Set	The ports exposed by the container at runtime. Format: port_number ['/' udp/tcp]
projectMainClass	String	The main class in the project module to use when building the project JMOD package. <ul style="list-style-type: none"> • <i>User property</i> : winter.runtime.projectMainClass
registry	String	The registry part of the target image reference. Defined as: <pre> \${registry}/\${repository}/\${name}:\${project.version} </pre> <ul style="list-style-type: none"> • <i>User property</i> : winter.container.registry
repository	String	The repository part of the target image reference. Defined as: <pre> \${registry}/\${repository}/\${name}:\${project.version} </pre> <ul style="list-style-type: none"> • <i>User property</i> : winter.container.repository
resolveProjectMainClass	boolean	Resolve the project main class when not specified explicitly. <ul style="list-style-type: none"> • <i>User property</i> : winter.runtime.projectMainClass • <i>Default</i> : false

resourceDirectory	File	The path to resources that override resulti resources. • <i>User property</i> : winter.app.resourceDi
skip	boolean	Skips the generation of the application. • <i>User property</i> : winter.app.skip
stripDebug	boolean	Strip debug information from the resulting • <i>User property</i> : winter.image.stripDeb • <i>Default</i> : true
stripNativeCommands	boolean	Strip native command (eg. java...) from th image. • <i>User property</i> : winter.image.stripNativeCommands • <i>Default</i> : true
user	String	The user and group used to run the contain as: user / uid [':' group / gid]
vendor	String	The application vendor. • <i>User property</i> : winter.app.vendor • <i>Default</i> : \${project.organization.name}
verbose	boolean	Enables verbose logging. • <i>User property</i> : winter.verbose • <i>Default</i> : false
vm	String	Select the HotSpot VM in the output image 'client' / 'server' / 'minimal' / 'all' • <i>User property</i> : winter.image.vm
volumes	Set	The container's mount points.
windowsConfiguration	WindowsConfiguration	Windows specific configuration.

Parameter details

<addModules>

The modules to add to the resulting image.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.addModules

<addOptions>

The options to prepend before any other options when invoking the JVM in the resulting image.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.addOptions

<attach>

Attach the resulting image archives to the project to install them in the local Maven repository and deploy them to remote repositories.

- **Type:** boolean
- **Required:** yes
- **User property:** winter.image.attach
- **Default:** true

<automaticLaunchers>

Enables the automatic generation of launchers based on the main classes extracted from the application module. If enabled, a launcher is generated for all main classes other than the main launcher.

- **Type:** boolean
- **Required:** no
- **User property:** winter.app.automaticLaunchers
- **Default:** false

<bindServices>

Link in service provider modules and their dependencies.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.bindServices
- **Default:** false

<compress>

The compress level of the resulting image: 0=No compression, 1=constant string sharing, 2=ZIP.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.compress

<configurationDirectory>

A directory containing user-editable configuration files that will be copied to the resulting image.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.configurationDirectory
- **Default:** \${project.basedir}/src/main/conf/

<copyright>

The application copyright.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.app.copyright

<description>

The description of the application.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.app.description
- **Default:** \${project.description}

<dockerEnvironment>

The Docker environment variables used by the Docker CLI executable.

- **Type:** java.util.Map
- **Required:** no

<dockerExecutable>

The path to the Docker CLI executable used to load the image in the Docker daemon.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.container.docker.executable

<environment>

The container's environment variables.

- **Type:** java.util.Map
- **Required:** no

<excludeArtifactIds>

Comma separated list of Artifact names to exclude.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.excludeArtifactIds

<excludeClassifiers>

Comma Separated list of Classifiers to exclude. Empty String indicates don't exclude anything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** excludeClassifiers

<excludeGroupIds>

Comma separated list of GroupId Names to exclude.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.excludeGroupIds

<excludeScope>

Scope to exclude. An Empty string indicates no scopes (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.excludeScope

<executable>

The executable in the application image to use as image entry point. The specified name should correspond to a declared application image launchers or the project artifact id if no launcher was specified.

- **Type:** java.lang.String
- **Required:** yes
- **User property:** winter.app.executable
- **Default:** \${project.artifactId}

<formats>

A list of archive formats to generate (eg. zip, tar.gz...)

- **Type:** java.util.Set
- **Required:** yes
- **Default:** zip

<from>

The base container image.

- **Type:** java.lang.String
- **Required:** yes
- **User property:** winter.container.from
- **Default:** debian:buster-slim

<ignoreSigningInformation>

Suppress a fatal error when signed modular JARs are linked in the image.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.ignoreSigningInformation
- **Default:** false

<imageFormat>

The format of the container image.

- **Type:** com.google.cloud.tools.jib.api.buildplan.ImageFormat
- **Required:** no
- **User property:** winter.container.imageFormat
- **Default:** Docker

<includeArtifactIds>

Comma separated list of Artifact names to include. Empty String indicates include everything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeArtifactIds

<includeClassifiers>

Comma Separated list of Classifiers to include. Empty String indicates include everything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeClassifiers

<includeGroupIds>

Comma separated list of GroupIds to include. Empty String indicates include everything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeGroupIds

<includeScope>

Scope to include. An Empty string indicates all scopes (default). The scopes being interpreted are the scopes as Maven sees them, not as specified in the pom. In summary:

- runtime scope gives runtime and compile dependencies,
- compile scope gives compile, provided, and system dependencies,
- test (default) scope gives all dependencies,
- provided scope just gives provided dependencies,
- system scope just gives system dependencies.
- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeScope

<installDirectory>

Absolute path of the installation directory of the application on OS X or Linux. Relative sub-path of the installation location of the application such as 'Program Files' or 'AppData' on Windows.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.app.installDirectory

<jmodsOverrideDirectory>

A directory containing module descriptors to use to modularize unnamed dependency modules and which override the ones that are otherwise generated.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.jmodsOverrideDirectory
- **Default:** \${project.basedir}/src/jmods/

<labels>

The labels to apply to the container image.

- **Type:** java.util.Map
- **Required:** no

<launchers>

A list of extra launchers to include in the resulting application.

