# Runtime

# Table of Contents

A Jakarta EE 10 Core profile runtime.

## General

The product has several ways of operation

- standalone; Start up the runtime with one or more applications.
- domain; Start up the runtime and remotely administer the environment.
- clustered; Remote administration of several instances all running the same applications(s)
- embedded; Start the runtime as part of your application or test.

Applications are internally identified by the contextroot they are available (must be unique on the runtime) and when performing remote administration, applications are indicated by their deployment name.

## Known issues

- In order to have CDI injection in a JAX-RS resource to be working, the JAX-RS resource requires a CDI scope annotation.

## Installation

Unzip the *atbash-runtime.zip* file to a location of your choice. The archive contains an executable JAR file in the root and the dependencies in the *lib* folder.

From within the root folder, execute the command

```
java -jar atbash_runtime.jar <options>
```

(It is not possible to create a shaded executable jar file since that would make the entire code base a Bean Defining Archive in terms of CDI and will results in errors)

## Modules

### Core module

**Reacts on : Configuration changes**

Exposes some crucial data and services

- RunData: All essential data of the current run of the Runtime
- WatcherService : A service to perform some logging (with JFR events if activated) and storing some POJO that might be exposed through JMX (if activated)

## Config module

**Dependent on : Core Module**

Responsible for reading the configuration file, create them if needed, including the profile.

It exposes the following objects

- RuntimeConfiguration : The entire configuration of the current runtime instance
- PersistedDeployments : The Deployment information on the archives that are stored during the previous run.

## Logging Module

**Dependent on : Config Module**

Based on the configuration information, it starts the logging. It also ends the 'temporary logging'.

It does not expose any Objects.

## Jetty module

**Dependent on : Config Module**

It starts up the Jetty server based on the configuration information. When there is no `health` module active, it makes sure there is a basic handler for */health* so that system that inquiry if the runtime is healthy, behave properly.

It exposes

- HandlerCollection : So that other modules can add their own support for specifications or endpoints.

The module is also responsible for deploying applications that only use HTML or Servlet resources.

## Jersey module

**Dependent on : Jetty Module**

The module is responsible for deploying applications that use JAX-RS resources.

# Logging

The Runtime controls the entire logging subsystem.

The configuration is performed through a properties file that configures the Java Util Logging system. The file is expected at `<instance>/logging.properties` and is created from a default file when not found.

Within the Runtime itself, the logging is performed through SLF4J API. A bridge to Java Util Logging is included. The `be.atbash.runtime.logging.handler.LogFileHandler` handles the logging to file.

By default, only some important messages are written to the console and if needed, the `--logToConsole` is required to view logging entries also on the console.

## Log format

Atbash Runtim comes included with 4 different log formats. The default one is the SimpleLogFormat but the formatter can be changed by defining the fully qualified class name within the *logging.properties* file.

```
be.atbash.runtime.logging.handler.LogFileHandler.formatter=be.atbash.runtime.logging.h
andler.formatter.SimpleLogFormatter
```

### SimpleLogFormatter

The SimpleLogFormatter shows the basic information of the log records like this

```
Jun 11, 2022 22:33:45 be.atbash.runtime.MainRunnerHelper#logStartupTime INFO: CLI-103:
Started Atbash Runtime in 0.882 secs
```

which corresponds to the following String format that is used by the logger.

```
%1$tb %1$td, %1$tY %1$tT %2$s %4$s: %5$s%6$s%n
```

parameter 1 → Timestamp of the log entry parameter 2 → Name of the logger or the Class#Method name where the log entry is created. parameter 3 → Name of the logger (not shown in the default format) parameter 4 → Name of the log Level parameter 5 → Code of the log message parameter 6 → log message, includes the stacktrace in case the log entry is associated with a Throwable.

When you or the system, is making use of the Mapped Diagnostic Context (MDC) the entries are shown just before the actual message and can be configured by referencing parameter 7.

You can customise the output of the log format by the Simple Log formatter by defining the desired layout through the *log.properties* file content

```
be.atbash.runtime.logging.handler.formatter.SimpleLogFormatter.format=%1$tb %1$td,
%1$tY %1$tT %2$s %4$s: %5$s%6$s%n
be.atbash.runtime.logging.handler.formatter.SimpleLogFormatter.format.mdc=%1$tb %1$td,
%1$tY %1$tT %2$s %4$s: [%7$s]%5$s%6$s%n
```

### JSONLogger

Atbash Runtime has a JSON Log formatter included that can be selected by defining its fully qualified class name in the *logging.properties* file for the formatter key.

```
be.atbash.runtime.logging.handler.LogFileHandler.formatter=be.atbash.runtime.logging.h
andler.formatter.JSONLogFormatter
```

The generated log entries looks like

```
{"Level":"INFO","LevelValue":"800","LogMessage":"CLI-103: Started Atbash Runtime in
1.152 secs","LoggerName":"be.atbash.runtime.RuntimeMain","MessageID":"CLI-
103","ThreadID":"1","ThreadName":"main","TimeMillis":"1655060565219","Timestamp":"2022
-06-12T21:02:45.219+0200"}
```

By default, following information is included

- The Log Level name and value
- The Log message
- The Message Id
- The Thread id and name where the log entry is generated
- The time information (as millis since 1 JAn 1970 and as String according to RFC3339)
- When an Exception is associated with the entry, an additional `Throwable` property key that is a Json Object structure containing the Exception message and the StackTrace
- Additional Json properties can be included for the MDC data.

When the Log Level is FINE or lower, the class and method name are included in the output.