- **Type:** java.util.List
- **Required:** no

<legalDirectory>

A directory containing legal notices that will be copied to the resulting image.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.legalDirectory
- **Default:** \${project.basedir}/src/main/legal/

<licenseFile>

The path to the application license file.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.app.licenseFile
- **Default:** \${project.basedir}/LICENSE

<linuxConfiguration>

Linux specific configuration.

- **Type:**
io.winterframework.tool.maven.internal.task.CreateProjectApplicationTask\$LinuxConfigurati
- **Required:** no

<macOSConfiguration>

MacOS specific configuration.

- **Type:** io.winterframework.tool.maven.internal.task.CreateProjectApplicationTask\$MacOSConfigura
- **Required:** no

<manDirectory>

A directory containing man pages that will be copied to the resulting image.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.manDirectory
- **Default:** \${project.basedir}/src/main/man/

<overwriteIfNewer>

Overwrite dependencies that don't exist or are older than the source.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.overwriteIfNewer
- **Default:** true

<ports>

The ports exposed by the container at runtime defined as: port_number ['/' udp/tcp]

- **Type:** java.util.Set
- **Required:** no

<projectMainClass>

The main class in the project module to use when building the project JMOD package.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.runtime.projectMainClass

<registry>

The registry part of the target image reference defined as:
\${registry}/\${repository}/\${name}:\${project.version}

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.container.registry

<repository>

The repository part of the target image reference defined as:
`${registry}/${repository}/${name}:${project.version}`

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.container.repository

<resolveProjectMainClass>

Resolve the project main class when not specified explicitly.

- **Type:** boolean
- **Required:** no
- **User property:** winter.runtime.projectMainClass
- **Default:** false

<resourceDirectory>

The path to resources that override resulting package resources.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.app.resourceDirectory

<skip>

Skips the generation of the application.

- **Type:** boolean
- **Required:** no
- **User property:** winter.app.skip

<stripDebug>

Strip debug information from the resulting image.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.stripDebug
- **Default:** true

<stripNativeCommands>

Strip native command (eg. java...) from the resulting image.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.stripNativeCommands

- **Default:** true

<user>

The user and group used to run the container defined as: user / uid [':' group / gid]

- **Type:** java.lang.String
- **Required:** no

<vendor>

The application vendor.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.app.vendor
- **Default:** \${project.organization.name}

<verbose>

Enables verbose logging.

- **Type:** boolean
- **Required:** no
- **User property:** winter.verbose
- **Default:** false

<vm>

Select the HotSpot VM in the output image defined as: 'client' / 'server' / 'minimal' / 'all'

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.vm

<volumes>

The container's mount points.

- **Type:** java.util.Set
- **Required:** no

<windowsConfiguration>

Windows specific configuration.

- **Type:**
io.winterframework.tool.maven.internal.task.CreateProjectApplicationTask\$WindowsConfigu
- **Required:** no

winter:build-image-tar

Full name:

io.winterframework.tool:winter-maven-plugin:1.0.0:build-image-tar

Description:

Builds a container image to a TAR archive that can be later loaded into Docker:

```
$ docker load --input target/<image>.tar
```

Attributes:

- Requires a Maven project to be executed.
- Requires dependency resolution of artifacts in scope: compile+runtime.
- Requires dependency collection of artifacts in scope: compile+runtime.
- Since version: 1.0.
- Binds by default to the lifecycle phase: package.

Required parameters

Name	Type	Default
attach	boolean	Attach the resulting image archives to the project to install them in the local Maven repository and deploy them to remote repositories. <ul style="list-style-type: none">• <i>User property</i> : winter.image.attach• <i>Default</i> : true
executable	String	The executable in the application image to use as image entry point. The specified name should correspond to a declared application image launchers or the project artifact id if no launcher was specified. <ul style="list-style-type: none">• <i>User property</i> : winter.app.executable• <i>Default</i> : \${project.artifactId}
formats	Set	A list of archive formats to generate (eg. zip, tar.gz...) <ul style="list-style-type: none">• <i>Default</i> : zip
from	String	The base container image. <ul style="list-style-type: none">• <i>User property</i> : winter.container.from• <i>Default</i> : debian:buster-slim

Optional parameters

Name	Type	Description
addModules	String	The modules to add to the resulting image. <ul style="list-style-type: none"> <i>User property</i> : winter.image.addMod
addOptions	String	The options to prepend before any other o invoking the JVM in the resulting image. <ul style="list-style-type: none"> <i>User property</i> : winter.image.addOpti
automaticLaunchers	boolean	Enables the automatic generation of launc on the main classes extracted from the ap module. If enabled, a launcher is generate main classes other than the main launcher. <ul style="list-style-type: none"> <i>User property</i> : winter.app.automaticl <i>Default</i> : false
bindServices	boolean	Link in service provider modules and their dependencies. <ul style="list-style-type: none"> <i>User property</i> : winter.image.bindServ <i>Default</i> : false
compress	String	The compress level of the resulting image compression, 1=constant string sharing, 2 <ul style="list-style-type: none"> <i>User property</i> : winter.image.compress
configurationDirectory	File	A directory containing user-editable config that will be copied to the resulting image. <ul style="list-style-type: none"> <i>User property</i> : winter.image.configurationDirectory <i>Default</i> : \${project.basedir}/src/main,
copyright	String	The application copyright. <ul style="list-style-type: none"> <i>User property</i> : winter.app.copyright
description	String	The description of the application. <ul style="list-style-type: none"> <i>User property</i> : winter.app.description <i>Default</i> : \${project.description}
environment	Map	The container's environment variables.
excludeArtifactIds	String	Comma separated list of Artifact names to <ul style="list-style-type: none"> <i>User property</i> : winter.image.exclude,
excludeClassifiers	String	Comma Separated list of Classifiers to exc String indicates don't exclude anything (de <ul style="list-style-type: none"> <i>User property</i> : excludeClassifiers
excludeGroupIds	String	Comma separated list of GroupId Names t <ul style="list-style-type: none"> <i>User property</i> : winter.image.exclude,

excludeScope	String	Scope to exclude. An Empty string indicates (default). <ul style="list-style-type: none"> <i>User property</i> : winter.image.excludeScope
ignoreSigningInformation	boolean	Suppress a fatal error when signed module is linked in the image. <ul style="list-style-type: none"> <i>User property</i> : winter.image.ignoreSigningInformation <i>Default</i> : false
imageFormat	ImageFormat	The format of the container image. <ul style="list-style-type: none"> <i>User property</i> : winter.container.imageFormat <i>Default</i> : Docker
includeArtifactIds	String	Comma separated list of Artifact names to include. Empty String indicates include everything. <ul style="list-style-type: none"> <i>User property</i> : winter.image.includeArtifactIds
includeClassifiers	String	Comma Separated list of Classifiers to include. Empty String indicates include everything (default). <ul style="list-style-type: none"> <i>User property</i> : winter.image.includeClassifiers
includeGroupIds	String	Comma separated list of GroupIds to include. Empty String indicates include everything (default). <ul style="list-style-type: none"> <i>User property</i> : winter.image.includeGroupIds
includeScope	String	Scope to include. An Empty string indicates (default). The scopes being interpreted are as Maven sees them, not as specified in the summary: <ul style="list-style-type: none"> <i>User property</i> : winter.image.includeScope
installDirectory	String	Absolute path of the installation directory of the application on OS X or Linux. Relative sub-installation location of the application such as 'Files' or 'AppData' on Windows. <ul style="list-style-type: none"> <i>User property</i> : winter.app.installDirectory
jmodsOverrideDirectory	File	A directory containing module descriptors to modularize unnamed dependency module and override the ones that are otherwise generated. <ul style="list-style-type: none"> <i>User property</i> : winter.image.jmodsOverrideDirectory <i>Default</i> : \${project.basedir}/src/jmods
labels	Map	The labels to apply to the container image
launchers	List	A list of extra launchers to include in the native application.
legalDirectory	File	A directory containing legal notices that will be included in the application.

		<p>to the resulting image.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.legalDir • <i>Default</i> : \${project.basedir}/src/main,
licenseFile	File	<p>The path to the application license file.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.app.licenseFile • <i>Default</i> : \${project.basedir}/LICENSE
linuxConfiguration	LinuxConfiguration	Linux specific configuration.
macOSConfiguration	MacOSConfiguration	MacOS specific configuration.
manDirectory	File	<p>A directory containing man pages that will be included in the resulting image.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.manDir • <i>Default</i> : \${project.basedir}/src/main,
overWriteIfNewer	boolean	<p>Overwrite dependencies that don't exist or are older than the source.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.overWriteIfNewer • <i>Default</i> : true
ports	Set	<p>The ports exposed by the container at runtime. Format: port_number ['/' udp/tcp]</p>
projectMainClass	String	<p>The main class in the project module to use when building the project JMOD package.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.runtime.projectMainClass
registry	String	<p>The registry part of the target image reference. Format: registry/repository/name:tag</p> <p>as:</p> <p><code>\${registry}/\${repository}/\${name}:\${tag}</code></p> <ul style="list-style-type: none"> • <i>User property</i> : winter.container.registry
repository	String	<p>The repository part of the target image reference. Format: registry/repository/name:tag</p> <p>defined as:</p> <p><code>\${registry}/\${repository}/\${name}:\${tag}</code></p> <ul style="list-style-type: none"> • <i>User property</i> : winter.container.repository
resolveProjectMainClass	boolean	<p>Resolve the project main class when not specified explicitly.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.runtime.projectMainClass • <i>Default</i> : false
resourceDirectory	File	<p>The path to resources that override resulti resources.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.app.resourceDir
skip	boolean	<p>Skips the generation of the application.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.app.skip

stripDebug	boolean	Strip debug information from the resulting image. <ul style="list-style-type: none">• <i>User property</i> : winter.image.stripDebug• <i>Default</i> : true
stripNativeCommands	boolean	Strip native command (eg. java...) from the image. <ul style="list-style-type: none">• <i>User property</i> : winter.image.stripNativeCommands• <i>Default</i> : true
user	String	The user and group used to run the container as: user / uid [':' group / gid]
vendor	String	The application vendor. <ul style="list-style-type: none">• <i>User property</i> : winter.app.vendor• <i>Default</i> : \${project.organization.name}
verbose	boolean	Enables verbose logging. <ul style="list-style-type: none">• <i>User property</i> : winter.verbose• <i>Default</i> : false
vm	String	Select the HotSpot VM in the output image: 'client' / 'server' / 'minimal' / 'all' <ul style="list-style-type: none">• <i>User property</i> : winter.image.vm
volumes	Set	The container's mount points.
windowsConfiguration	WindowsConfiguration	Windows specific configuration.

Parameter details

<addModules>

The modules to add to the resulting image.

- **Type**: java.lang.String
- **Required**: no
- **User property**: winter.image.addModules

<addOptions>

The options to prepend before any other options when invoking the JVM in the resulting image.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.addOptions

<attach>

Attach the resulting image archives to the project to install them in the local Maven repository and deploy them to remote repositories.

- **Type:** boolean
- **Required:** yes
- **User property:** winter.image.attach
- **Default:** true

<automaticLaunchers>

Enables the automatic generation of launchers based on the main classes extracted from the application module. If enabled, a launcher is generated for all main classes other than the main launcher.

- **Type:** boolean
- **Required:** no
- **User property:** winter.app.automaticLaunchers
- **Default:** false

<bindServices>

Link in service provider modules and their dependencies.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.bindServices
- **Default:** false

<compress>

The compress level of the resulting image: 0=No compression, 1=constant string sharing, 2=ZIP.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.compress

<configurationDirectory>

A directory containing user-editable configuration files that will be copied to the resulting image.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.configurationDirectory
- **Default:** \${project.basedir}/src/main/conf/

<copyright>

The application copyright.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.app.copyright

<description>

The description of the application.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.app.description
- **Default:** \${project.description}

<environment>

The container's environment variables.