Some properties can be excluded by defining it as an exclude field.

```
be.atbash.runtime.logging.handler.LogFileHandler.excludeFields=x,y,z
```

As indicated in the example, the keys needs to be separated by a `,` and the following keys are supported

- `tid`: no Thread Id and Thread name in the log entry
- `timeMillis` The field *TimeMillis* is omitted in the output.
- `levelValue` The field *LevelValue* is omitted in the output.

## ODLLogger

Atbash Runtime has the ODL (Oracle Diagnostic Logger) formatter included based on the Glassfish and Payara products. It can be selected by defining its fully qualified class name in the *logging.properties* file for the formatter key.

```
be.atbash.runtime.logging.handler.LogFileHandler.formatter=be.atbash.runtime.logging.h
andler.formatter.ODLLogFormatter
```

The generated log entries looks like

```
[2022-06-13T18:39:34.818+0200] [INFO] [] [be.atbash.runtime.RuntimeMain] [tid:
_ThreadID=1 _ThreadName=main] [timeMillis: 1655138374818] [levelValue: 800] CLI-103:
Started Atbash Runtime in 0.998 secs
```

Fields are encapsulated in `[]` and the optional fields are structured like `[<key>: <value>]` including the MDC values.

By default, following information is included

- The Log Level name and value
- The Log message
- The Thread id and name where the log entry is generated
- The time information (as millis since 1 Jan 1970 and as String according to RFC3339)
- When an Exception is associated with the entry, the exception message and the Stacktrace is part of the log message.

When the Log Level is FINE or lower, the class and method name are included in the output (within the single field containing several log field entries).

Some values can be excluded by defining it as an exclude field.

```
be.atbash.runtime.logging.handler.LogFileHandler.excludeFields=x,y,z
```

As indicated in the example, the keys needs to be separated by a `,` and the following keys are supported

- `tid`: no Thread Id and Thread name in the log entry
- `timeMillis` The field *TimeMillis* is omitted in the output.
- `levelValue` The field *LevelValue* is omitted in the output.

The ODL format also supports colored output which is only useful for console output as the log entries otherwise contain some 'special' characters.

The activation can be done by having the following entry within *logging.properties* file

```
be.atbash.runtime.logging.handler.formatter.ODLLogFormatter.ansiColor=true
```

## Uniform Logger

Atbash Runtime has a Uniform Log formatter included based on the Glassfish and Payara products. It can be selected by defining its fully qualified class name in the *logging.properties* file for the formatter key.

```
be.atbash.runtime.logging.handler.LogFileHandler.formatter=be.atbash.runtime.logging.h
andler.formatter.UniformLogFormatter
```

The generated log entries looks like

```
[#|2022-06-
13T13:20:50.690+0200|INFO|be.atbash.runtime.RuntimeMain|_ThreadID=1;_ThreadName=main;_
TimeMillis=1655119250690;_LevelValue=800;|CLI-103: Started Atbash Runtime in 0.984
secs|#]
```

Fields are seperated by `|` (but this can be configured) and there is one single field containing the optional fields and the MDC values.

By default, following information is included

- The Log Level name and value
- The Log message
- The Thread id and name where the log entry is generated
- The time information (as millis since 1 Jan 1970 and as String according to RFC3339)
- When an Exception is associated with the entry, the exception message and the Stacktrace is part of the log message.

When the Log Level is FINE or lower, the class and method name are included in the output (within the single field containing several log field entries).

Some values can be excluded by defining it as an exclude field.

```
be.atbash.runtime.logging.handler.LogFileHandler.excludeFields=x,y,z
```

As indicated in the example, the keys needs to be separated by a `,` and the following keys are supported

- `tid`: no Thread Id and Thread name in the log entry
- `timeMillis` The field *TimeMillis* is omitted in the output.
- `levelValue` The field *LevelValue* is omitted in the output.

The UniformLogger also supports colored output which is only useful for console output as the log entries otherwise contain some 'special' characters.

The activation can be done by having the following entry within *logging.properties* file

```
be.atbash.runtime.logging.handler.formatter.UniformLogFormatter.ansiColor=true
```

## Features

When the runtime starts, a rotation of the log file happens (if the instance has already a log file from the previous run).

The rotation of the log file is recommended (to avoid log files that become too large and can't be opened anymore)

There are several options for rotation.

1. Size based: This is the default that is active.When the file becomes larger that the configured size (2 MB by default), a rotation happens.
2. Daily: A rotation can be requested at the start of the day (based on the clock of the server running the process).
3. Time based: A rotation every x minutes can be requested.

The size based option is combined with the time based options.So unless you disable the size based rotation, you might so rotation happening of the log file in between the time based one.

Time based rotation (including the daily one) has a granularity of 1 minute.So the rotation can happen at maximum 1 minute before or after the intended moment.

Daily rotation has precedence over the time based rotation.

## SLF4J message format and Resource Bundle

The Atbash Runtime includes a special SLF4J to Java util Logging bridge.This bridge allows you to use Resource Bundles in combination with SLF4J.For a SLF4J logger, the corresponding ResourceBundle is looked up according to this mapping.

`LoggerFactory.getLogger(Foo.class);` → ResourceBundle is `msg.the.package.to.FOO.properties`.

When the message passed to the SLF4J logger contains the `{}` placeholder, the message formatting is performed according the SLF4J rules. Otherwise the Java Util Logging formatter is responsible for creating the final message.

More functionality in the next phase of the Logger module rewrite.

# Logging - Technical

## Early logging

Since the logging module requires the Configuration module to be started (so that configuration is known) the logging cannot be configured immediately after the Java program starts. The initial logging is captured by an `EarlyLogHandler` that stores the LogRecords within memory until the logging module is ready.

The Early Logging is configured through this call:

```
LoggingManager.getInstance().initializeEarlyLogging(logToConsole);
```

Performed steps are

- Define *RuntimeLogManager* as Log manager.
- Set logToConsole value as System property (`LoggingUtil.SYSTEM_PROPERTY_LOGGING_CONSOLE = "runtime.logging.console"`)
- Remove all handlers from the Root Logger.
- Add the Early Logging Handler.

When the Logging module has configured the Logging system (defined the logging properties), it *terminates* the early logging. All messages stored in memory are written out to the loggers so they will appear in the logging file.

## Important console logging

When specifying `--logToConsole` all logging also appears on the console when starting the Runtime. Without this option, only a few important messages appears.

Any part of the code can write to the console by getting a special logger like this.

```
LoggingManager.getInstance().getMainLogger(xxx.class);
```

This will return a Logger that is properly setup forthe Console according the logToConsole option.

# Logging Configuration

Following option are valid when using the logging to file.

## Log File location

This can be specified using the `file` option. This should just be the file name of the log file. If using a relative location or absolute path, the log file will not be located within the configuration directory.

## Rotation at End of Day

By setting the option `rotationOnDateChange` to *true*, the log file is rotated at the end of the day. The rotation based on number of minutes is ignored in this case but it can be combined with the rotation based on size.

## Rotation every n minutes

A rotation every *n* minutes can be achieved by setting the option `rotationTimelimitInMinutes=n`. This option is ignored when a rotation at end of day is requested but can be combined with rotation of the log file based on his size.

## Rotation based on file size

A rotation based on the size of the log file can be achieved by setting the required size in the `rotationLimitInBytes` option. A valid smaller than 500000 (bytes) is considered as too small and rejected. A valid of 0 disables the rotation based on file size.

## Maximum number of history files

To reduce the number of old server log files that are kept in the directory, set the maximum number through the `maxHistoryFiles` option. The default value is *10* and this value is also used when an invalid value is defined for the option (like a negative number or not a number at all). When you specify the value *0*, no file is deleted and the user is responsible himself to delete files as otherwise the disk might get full of log files.

This maximum number does not work correctly when the log file location is changed at runtime when.

## Compression of rotated file

After the log file is rotated, a compression can be performed to reduce the amount of disk space it takes. Set the `compressOnRotation` option to true to perform a GZIP compression of the file.

# Command line option

## Runtime

java -jar atbash-runtime.jar <options>

**-r|--root**: Location of 'domain' directory, by default current directory.

**-n|--configname**: Configuration name within the domain, by default `default`.

**-v|--verbose**: When active, more messages are shown in the log. This can be enforced 'permanent' by putting `be.atbash.runtime.level=ALL` in the logging configuration.

**--watcher**: Defines the type of the diagnostics, valid values are
`MINIMAL`: (default) only the events from the Core module are sent to JFR.
`OFF`: JFR and JMX are disabled
`JFR`: Events are send to JFR system and Recording is started which is dump to file at JVM exit.
`JMX`: Some data is available within the JMX system.
`ALL`: Combination of `JFR` and `JMX`

**-p|--profile**: The name of the active profile. The Profile defines a set of modules that are started. When not specified, the profile id `default`. Also `domain` is supported.

**-m|--modules**: Change the modules that are active. the option is a comma separated ist list of *module definitions*.
A module definition consist of the module name that is optionally prepended by `+` or `-` or no action indicator.
The final list of modules that are used is determined by the profile and then changes are done as following

No action, the modules defined in the profile are replaced by these modules.
`+` The module is added to the current list of modules
`-` The module is removed from the current list of modules

The mandatory modules *Config* and *Logging* are always added to the list.

The `--modules` option is an expert setting and using it incorrectly can result in a failure to run your application.

**--port**: The main HTTP port for accepting requests. By default 8080.

**--stateless**: No configuration information is stored on disk for future runs. Ideal for creating stateless containers in combination with `--no-logToFile`.

**--logToConsole**: Send all logging to the console. This overrules the value in the logging configuration file (TBC)

**--no-logToFile**: Does not send logging to the file. This overrules the value in the logging configuration file (TBC)

zero, one or more WAR files can be added to the command line that needs to be deployed. Also the applications that are already 'deployed' within the configuration are started.

**--datafile** Defines the configuration data properties file with key values pairs for the application(s) deployment data. These deployment data can influence the configuration of the modules. For an overview, see ??? ( FIXME create such a section)

**-c|--configfile** Defines the configuration file with commands that needs to be executed. after start up of the runtime but before the applications are deployed. The file can contain comments when the line starts with `#`. Otherwise the line need to contain a valid command as specified in the next section *CLI*

The exit status of the process is

- 0: normal exit
- -1: Incorrect commandline options specified
- -2: Runtime failed to start due to some error.

# CLI

java -jar atbash-cli.jar <command> <options>

### create-config

Create a configuration within the domain.

xx To be documented xx

### Options for all remote commands

-h | --host (default - localhost): Host name or IP address of the machine running the Atbash runtime in Domain mode.

-p | --port (default - 8080): Port number assigned the process running the Atbash runtime in Domain mode

-f | --format (default - TEXT): Format output of the Remote CLI commands. Support values are TEXT and JSON.

### status (remote)

Returns the status and information of the runtime with as output.

version - version number of the runtime. modules - the active modules of the runtime. uptime - The time the runtime is already running.

# deploy (remote)

Deploys the application on the runtime, and optionally define the context root.

`deploy <file>`

`deploy --contextroot <root> <file>`

`deploy --contextroot <root1,root2> <file1> <file2>`

Altough deploying multiple applications is supported, it is not recommended as the outcome might not be clear in case one of the deployments fails.

# list-applications (remote)

List all applications running on the runtime.