- **Type:** java.util.Map
- **Required:** no

<excludeArtifactIds>

Comma separated list of Artifact names to exclude.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.excludeArtifactIds

<excludeClassifiers>

Comma Separated list of Classifiers to exclude. Empty String indicates don't exclude anything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** excludeClassifiers

<excludeGroupIds>

Comma separated list of GroupId Names to exclude.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.excludeGroupIds

<excludeScope>

Scope to exclude. An Empty string indicates no scopes (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.excludeScope

<executable>

The executable in the application image to use as image entry point. The specified name should correspond to a declared application image launchers or the project artifact id if no launcher was specified.

- **Type:** java.lang.String
- **Required:** yes
- **User property:** winter.app.executable
- **Default:** \${project.artifactId}

<formats>

A list of archive formats to generate (eg. zip, tar.gz...)

- **Type:** java.util.Set
- **Required:** yes
- **Default:** zip

<from>

The base container image.

- **Type:** java.lang.String
- **Required:** yes
- **User property:** winter.container.from
- **Default:** debian:buster-slim

<ignoreSigningInformation>

Suppress a fatal error when signed modular JARs are linked in the image.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.ignoreSigningInformation
- **Default:** false

<imageFormat>

The format of the container image.

- **Type:** com.google.cloud.tools.jib.api.buildplan.ImageFormat
- **Required:** no
- **User property:** winter.container.imageFormat
- **Default:** Docker

<includeArtifactIds>

Comma separated list of Artifact names to include. Empty String indicates include everything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeArtifactIds

<includeClassifiers>

Comma Separated list of Classifiers to include. Empty String indicates include everything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeClassifiers

<includeGroupIds>

Comma separated list of GroupIds to include. Empty String indicates include everything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeGroupIds

<includeScope>

Scope to include. An Empty string indicates all scopes (default). The scopes being interpreted are the scopes as Maven sees them, not as specified in the pom. In summary:

- runtime scope gives runtime and compile dependencies,
- compile scope gives compile, provided, and system dependencies,
- test (default) scope gives all dependencies,
- provided scope just gives provided dependencies,
- system scope just gives system dependencies.
- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeScope

<installDirectory>

Absolute path of the installation directory of the application on OS X or Linux. Relative sub-path of the installation location of the application such as 'Program Files' or 'AppData' on Windows.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.app.installDirectory

<jmodsOverrideDirectory>

A directory containing module descriptors to use to modularize unnamed dependency modules and which override the ones that are otherwise generated.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.jmodsOverrideDirectory
- **Default:** \${project.basedir}/src/jmods/

<labels>

The labels to apply to the container image.

- **Type:** java.util.Map
- **Required:** no

<launchers>

A list of extra launchers to include in the resulting application.

- **Type:** java.util.List
- **Required:** no

<legalDirectory>

A directory containing legal notices that will be copied to the resulting image.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.legalDirectory
- **Default:** \${project.basedir}/src/main/legal/

<licenseFile>

The path to the application license file.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.app.licenseFile
- **Default:** \${project.basedir}/LICENSE

<linuxConfiguration>

Linux specific configuration.

- **Type:**
io.winterframework.tool.maven.internal.task.CreateProjectApplicationTask\$LinuxConfigurati
- **Required:** no

<macOSConfiguration>

MacOS specific configuration.

- **Type:**
io.winterframework.tool.maven.internal.task.CreateProjectApplicationTask\$MacOSConfigura
- **Required:** no

<manDirectory>

A directory containing man pages that will be copied to the resulting image.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.manDirectory
- **Default:** \${project.basedir}/src/main/man/

<overWriteIfNewer>

Overwrite dependencies that don't exist or are older than the source.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.overWriteIfNewer
- **Default:** true

<ports>

The ports exposed by the container at runtime defined as: port_number ['/' udp/tcp]

- **Type:** java.util.Set
- **Required:** no

<projectMainClass>

The main class in the project module to use when building the project JMOD package.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.runtime.projectMainClass

<registry>

The registry part of the target image reference defined as:

`${registry}/${repository}/${name}:${project.version}`

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.container.registry

<repository>

The repository part of the target image reference defined as:

`${registry}/${repository}/${name}:${project.version}`

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.container.repository

<resolveProjectMainClass>

Resolve the project main class when not specified explicitly.

- **Type:** boolean
- **Required:** no
- **User property:** winter.runtime.projectMainClass
- **Default:** false

<resourceDirectory>

The path to resources that override resulting package resources.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.app.resourceDirectory

<skip>

Skips the generation of the application.

- **Type:** boolean
- **Required:** no
- **User property:** winter.app.skip

<stripDebug>

Strip debug information from the resulting image.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.stripDebug
- **Default:** true

<stripNativeCommands>

Strip native command (eg. java...) from the resulting image.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.stripNativeCommands
- **Default:** true

<user>

The user and group used to run the container defined as: user / uid [':' group / gid]

- **Type:** java.lang.String
- **Required:** no

<vendor>

The application vendor.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.app.vendor
- **Default:** \${project.organization.name}

<verbose>

Enables verbose logging.

- **Type:** boolean
- **Required:** no
- **User property:** winter.verbose
- **Default:** false

<vm>

Select the HotSpot VM in the output image defined as: 'client' / 'server' / 'minimal' / 'all'

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.vm

<volumes>

The container's mount points.

- **Type:** java.util.Set
- **Required:** no

<windowsConfiguration>

Windows specific configuration.

- **Type:**
io.winterframework.tool.maven.internal.task.CreateProjectApplicationTask\$WindowsConfigu
- **Required:** no

winter:build-runtime

Full name:

io.winterframework.tool:winter-maven-plugin:1.0.0:build-runtime

Description:

Builds the project runtime image.

A runtime image is a custom Java runtime containing a set of modules and their dependencies.

Attributes:

- Requires a Maven project to be executed.
- Requires dependency resolution of artifacts in scope: compile+runtime.
- Requires dependency collection of artifacts in scope: compile+runtime.
- Since version: 1.0.