## undeploy (remote)

Undeploy an application

`undeploy <name>`

## set (remote)

Set configuration parameters for modules. The parameters must be in the form of <module>.<key>=value. Multiple parameters can be specified on the same set command (space separated)

`set mp-config.validation.disable=true`

## set-logging-configuration (remote)

Sets the logging configuration parameters. The parameters must be in the form of <key>=<value>. Multiple parameters can be specified on the same command (space separated)

- set-logging-configuration file=<log-file>
- set-logging-configuration rotationOnDateChange=true|false
- set-logging-configuration rotationTimelimitInMinutes=0
- set-logging-configuration rotationLimitInBytes=
- set-logging-configuration maxHistoryFiles=
- set-logging-configuration compressOnRotation=

You can also specify, in addition to the above parameters, a new logging properties file that must be used (in stead of sending individual properties) If you combine a file and parameters, the parameters are applied *after* the file is used. There

- set-logging-configuration --file <logging.properties.file>

# MicroProfile Config

## Validation

For some CDI validation messages, the log contains more details why a certain dependency is not found.

For example

```
WELD-001408: Unsatisfied dependencies for type int with qualifiers @ConfigProperties
at injection point [BackedAnnotatedField] @Inject @ConfigProperties private
be.rubus.runtime.examples.mpconfig.properties.ConfigPropertiesResource.wrong
```

has the following clarification in the log.

```
[SEVERE] [MPCONFIG-012] [be.atbash.runtime.config.mp.ConfigExtension] [tid:
_ThreadID=1 _ThreadName=main] [timeMillis: 1643116914487] [levelValue: 1000] MPCONFIG-
012: Injection point with @ConfigProperties is not supported with a primitive or array
and found Type 'int' at
be.rubus.runtime.examples.mpconfig.properties.ConfigPropertiesResource.wrong
```

## Module disabled when no `microprofile-config.properties` file

The Microprofile configuration functionality is disabled when there is no `microprofile-config.properties` file detected in the Web Archive that is deployed. The CDI scanning of the WAR file is not performed for any Microprofile Config related artifacts. If they are present, the deployment might fail, or you may encounter issues during runtime.

If you want to have the Microprofile Config functionality available without having the `microprofile-config.properties` file (because you are using other configuration sources), you can perform the following command

```
java -jar atbash-cli.jar set mp-config.enabled.forced=true
```

or define the set command within the configuration file supplied during start up.

You can also use the properties file you specify ith --datafile command line parameter to activate the MicroProfile Config functionality for a certain deployment.

```
java -jar atbash-runtime.jar --datafile path/to/deploy.properties /path/to/app.war
```

with the content for the *deploy.properties* as

```
mp-config.enabled=true
```

## Disable validation

The MicroProfile Configuration specification requires that all injection points to be validated to make sure that all required configuration value are present.

There are cases that you fo not want that this validation happens. You want to deploy the application before the source with the Configuration values are accessible for ready.

You can bypass this validation by defining the Runtime setting `mp-config.validation.disable` with a value *true*. You can do this with the set command, using the `MainCLI` tool ir by defining the command in the configuration filr and specify it when start up the runtime.

```
java -jar atbash-cli.jar set mp-config.validation.disable=true
```

or

```
java -jar atbash-runtime.jar -c config.txt <path-to>/<application>.war
```

with the contents of *config.txt* equals to

```
set mp-config.validation.disable=true
```

## Additional Properties resources

By default, only `microprofile-config.properties` file within the *META-INF* directory is searched for configuration properties. Atbash Runtime has the possibility to define additional locations of properties *files* that can be added as ConfigSource.properties You can define the location of these additional locations by using the key `atbash.config.locations`. You can define this within the `microprofile-config.properties` file or as System Property or Environment Variable (it must be one of the 3 default Configuration sources)

The following types are supported.

- URl when value starts with *http:* or *https:*.
- Classpath when value starts with *classpath:*.
- Files when when value start with *file:* but this can be omitted.

Also the properties files that supplied with the `--datafile` command line property is used as MicroProfile Config source with a default priority of 200.

# MicroProfile JWT Auth

## Activation

This module is not active by default as it is not art of the Jakarta Core Profile. You can activate it in two ways

- -m +jwt-auth
- -p all

The first command line option activates the module on top of the defaults modules of the Core profile. The second option activates all modules known by Atbash Runtime, including the MicroProfile JWT Auth module.

Even when it is activated, an application can make use of it when at deployment time the

`@LoginConfig` annotation is found at the class level of any class. The class doesn't need to be defined as a CDI bean either.

This is less strict as the specification indicates.

## Validation

At deployment time, the following configuration values are checked and deployment fails if there is a problem.

- The MicroProfile Config based property key `mp.jwt.verify.issuer` must result in a non-blank value.
- When the MicroProfile Config based property key `mp.jwt.verify.publickey.location` is not defined, `mp.jwt.verify.publickey.location` must resolve to a location that exists. An URL is not checked during deployment to allow the deployment of the application before the other service providing the keys is accessible.

## Less strict implementation

Several aspects of the implementation are implemented in a less strict manner as they impose unnecessary constraints or restrict the JOSE specification so that JWT Auth is not usable in a microservice environment that includes components written in other languages or frameworks.

Some rules can be made strict as defined in the TCK by setting the JVM System property `atbash.runtime.tck.jwt`.

Some parameter's interpretation is changed so that the application to be functional in a multi-tenancy or multi-source environment. Since the goal is to be used in a microservices environment, it can be part of several systems operating under different tools providing different type of tokens.

- The MicroProfile Config based property key `mp.jwt.token.header` is used in a more loosely way. Unless the value is explicitly set to *Authorization*, the presence of a token within a Cookie is also accepted when no header found. Can be made restrictive by using the System property.
- The indication of a decryption key does not exclude the possibility to use also a Signed token. Valid tokens can either be signed or encrypted. Can be made restrictive by using the System property.
- The properties `mp.jwt.verify.publickey.location` and `mp.jwt.decrypt.key.location` are interpreted as a List (comma separated values) and keys are loaded from all the specified locations. This is part of the support for the multi-tenancy / multi-source support.
- The property `mp.jwt.verify.issuer` is also interpreted as a List (comma separated values) and the issuer found in the token must be present in this list. This is part of the support for the multi-tenancy / multi-source support.
- As encryption algorithm, RSA-OAEP is not supported as it uses the SHA-1 hashing which is indicated as insecure since 2017. RSA-OAEP might not be affected but as a precaution, only RSA-OAEP-256 is supported (containing the SHA 2 family hashing algorithm)

## Tracing

When the Deployment Data `jwt-auth.tracing.active` is set to true, additional logging is added to indicate why the authentication failed. It gives you detailed info about what part (signature validation failed, missing claims, ...) resulted in the authentication or authorization failure.

It also assigns a unique id to dach request and places that in the MDC context of SLF4J. So any log entry that you make within your application also has this unique identification of the request.

## Additional functionality

- The Claims that specify a date value (like *exp*, *iat*, etc) but also custom claims that are transmitted as a number but actually represent the number of seconds since 1 Jan 1970, can be injected as a Date.
- In contrast to JWT Auth Specification, a JAX-RS resource method that is not annotated with a permission annotation (@RolesAllowed or @permitAll), will receive the @DenyAll annotation as this is probably an oversight of the developer. These occurrences are reported in the log the first time they are called.

# MicroStream integration

## Activation

This module is not active by default as it is not part of the Jakarta Core Profile. You can activate it in two ways

- -m +microstream
- -p all

The first command line option activates the module on top of the defaults modules of the Core profile. The second option activates all modules known by Atbash Runtime, including the MicroStream integration module.

## 3 integration aspects

The `EmbeddedStorageManager` can be configured by defining configuration values through MicroProfile config properties keys. All the supported configuration values can be found on Configuration properties with the following conversion rules for the name

- The `_` is replaced by a dot `.`
- They keys are prefixed by `one.microstream.`

The root object for the Storage Manager needs to be annotated with `@Storage`. When using the recommended bean discovery mode *annotated* this means that also a bean defining annotation like `@ApplicationScoped` needs to be specified on this class. However, this scope is ignored and the bean is always made available with scope `@ApplicationScoped`. This annotation makes it possible to recognize the CDI bean that needs to be used as root object and will be used to initialize an empty storage.

The last integration relates to the indication when a part of the graph is modified and needs to be written by the MicroStream Storage Manager. This can be done by using the `DirtyMarker` in combination with the `@Store` CDI interceptor.

For example

```
@Inject
private DirtyMarker marker;

@Inject
private Root root;  // The class that is marked as storage root.

@Store
public void addUser(User user) {
   // Some additional code
   marker.mark(root.getUsers()).add(user);
   // some more code
}
```

We need to supply the Storage manager the object that has changed, for example the *ArrayList* where we added an item, the object where where we changed the description as a String. With the `DirtyMarker` this can be done by the developer.

At the end of the method, the CDI interceptor will call the `StorageManager.store(Object)` method with all the objects that are marked as dirty. There are 2 members of the annotation that can be used to tune the processing.

```
@Store(asynchronous = true, clearLazy = true)
```

Storing will happen by default in a separate thread, asynchronous. If you do not want this, you can specify the value false with a small performance penalty for your user request.

When you have marked a Lazy reference, it will be cleared by the interceptor unless you specify the value *false* for the member.

You do not need to use the `@Store` annotation. At the end of the request, all objects that are marked will be stored anyway, asynchronous and when it is a Lazy reference it will also be cleared.

## Customizer and Initializer

The *StorageManager* configuration and Root object can be customized and initialized.

When you define one or more CDI beans that implement the `EmbeddedStorageFoundationCustomizer` interface, the code within the `customize` method will be executed after the

*EmbeddedStorageFoundation* is created based on the values found in the configuration.

The initial data of the root object can be added when you define a CDI bean that implements `StorageManagerInitializer` interface. The `initialize` method is called just before the StorageManager object is handed over to the CDI container.

Depending on the fact if you have defined a class with `@Storage` , the root object will always be available (when you have used the annotation) or only when there are already data in the storage.

These are the steps that are performed

- Create the *EmbeddedStorageFoundation* based on the configuration values
- Execute the customizer, CDI beans having *EmbeddedStorageFoundationCustomizer* interface.
- Start the Storage Manager
- Execute the initializers, CDI beans having *StorageManagerInitializer* interface.

## Limitations

MicroStream Storage Manager cannot be used in a clustered environment pointing to the same data location. Additional coding effort is required (for example, have a look at communication documentation). In a future version of the integration, some put of the box solution might be made available.

# BOM

## Reasoning

Since there is no real artifact available, like the Web Profile API of Jakarta EE, that provide you with all your classes required during the compilation of your application, a specific artifact is created.

There are 2 types of artifacts created

- A BOM that allows you to include exactly those dependencies that you use in your application like, Servlet JAX-RS, JSON and MicroProfile Config for example.
- An artifact that includes all api classes that correspond with the Core profile. It contains all specifications that would have appeared in Jakarta EE 10.

## Using the BOM

To use specify each specification separately, use the following steps.