- Binds by default to the lifecycle phase: package.

Required parameters

Name	Type	Default
attach	boolean	Attach the resulting image archives to the project to install them in the local Maven repository and deploy them to remote repositories. <ul style="list-style-type: none">• <i>User property</i> : winter.image.attach• <i>Default</i> : true
formats	Set	A list of archive formats to generate (eg. zip, tar.gz...) <ul style="list-style-type: none">• <i>Default</i> : zip

Optional parameters

Name	Type	Description
addModules	String	The modules to add to the resulting image. <ul style="list-style-type: none"> <i>User property</i> : winter.image.addModules
addOptions	String	The options to prepend before any other options when invoking the JVM in the resulting image. <ul style="list-style-type: none"> <i>User property</i> : winter.image.addOptions
bindServices	boolean	Link in service provider modules and their dependencies. <ul style="list-style-type: none"> <i>User property</i> : winter.image.bindServices <i>Default</i> : false
compress	String	The compress level of the resulting image: 0=No compression, 1=constant string sharing, 2=ZIP. <ul style="list-style-type: none"> <i>User property</i> : winter.image.compress
configurationDirectory	File	A directory containing user-editable configuration files that will be copied to the resulting image. <ul style="list-style-type: none"> <i>User property</i> : winter.image.configurationDirectory <i>Default</i> : \${project.basedir}/src/main/conf/
excludeArtifactIds	String	Comma separated list of Artifact names to exclude. <ul style="list-style-type: none"> <i>User property</i> : winter.image.excludeArtifactIds
excludeClassifiers	String	Comma Separated list of Classifiers to exclude. Empty String indicates don't exclude anything (default). <ul style="list-style-type: none"> <i>User property</i> : excludeClassifiers
excludeGroupIds	String	Comma separated list of GroupId Names to exclude. <ul style="list-style-type: none"> <i>User property</i> : winter.image.excludeGroupIds
excludeScope	String	Scope to exclude. An Empty string indicates no scopes (default). <ul style="list-style-type: none"> <i>User property</i> : winter.image.excludeScope
ignoreSigningInformation	boolean	Suppress a fatal error when signed modular JARs are linked in the image. <ul style="list-style-type: none"> <i>User property</i> : winter.image.ignoreSigningInformation <i>Default</i> : false
includeArtifactIds	String	Comma separated list of Artifact names to include. Empty String indicates include everything (default). <ul style="list-style-type: none"> <i>User property</i> : winter.image.includeArtifactIds
includeClassifiers	String	Comma Separated list of Classifiers to include. Empty String indicates include everything (default).

		<ul style="list-style-type: none"> • <i>User property</i> : winter.image.includeClassifiers
includeGroupIds	String	<p>Comma separated list of GroupIds to include. Empty String indicates include everything (default).</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.includeGroupIds
includeScope	String	<p>Scope to include. An Empty string indicates all scopes (default). The scopes being interpreted are the scopes as Maven sees them, not as specified in the pom. In summary:</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.includeScope
jmodsOverrideDirectory	File	<p>A directory containing module descriptors to use to modularize unnamed dependency modules and which override the ones that are otherwise generated.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.jmodsOverrideDirectory • <i>Default</i> : \${project.basedir}/src/jmods/
launchers	List	<p>A list of launchers to include in the resulting runtime.</p>
legalDirectory	File	<p>A directory containing legal notices that will be copied to the resulting image.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.legalDirectory • <i>Default</i> : \${project.basedir}/src/main/legal/
manDirectory	File	<p>A directory containing man pages that will be copied to the resulting image.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.manDirectory • <i>Default</i> : \${project.basedir}/src/main/man/
overWritelfNewer	boolean	<p>Overwrite dependencies that don't exist or are older than the source.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.overWritelfNewer • <i>Default</i> : true
projectMainClass	String	<p>The main class in the project module to use when building the project JMOD package.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.runtime.projectMainClass
resolveProjectMainClass	boolean	<p>Resolve the project main class when not specified explicitly.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.runtime.projectMainClass • <i>Default</i> : false
skip	boolean	<p>Skips the generation of the runtime.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.runtime.skip
stripDebug	boolean	<p>Strip debug information from the resulting image.</p> <ul style="list-style-type: none"> • <i>User property</i> : winter.image.stripDebug • <i>Default</i> : true

stripNativeCommands	boolean	Strip native command (eg. java...) from the resulting image. <ul style="list-style-type: none"> • <i>User property</i> : winter.image.stripNativeCommands • <i>Default</i> : true
verbose	boolean	Enables verbose logging. <ul style="list-style-type: none"> • <i>User property</i> : winter.verbose • <i>Default</i> : false
vm	String	Select the HotSpot VM in the output image defined as: 'client' / 'server' / 'minimal' / 'all' <ul style="list-style-type: none"> • <i>User property</i> : winter.image.vm

Parameter details

<addModules>

The modules to add to the resulting image.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.addModules

<addOptions>

The options to prepend before any other options when invoking the JVM in the resulting image.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.addOptions

<attach>

Attach the resulting image archives to the project to install them in the local Maven repository and deploy them to remote repositories.

- **Type:** boolean
- **Required:** yes
- **User property:** winter.image.attach
- **Default:** true

<bindServices>

Link in service provider modules and their dependencies.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.bindServices
- **Default:** false

<compress>

The compress level of the resulting image: 0=No compression, 1=constant string sharing, 2=ZIP.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.compress

<configurationDirectory>

A directory containing user-editable configuration files that will be copied to the resulting image.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.configurationDirectory
- **Default:** \${project.basedir}/src/main/conf/

<excludeArtifactIds>

Comma separated list of Artifact names to exclude.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.excludeArtifactIds

<excludeClassifiers>

Comma Separated list of Classifiers to exclude. Empty String indicates don't exclude anything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** excludeClassifiers

<excludeGroupIds>

Comma separated list of GroupId Names to exclude.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.excludeGroupIds

<excludeScope>

Scope to exclude. An Empty string indicates no scopes (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.excludeScope

<formats>

A list of archive formats to generate (eg. zip, tar.gz...)