Within the `<dependencyManagement>` section of the *pom.xml*, define the following dependency.

```
            <dependency>
                <groupId>be.atbash.runtime.api</groupId>
                <artifactId>bom</artifactId>
                <version>${atbash.runtime.version}</version>
                <scope>import</scope>
                <type>pom</type>
            </dependency>
```

Each specification you need can then be defined within the `<dependencies>` section. For example;

```
        <dependency>
            <groupId>jakarta.ws.rs</groupId>
            <artifactId>jakarta.ws.rs-api</artifactId>
            <scope>provided</scope>
        </dependency>

        <dependency>
            <groupId>jakarta.enterprise</groupId>
            <artifactId>jakarta.enterprise.cdi-api</artifactId>
            <scope>provided</scope>
        </dependency>
```

## Using the core-api artifacts

The `core-api` artifact is similar to the Jakarta EE Web Profile artifact. A single JAR file contains all classes of all specifications within that profile.

By adding this dependency to your *pom.xml* file

```
            <dependency>
                <groupId>be.atbash.runtime.api</groupId>
                <artifactId>core-api</artifactId>
                <version>${atbash.runtime.version}</version>
                <scope>provided</scope>
            </dependency>
```

You can use the Servlet API, CDI api classes, JAX-RS, JSON-P, JSON-B, Jakarta Interceptor, Jakarta Annotations and the MicroProfile Config classes within your application. This corresponds to the specifications that are supported by the *default* profile of the Atbash runtime.

## Using the full-api artifacts

The `full-api` artifact is similar to the Jakarta EE Web Profile artifact. A single JAR file contains all classes of all specifications within that profile.

By adding this dependency to your *pom.xml* file

```
            <dependency>
                <groupId>be.atbash.runtime.api</groupId>
                <artifactId>full-api</artifactId>
                <version>${atbash.runtime.version}</version>
                <scope>provided</scope>
            </dependency>
```

On top of the specifications available within the *core-api*, you have the MicroProfile JWT Auth API, MicroStream integration classes, MicroStream v7 and the SLF4J API classes available for your application code.

# Packager

The main artifact, `atbash-runtime.zip` contains all modules and depending on the profile, some modules are active and other not. Although a module is not active, it is still on the classpath and might have an impact on the performance of the runtime.

The Runtime packager, build by the *packager* module, can create a new *executable* that contains only those modules that you have specified. This also means that when a certain module is not included, some functionality is not available on the runtime.

The packager generates a *pom.xml* that can be used to generate the runtime with the modules that you have specified.

## Usage

The packager can be used on the command line as follows:

```
java -jar atbash-packager.jar -r <path> -a <artifact> -m <modules>
```

-r/--root: Indicates the target directory where the maven project will be created. This directory should not exist.

-a/--artifact: Name of the custom packaging. It will be the name of the zip file that is created when you execute the generated project and also the name of the main executable jar file.

-m/--modules: A comma separated list of modules names that must be included in the packaging. The supported module names are

- jetty: Handles user requests and provides Servlet Specification
- jersey: Brings the JAX-RS support but also the JSON-P and JSON-B
- mp-config: Brings in the MicroProfile Config support.
- jwt-auth: Brings in the MicroProfile JWT Authentication support.
- microstream: Brings in the MicroStream support.

The modules for the core, configuration, logging, and CDI (Weld) are always included.

# Integrations

## Java Flight Recorder

Atbash Runtime has support for the Java Flight recorder functionality. When defining the option `--watcher JFR` or `--watcher ALL` for the instance/domain, a JFR Recording is started automatically and written out when the JVM exits. Several messages are also created as JFR events and allow the analysis of the JVM behaviour.

The JFR file is created in the current directory.

## Docker

A Docker image is available to run the Atbash Runtime containerized. It is based on JDK 11 and has several environment variables to configure its configuration. The Runtime runs with the stateless option, no files are created unless one configuration file in the temporary folder.

## Basic usage

```
FROM runtime-main:0.4
COPY myservice.war $DEPLOYMENT_DIR
```

The $DEPLOYMENT_DIR is a variable that is also passed to the start up command as the directory that is scanned for WAR applications that needs to be deployed.

## Environment variables

These are the environment variables that can be used to modify the configuration:

- **MEM_MAX_RAM_PERCENTAGE** (default "70.0"): The amount of the Java Heap within the container as a percentage of the total memory assigned to the container.
- **MEM_XSS** (default 512k) Value of the stack size.
- **JVM_ARGS**: Other JVM Arguments.
- **ATBASH_ARGS**: Options for the Atbash Runtime.
- **STATELESS**: Has by default the value *true* which results in the addition of the options `--stateless --no-logToFile` to the options passed to Atbash Runtime. When you define another value for the environment variable, you can have the non stateless behaviour.

This is the command that starts the instance.

```
exec java -XX:MaxRAMPercentage=${MEM_MAX_RAM_PERCENTAGE} -Xss${MEM_XSS}
-XX:+UseContainerSupport ${JVM_ARGS} -jar atbash-runtime.jar --logToConsole
--deploymentdirectory ${DEPLOYMENT_DIR} ${ATBASH_ARGS}
```

### Configuration file

When a configuration file is required to perform the setup of the runtime after booting up, copy the file with all the commands to the location `$CONFIG_FILE_LOCATION`.

```
COPY config.boot $CONFIG_FILE_LOCATION
```

### Examples

Run a domain mode Runtime instance

```
docker run -d --name atbash -p 8080:8080 -e ATBASH_ARGS='--profile domain' runtime-
main:0.3
```

Map a directory to the deployment directory so that it can run an application without assembling a specific Docker Image.

```
docker run -d --name atbash -p 8080:8080 -v "$PWD"/target:/opt/atbash/deployment
runtime-main:0.3
```

# Testing

## Runtime Testing Framework

The Testing framework, based on testContainers has the following goals.

1. Allow the developer to test out application on Atbash Runtime.
2. Allow advanced testing of the Atbash Runtime features itself.

The second goals allow us to automate as many as possible of the test cases of Atbash Runtime itself.