- **Type:** java.util.Set
- **Required:** yes
- **Default:** zip

<ignoreSigningInformation>

Suppress a fatal error when signed modular JARs are linked in the image.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.ignoreSigningInformation
- **Default:** false

<includeArtifactIds>

Comma separated list of Artifact names to include. Empty String indicates include everything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeArtifactIds

<includeClassifiers>

Comma Separated list of Classifiers to include. Empty String indicates include everything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeClassifiers

<includeGroupIds>

Comma separated list of GroupIds to include. Empty String indicates include everything (default).

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeGroupIds

<includeScope>

Scope to include. An Empty string indicates all scopes (default). The scopes being interpreted are the scopes as Maven sees them, not as specified in the pom. In summary:

- runtime scope gives runtime and compile dependencies,
- compile scope gives compile, provided, and system dependencies,
- test (default) scope gives all dependencies,
- provided scope just gives provided dependencies,
- system scope just gives system dependencies.
- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.includeScope

<jmodsOverrideDirectory>

A directory containing module descriptors to use to modularize unnamed dependency modules and which override the ones that are otherwise generated.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.jmodsOverrideDirectory
- **Default:** \${project.basedir}/src/jmods/

<launchers>

A list of launchers to include in the resulting runtime.

- **Type:** java.util.List
- **Required:** no

<legalDirectory>

A directory containing legal notices that will be copied to the resulting image.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.legalDirectory
- **Default:** \${project.basedir}/src/main/legal/

<manDirectory>

A directory containing man pages that will be copied to the resulting image.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.image.manDirectory
- **Default:** \${project.basedir}/src/main/man/

<overwriteNewer>

Overwrite dependencies that don't exist or are older than the source.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.overwriteNewer
- **Default:** true

<projectMainClass>

The main class in the project module to use when building the project JMOD package.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.runtime.projectMainClass

<resolveProjectMainClass>

Resolve the project main class when not specified explicitly.

- **Type:** boolean
- **Required:** no
- **User property:** winter.runtime.projectMainClass
- **Default:** false

<skip>

Skips the generation of the runtime.

- **Type:** boolean
- **Required:** no
- **User property:** winter.runtime.skip

<stripDebug>

Strip debug information from the resulting image.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.stripDebug
- **Default:** true

<stripNativeCommands>

Strip native command (eg. java...) from the resulting image.

- **Type:** boolean
- **Required:** no
- **User property:** winter.image.stripNativeCommands
- **Default:** true

<verbose>

Enables verbose logging.

- **Type:** boolean
- **Required:** no
- **User property:** winter.verbose
- **Default:** false

<vm>

Select the HotSpot VM in the output image defined as: 'client' / 'server' / 'minimal' / 'all'

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.image.vm

winter:help

Full name:

io.winterframework.tool:winter-maven-plugin:1.0.0:help

Description:

Display help information on winter-maven-plugin. Call `mvn winter:help -Ddetail=true -Dgoal=<goal-name>` to display parameter details.

Attributes:

Optional parameters

Name	Type	Description
detail	boolean	If true, display all settable properties for each goal. <ul style="list-style-type: none">• <i>User property</i> : detail• <i>Default</i> : false
goal	String	The name of the goal for which to show help. If unspecified, all goals will be displayed. <ul style="list-style-type: none">• <i>User property</i> : goal
indentSize	int	The number of spaces per indentation level, should be positive. <ul style="list-style-type: none">• <i>User property</i> : indentSize• <i>Default</i> : 2
lineLength	int	The maximum length of a display line, should be positive. <ul style="list-style-type: none">• <i>User property</i> : lineLength• <i>Default</i> : 80

Parameter details

<detail>

If true, display all settable properties for each goal.

- **Type:** boolean
- **Required:** no
- **User property:** detail
- **Default:** false

<goal>

The name of the goal for which to show help. If unspecified, all goals will be displayed.

- **Type:** java.lang.String
- **Required:** no
- **User property:** goal

<indentSize>

The number of spaces per indentation level, should be positive.

- **Type:** int
- **Required:** no
- **User property:** indentSize
- **Default:** 2

<lineLength>

The maximum length of a display line, should be positive.

- **Type:** int
- **Required:** no
- **User property:** lineLength
- **Default:** 80

winter:run

Full name:

io.winterframework.tool:winter-maven-plugin:1.0.0:run

Description:

Runs the project application.

Attributes:

- Requires a Maven project to be executed.
- Requires dependency resolution of artifacts in scope: compile+runtime.
- Requires dependency collection of artifacts in scope: compile+runtime.
- Since version: 1.0.
- Binds by default to the lifecycle phase: validate.

Optional parameters

Name	Type	Description
addUnnamedModules	boolean	Adds the unnamed modules when executing the application. <ul style="list-style-type: none"> • <i>User property</i> : winter.exec.addUnnamedModules • <i>Default</i> : true
arguments	String	The arguments to pass to the application.
commandLineArguments	String	The command line arguments to pass to the application. This parameter overrides AbstractExecMojo.arguments when specified. <ul style="list-style-type: none"> • <i>User property</i> : winter.run.arguments
configurationDirectory	File	A directory containing user-editable configuration files that will be copied to the image to execute. <ul style="list-style-type: none"> • <i>User property</i> : winter.exec.configurationDirectory • <i>Default</i> : \${project.basedir}/src/main/conf/
jmodsOverrideDirectory	File	A directory containing module descriptors to use to modularize unnamed dependency modules and which override the ones that are otherwise generated. <ul style="list-style-type: none"> • <i>User property</i> : winter.exec.jmodsOverrideDirectory • <i>Default</i> : \${project.basedir}/src/jmods/
mainClass	String	The main class to use to run the application. If not specified, a main class is automatically selected. <ul style="list-style-type: none"> • <i>User property</i> : winter.exec.mainClass
overWritelfNewer	boolean	Overwrites dependencies that don't exist or are older than the source. <ul style="list-style-type: none"> • <i>User property</i> : winter.exec.overWritelfNewer • <i>Default</i> : true
skip	boolean	Skips the execution. <ul style="list-style-type: none"> • <i>User property</i> : winter.exec.skip
verbose	boolean	Enables verbose logging. <ul style="list-style-type: none"> • <i>User property</i> : winter.verbose • <i>Default</i> : false
vmOptions	String	The VM options to use when executing the application. <ul style="list-style-type: none"> • <i>User property</i> : winter.exec.vmOptions • <i>Default</i> : -Dorg.apache.logging.log4j.simplelog.level=INFO -Dorg.apache.logging.log4j.level=INFO
workingDirectory	File	The working directory of the application. <ul style="list-style-type: none"> • <i>User property</i> : winter.run.workingDirectory • <i>Default</i> : \${project.build.directory}/maven-winter/working

Parameter details

<addUnnamedModules>

Adds the unnamed modules when executing the application.