The framework is based on JUnit 5 and make use of the TestContainers framework.

## Setup

Add the following dependency to your project.

```
    <dependency>
        <groupId>be.atbash.runtime.testing</groupId>
        <artifactId>framework</artifactId>
        <version>${runtime.testing.framework.version}</version>
        <scope>test</scope>
    </dependency>
```

And create a class that extends from `be.atbash.runtime.testing.AbstractAtbashTest` and annotate it with `@AtbashContainerTest`.

```
@AtbashContainerTest
public class MyTestIT extends AbstractAtbashTest {
    // ...
}
```

The annotation makes sure the JUnit 5 extension is used and that a container running the Atbash Runtime is started. The parent class give you access to many useful methods to test out your application and interact with the container.

The application within the project itself will be added to the custom-created container. It is deployed as `test.war` and thus accessible under the context root `/test`. When it is a web application project, the war file will be build during package phase and available for testing (integration-test). When you run the test from within the IDE, you need to make sure the application is build before the test runs.

The abstract parent class give you some easy access to the root of the deployed application. `atbash` is the name of the variable holding the container and the method `getClientWebTargetApplication()` returns a Rest Client Target to the deployed application.

```
    @Test
    public void testEndpoint() {

        String result = getClientWebTargetApplication(atbash).path("/api/person")
.request().get(String.class);
```

## Custom image

Instead of using the official image of the Atbash Runtime, you can also define your own custom Docker script. Define the *Dockerfile* in the directory `src/docker/<name>` together with all the other resources that are needed for the creation of the image. The directory itself and all subdirectories will be available for the docker build image command.

The test class need to specify the name of the custom image in the `@AtbashContainerTest` annotation.

```
@AtbashContainerTest(value = "<name>")
```

So the name of the directory must be used as the content for *value* member.

Unless otherwise configured, see next section, the test application is also added to the image and deployed by default.

## Define start up options

You can define some startup options for the Atbash Runtime process through the `startupParameters` member of the `@AtbashContainerTest` annotation.

To start the Runtime within the Docker container in domain mode, use the following definition.

```
@AtbashContainerTest(startupParameters = "--profile domain")
```

## No Test Application

For some tests, you do not want that the framework adds the application as test application to the image. You can indicate this by specifying the `testApplication` member of the annotation.

```
@AtbashContainerTest(value = "<name>", testApplication = false)
```

This will also work with the standard image but is used most of the time in combination with a custom image.

## Version of the Docker Image

The version of the Atbash Runtime Test framework determines the version of the Docker image that will be used. In our example we used earlier, the value of `${runtime.testing.framework.version}` determine the version of the Docker image.

You can override this value by setting the Java System property `atbash.runtime.version` to the desired value.

If you are using a custom image that is derived from the official Atbash Runtime image, the version is also updated automatically.

In case the Dockerfile contains these lines

```
FROM runtime-main:0.4
...
```

And you are running version `0.5` of the Atbash Runtime Test framework (or you have specified the System property to that value), the Dockerfile content is rewritten on the fly to

```
FROM runtime-main:0.5
...
```

## Advanced definition of Docker Version

The previous section covered the basics about the version of the Docker Image that is used for the test run. The value of the *value* member of `@AtbashContainerTest` is interpreted as 3 parts that are separated by `-`.

```
<name>-<version>-<jdk>
```

The <*name*> is already explained in the previous section. If not specified or `default` is entered, the official Docker image is taken. Any other value defines the name of the directory that contains the definition of the Docker image.

The <*version*> determines the version of the docker image that is being used. If not specified, it is the version of the Test framework or the System property as indicated in previous section. But we can also define a version hardcoded in the string value and in that case this value is used. This version can contain a `-` itself when followed by `SNAPSHOT` or `RC`.

The <*jdk*> value indicates the JDK version of the Docker Image. When no value specified, the jdk11 image is used. There is also a JDK 17, 18 and 19 based image available.

## Volume mapping

With the `volumeMapping` member, it is possible to define mappings of directories between the host running the test and the container.

```
@AtbashContainerTest(volumeMapping = {"target/storage", "/opt/atbash/storage"})
```

The member accepts an array of Strings, but they always need to specified in pairs. If not, a JUnit Assertion Exception will be thrown.

Of each pair, the first one denotes the directory on the host.This can be an absolute or relative path against the directory used for running the test. This path will be converted to an absolute path and together with the second path which denotes an absolute path within the container it is used to create a volume mapping.

## Runtime Log

When the test case fails, the Runtime log is shown 'on the console' to facilitate what went wrong. You can also follow the log content during the execution of the test by setting the `liveLogging` member.

```
@AtbashContainerTest(liveLogging = true)
```

If you don't want to change the source code, you can always set the system property `atbash.test.container.logging.live` to true and the live logging will be activated.

Within your test, you can access the log content with the following statement.

```
String logContent = atbash.getLogs();
```

## Additional containers

If your test needs additional resources provided by other containers, The test framework will start them also at the beginning of the test. The *public static* field must be annotated by `@Container` and the type must be assignable to `GenericContainer`.

If you need a specific image of a container and can't use the default Testcontainers class, you can use the `DockerImageContainer` type of the framework. The variable name indicates the name of the directory where the Docker image definition (Dockerfile and optional the other files) is located (just as with the Custom images we saw earlier)

TODO: Describe how you can have multiple instances of the Atbash Runtime container to have a cluster.

## Container Ready

The framework waits up to 15 secs before the containers are ready and uses the vaue of the `/health` endpoint to determine if the process is ready. The

You can enlarge the wait time by defining a value for the `atbash_test_timeout_factor` environment value. It determines the factor for the increase of the wait time. A value of *2.0* results in a wait time of 30 secs.

## Remote Debugging

The official Docker Image can be used to start the Container with debug.

You can activate the debug mode for a test by using the `debug` member of the annotation.

```
@AtbashContainerTest(debug = true)
```

This adds the necessary options to the *JVM_ARGS* environment variable for the container to suspend the JVM startup until a debugger is connected. The wait time for the container to start up is also extended to 120 secs. Make sure you have connected the debugger to the port 5005 before this time limit.

## Atbash Embedded

The Atbash Runtime can also be embedded in your application. You can start the runtime with a Archive file and can respond to user requests.

The Embedded Runtime is mainly created for the Arquillian Connector.

### Dependency

Add the following Maven dependency to your project

```
<dependency>
    <groupId>be.atbash.runtime</groupId>
    <artifactId>runtime-embedded</artifactId>
    <version>${atbash.version}</version>
</dependency>
```

### Define Configuration

There is a Configuration Builder available to define the configuration for the Runtime.

```
    ConfigurationParameters parameters = new EmbeddedConfigurationBuilder(new File(
"myservice.war"))
            .withProfile("domain")
            .build();
```

Not all options of the configuration can be set as there are several options that are fixed. The configuration has always the stateless option active, no logging to file and no JFR or JMX active.

Based on the configuration, the Runtime can be started with

```
new AtbashEmbedded(parameters).start();
```

### Known issues

It is not possible to create a shaded executable jar file since that would make the entire code base a Bean Defining Archive in terms of CDI and will results in errors.

## Arquillian Connector

With the Atbash Runtime Arquillian Connector, you can run the Arquillian tests on the Atbash embedded product. This connector is mainly developped for running the TCK tests, but can also be used to test out the your application on the Atbash Runtime implementation.

In general, testing using the Atbash Testing framework, based on TestContainers will be easier and resulting in better tests as there is no need to create a specific archive with only the clases that that you want to test.

## Support

The connector is compiled against JDK 11 and requires Arquillian version 1.7.0 as that is the version that is using the `jakarta` namespace just as Atbash Runtime is.

The connector supports only deploying WAR files nd does not support deploying individual `Descriptor` s.

## Setup and configuration

You need to add the following dependecy to your project `pom.xml` file.

```
<dependency>
    <groupId>be.atbash.runtime.testing</groupId>
    <artifactId>arquillian-atbash-embedded</artifactId>
    <version>0.3</version>
    <scope>test</scope>
</dependency>
```

The selection of the connector can be done using the following snippet in the `arquillian.xml` file in your project.

```
<container qualifier="atbash" default="true">
    <configuration>
        <property name="keepArchive">true</property>
        <property name="profile">default</property>
    </configuration>
</container>
```

The connector has the following configuration properties

- keepArchive: parameter to indicate that the archive file in the temporary directory is not cleaned up after the test. This can be helpful if you want to inspect the contents of the wat file.
- profile: The name of the profile that must be activated for the test run. By default, this is the *default* profile but it can be any name that is defined in the *profiles.json* file.

# Remote CLI domain API

Overview of the endpoints when the domain mode is active and can be used to interact with the runtime remotely.

This API is also used by the `runtime-cli` tool.

Each API returns a JSON result.

### GET - domain/status

Returns the status and information of the runtime.

### POST - domain/deploy

Deploy application.

### GET - domain/list-applications

List all applications running on the runtime.

### POST - domain/undeploy

Undeploy an application

### POST - domain/set

Set configuration parameters for modules.

### POST - domain/set-logging-configuration

Set logging configuration parameters for logging module.

## Messages

Messages with a code number greater than 1000 are informational messages that are show in the log with the *verbose* option.
Values smaller than 1000 indicate some info, warning or error that is important for the operations of the environment.

### Core module

- DEPLOY-101(INFO): Start of the Deployment of the application.
- DEPLOY-102(INFO): End of the Deployment of the application.
- DEPLOY-103(WARNING): The archive cannot be found.
  When starting the Runtime and previously installed applications are started.
- DEPLOY-104(WARNING): The archive could not be inspected
  When deploying the application but the war file corrupt or the contents of the war file is empty.
- DEPLOY-105(WARNING): The archive cannot be found. When passing a file name on the command line.

- DEPLOY-106(WARNING): The archive file does not have the extension `.war`.

## Config module

- CONFIG-011(ERROR): Incorrect profile name specified.
- CONFIG-012(ERROR): Incorrect module name specified.
- CONFIG-013(ERROR): Unable to parse the content of a configuration file. (file is indicated in the error message)
- CONFIG-014(ERROR): The directory specified in the --root option does not exists.
- CONFIG-015(ERROR): The directory specified in the --root option is not a directory.
- CONFIG-016(ERROR): Cannot create the directory for the 'domain' (configuration, logs, …).
- CONFIG-017(ERROR): The directory for the 'domain' (configuration, logs, …) already exists and is not allowed for the Runtime CLI tool.
- CONFIG-018(WARN): The file *applications.json* cannot be written. This means that the application will not start the next time
- CONFIG-101(ERROR): The option specified for the *set* command must consist of <key>=<value> pairs separated by a space.
- CONFIG-102(ERROR): The key part of the option specified for the *set* command must be in the format <module>.<key-parts>
- CONFIG-103(ERROR): The line *n* in the configuration file could not be recognized as a valid command with parameters.
- CONFIG-104(INFO): Processing of the command on line *n* within the configuration file failed.

## Logging Module

## Jetty module

## Jersey module

# Expert

Some expert settings and information if you want to deep dive into the code.

## System property `traceModuleStartProcessing`

When setting the System property `traceModuleStartProcessing` you can trace each step in the parallel module startup by the `ModuleManager`` This can help you find out why the startup of the modules does not run as expected.