- **Type:** boolean
- **Required:** no
- **User property:** winter.exec.addUnnamedModules
- **Default:** true

<arguments>

The arguments to pass to the application.

- **Type:** java.lang.String
- **Required:** no

<commandLineArguments>

The command line arguments to pass to the application. This parameter overrides AbstractExecMojo.arguments when specified.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.run.arguments

<configurationDirectory>

A directory containing user-editable configuration files that will be copied to the image to execute.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.exec.configurationDirectory
- **Default:** \${project.basedir}/src/main/conf/

<jmodsOverrideDirectory>

A directory containing module descriptors to use to modularize unnamed dependency modules and which override the ones that are otherwise generated.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.exec.jmodsOverrideDirectory
- **Default:** \${project.basedir}/src/jmods/

<mainClass>

The main class to use to run the application. If not specified, a main class is automatically selected.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.exec.mainClass

<overwriteNewer>

Overwrites dependencies that don't exist or are older than the source.

- **Type:** boolean
- **Required:** no
- **User property:** winter.exec.overWriteNewer
- **Default:** true

<skip>

Skips the execution.

- **Type:** boolean
- **Required:** no
- **User property:** winter.exec.skip

<verbose>

Enables verbose logging.

- **Type:** boolean
- **Required:** no
- **User property:** winter.verbose
- **Default:** false

<vmOptions>

The VM options to use when executing the application.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.exec.vmOptions
- **Default:** -Dorg.apache.logging.log4j.simplelog.level=INFO -Dorg.apache.logging.log4j.level=INFO

<workingDirectory>

The working directory of the application.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.run.workingDirectory
- **Default:** \${project.build.directory}/maven-winter/working

winter:start

Full name:

io.winterframework.tool:winter-maven-plugin:1.0.0:start

Description:

Starts the project application without blocking the Maven build.

This goal is used together with the stop goal in the pre-integration-test and post-integration-test phases to run integration tests.

Attributes:

- Requires a Maven project to be executed.
- Requires dependency resolution of artifacts in scope: compile+runtime.
- Requires dependency collection of artifacts in scope: compile+runtime.
- Since version: 1.0.
- Binds by default to the lifecycle phase: pre-integration-test.

Optional parameters

Name	Type	Description
addUnnamedModules	boolean	Adds the unnamed modules when executing the application. <ul style="list-style-type: none"> • <i>User property</i> : winter.exec.addUnnamedModules • <i>Default</i> : true
arguments	String	The arguments to pass to the application.
configurationDirectory	File	A directory containing user-editable configuration files that will be copied to the image to execute. <ul style="list-style-type: none"> • <i>User property</i> : winter.exec.configurationDirectory • <i>Default</i> : \${project.basedir}/src/main/conf/
jmodsOverrideDirectory	File	A directory containing module descriptors to use to modularize unnamed dependency modules and which override the ones that are otherwise generated. <ul style="list-style-type: none"> • <i>User property</i> : winter.exec.jmodsOverrideDirectory • <i>Default</i> : \${project.basedir}/src/jmods/
mainClass	String	The main class to use to run the application. If not specified, a main class is automatically selected. <ul style="list-style-type: none"> • <i>User property</i> : winter.exec.mainClass
overWriteIfNewer	boolean	Overwrites dependencies that don't exist or are older than the source. <ul style="list-style-type: none"> • <i>User property</i> : winter.exec.overWriteIfNewer • <i>Default</i> : true
skip	boolean	Skips the execution. <ul style="list-style-type: none"> • <i>User property</i> : winter.exec.skip
timeout	long	The amount of time in milliseconds to wait for the application to start. <ul style="list-style-type: none"> • <i>User property</i> : winter.start.timeout • <i>Default</i> : 60000
verbose	boolean	Enables verbose logging. <ul style="list-style-type: none"> • <i>User property</i> : winter.verbose • <i>Default</i> : false
vmOptions	String	The VM options to use when executing the application. <ul style="list-style-type: none"> • <i>User property</i> : winter.exec.vmOptions • <i>Default</i> : -Dorg.apache.logging.log4j.simplelog.level=INFO -Dorg.apache.logging.log4j.level=INFO
workingDirectory	File	The working directory of the application. <ul style="list-style-type: none"> • <i>User property</i> : winter.run.workingDirectory • <i>Default</i> : \${project.build.directory}/maven-winter/working

Parameter details

<addUnnamedModules>

Adds the unnamed modules when executing the application.

- **Type:** boolean
- **Required:** no
- **User property:** winter.exec.addUnnamedModules
- **Default:** true

<arguments>

The arguments to pass to the application.

- **Type:** java.lang.String
- **Required:** no

<configurationDirectory>

A directory containing user-editable configuration files that will be copied to the image to execute.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.exec.configurationDirectory
- **Default:** \${project.basedir}/src/main/conf/

<jmodsOverrideDirectory>

A directory containing module descriptors to use to modularize unnamed dependency modules and which override the ones that are otherwise generated.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.exec.jmodsOverrideDirectory
- **Default:** \${project.basedir}/src/jmods/

<mainClass>

The main class to use to run the application. If not specified, a main class is automatically selected.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.exec.mainClass

<overWriteIfNewer>

Overwrites dependencies that don't exist or are older than the source.

- **Type:** boolean
- **Required:** no
- **User property:** winter.exec.overWriteIfNewer
- **Default:** true

<skip>

Skips the execution.

- **Type:** boolean
- **Required:** no
- **User property:** winter.exec.skip

<timeout>

The amount of time in milliseconds to wait for the application to start.

- **Type:** long
- **Required:** no
- **User property:** winter.start.timeout
- **Default:** 60000

<verbose>

Enables verbose logging.

- **Type:** boolean
- **Required:** no
- **User property:** winter.verbose
- **Default:** false

<vmOptions>

The VM options to use when executing the application.

- **Type:** java.lang.String
- **Required:** no
- **User property:** winter.exec.vmOptions
- **Default:** -Dorg.apache.logging.log4j.simplelog.level=INFO -Dorg.apache.logging.log4j.level=INFO

<workingDirectory>

The working directory of the application.

- **Type:** java.io.File
- **Required:** no
- **User property:** winter.run.workingDirectory
- **Default:** \${project.build.directory}/maven-winter/working

winter:stop

Full name:

io.winterframework.tool:winter-maven-plugin:1.0.0:stop

Description:

Stops the project application that has been previously started using the start goal.

This goal is used together with the start goal in the pre-integration-test and post-integration-test phases to run integration tests.

Attributes:

- Requires a Maven project to be executed.
- Since version: 1.0.
- Binds by default to the lifecycle phase: post-integration-test.

Optional parameters

Name	Type	Description
skip	boolean	Skips the execution. <ul style="list-style-type: none">• <i>User property</i> : winter.stop.skip
timeout	long	The amount of time in milliseconds to wait for the application to stop. <ul style="list-style-type: none">• <i>User property</i> : winter.stop.timeout• <i>Default</i> : 60000
verbose	boolean	Enables verbose logging. <ul style="list-style-type: none">• <i>User property</i> : winter.verbose• <i>Default</i> : false

Parameter details

<skip>

Skips the execution.

- **Type:** boolean
- **Required:** no
- **User property:** winter.stop.skip

<timeout>

The amount of time in milliseconds to wait for the application to stop.

- **Type:** long
- **Required:** no
- **User property:** winter.stop.timeout
- **Default:** 60000

<verbose>

Enables verbose logging.

- **Type:** boolean
- **Required:** no
- **User property:** winter.verbose
- **Default:** false



Winter OSS Parent

The Winter OSS parent POM provides OSS dependencies and plugin management to Winter components and applications.

Dependencies

GroupId	ArtifactId	Version
com.fasterxml.jackson.core	jackson-databind	2.12.3
com.google.cloud.tools	jib-core	0.18.0
io.lettuce	lettuce-core	6.1.1.RELEASE
io.netty	netty-all	4.1.63.Final
io.netty	netty-buffer	4.1.63.Final
io.netty	netty-codec-http2	4.1.63.Final
io.netty	netty-common	4.1.63.Final
io.netty	netty-resolver	4.1.63.Final
io.netty	netty-tcnative-boringssl-static	2.0.39.Final
io.netty	netty-transport	4.1.63.Final
io.netty	netty-transport-native-epoll	4.1.63.Final
io.netty	netty-transport-native-epoll	4.1.63.Final
io.netty	netty-transport-native-epoll	4.1.63.Final
io.netty	netty-transport-native-epoll	4.1.63.Final
io.netty	netty-transport-native-kqueue	4.1.63.Final
io.netty	netty-transport-native-kqueue	4.1.63.Final
io.projectreactor	reactor-core	3.4.6
net.java.dev.javacc	javacc	7.0.10
org.apache.commons	commons-compress	1.20
org.apache.commons	commons-lang3	3.12.0
org.apache.commons	commons-text	1.9
org.apache.logging.log4j	log4j-api	2.14.1
org.apache.logging.log4j	log4j-core	2.14.1
org.apache.maven	maven-artifact	\${maven.version}
org.apache.maven	maven-compat	\${maven.version}
org.apache.maven	maven-core	\${maven.version}
org.apache.maven	maven-plugin-api	\${maven.version}
org.apache.maven.plugin-tools	maven-plugin-annotations	3.6.0
org.apache.maven.shared	maven-common-artifact-filters	3.1.0

org.junit.jupiter	junit-jupiter-api	5.7.1
org.junit.jupiter	junit-jupiter-engine	5.7.1
org.junit.jupiter	junit-jupiter-params	5.7.1
org.junit.platform	junit-platform-commons	1.7.1
org.junit.platform	junit-platform-launcher	1.7.1
org.mockito	mockito-core	3.8.0
org.ow2.asm	asm	9.1
org.webjars	swagger-ui	3.48.0

Maven Plugins

GroupId	ArtifactId	Version
org.apache.maven.plugins	maven-antrun-plugin	3.0.0
org.apache.maven.plugins	maven-clean-plugin	3.1.0
org.apache.maven.plugins	maven-compiler-plugin	3.8.1
org.apache.maven.plugins	maven-dependency-plugin	3.1.2
org.apache.maven.plugins	maven-deploy-plugin	2.8.2
org.apache.maven.plugins	maven-gpg-plugin	3.0.1
org.apache.maven.plugins	maven-install-plugin	2.5.2
org.apache.maven.plugins	maven-jar-plugin	3.2.0
org.apache.maven.plugins	maven-javadoc-plugin	3.2.0
org.apache.maven.plugins	maven-plugin-plugin	3.6.1
org.apache.maven.plugins	maven-resources-plugin	3.2.0
org.apache.maven.plugins	maven-source-plugin	3.2.1
org.apache.maven.plugins	maven-surefire-plugin	2.22.2
org.codehaus.mojo	exec-maven-plugin	3.0.0
org.javacc.plugin	javacc-maven-plugin	3.0.3
org.sonatype.plugins	nexus-staging-maven-plugin	1.6.8

The Winter Framework is released under version 2.0 of the [Apache License](#).

Copyright © 2021, The Winter Framework