Meecrowave Configuration

Meecrowave configuration is centralized in `org.apache.meecrowave.Meecrowave$Builder` class.

Here are the main properties:

Name	Description
cdiConversation	Should CDI conversation be activated
clientAuth	HTTPS keystore client authentication
conf	Conf folder to synchronize
connectors	Custom connectors
cxfServletParams	Init parameters passed to CXF servlet
defaultSSLHostConfigName	The name of the default SSLHostConfig that will be used for secure https connections.
deleteBaseOnStartup	Should the directory be cleaned on startup if existing
dir	Root folder if provided otherwise a fake one is created in tmp-dir
host	Default host
http2	Activate HTTP 2
httpPort	HTTP port
httpsPort	HTTPS port
initializeClientBus	Should the client bus be set. If false the server one will likely be reused.
injectServletContainerInitializer	Should ServletContainerInitialize support injections.
jaxrsAutoActivateBeanValidation	Should bean validation be activated on JAX-RS endpoint if present in the classpath.
jaxrsDefaultProviders	If jaxrsProviderSetup is true the list of default providers to load (or defaulting to johnson jsonb and jsonp ones)
jaxrsLogProviders	Should JAX-RS providers be logged
jaxrsMapping	Default jaxrs mapping
jaxrsProviderSetup	Should default JAX-RS provider be configured
jaxwsSupportIfAvailable	Should @WebService CDI beans be deployed if cxf-rt-frontend-jaxws is in the classpath.
jsonbBinaryStrategy	Should JSON-B provider prettify the output
jsonbEncoding	Which encoding provider JSON-B should use
jsonbIJson	Should JSON-B provider comply to I-JSON
jsonbNamingStrategy	Should JSON-B provider prettify the output
jsonbNulls	Should JSON-B provider serialize nulls
jsonbOrderStrategy	Should JSON-B provider prettify the output

Name	Description
jsonbPrettify	Should JSON-B provider prettify the output
jsonpBufferStrategy	JSON-P JAX-RS provider buffer strategy (see johnzon)
jsonpMaxReadBufferLen	JSON-P JAX-RS provider read buffer limit size (see johnzon)
jsonpMaxStringLen	JSON-P JAX-RS provider max string limit size (see johnzon)
jsonpMaxWriteBufferLen	JSON-P JAX-RS provider write buffer limit size (see johnzon)
jsonpPrettify	Should JSON-P JAX-RS provider prettify the outputs (see johnzon)
jsonpSupportsComment	Should JSON-P JAX-RS provider support comments (see johnzon)
keepServerXmlAsThis	Don’t replace ports in server.xml
keyAlias	HTTPS keystore alias
keystoreFile	HTTPS keystore location
keystorePass	HTTPS keystore password
keystoreType	HTTPS keystore type
loggingGlobalSetup	Should logging be configured to use log4j2 (it is global)
loginConfig	web.xml login config
meecrowaveProperties	Loads a meecrowave properties, defaults to meecrowave.properties.
pidFile	A file path to write the process id if the server starts
properties	Passthrough properties
quickSession	Should an unsecured but fast session id generator be used
realm	realm
roles	In memory roles
scanningExcludes	A forced exclude list of jar names (comma separated values)
scanningIncludes	A forced include list of jar names (comma separated values)
scanningPackageExcludes	A forced exclude list of packages names (comma separated values)
scanningPackageIncludes	A forced include list of packages names (comma separated values)
securityConstraints	web.xml security constraint
serverXml	Provided server.xml

Name	Description
sharedLibraries	A folder containing shared libraries.
skipHttp	Skip HTTP connector
ssl	Use HTTPS
sslProtocol	HTTPS protocol
stopPort	Shutdown port if used or -1
tempDir	Temporary directory
tomcatAccessLogPattern	Activates and configure the access log valve. Value example: '%h %l %u %t "%r" %s %b "%{Referer}i" "%{User-Agent}i"'
tomcatAutoSetup	Add default servlet
tomcatFilter	A Tomcat JarScanFilter
tomcatNoJmx	(Experimental) Should Tomcat MBeans be skipped.
tomcatScanning	Should Tomcat scanning be used (@HandleTypes, @WebXXX)
tomcatWrapLoader	(Experimental) When deploying a classpath (current classloader), should meecrowave wrap the loader to define another loader identity but still use the same classes and resources.
useLog4j2JulLogManager	Should JUL logs be redirected to Log4j2 - only works before JUL usage.
useShutdownHook	Use shutdown hook to automatically stop the container on Ctrl+C
useTomcatDefaults	Should Tomcat default be set (session timeout, mime mapping etc...)
users	In memory users
watcherBouncing	Activate redeployment on directories update using this bouncing.
webResourceCached	Cache web resources
webSessionCookieConfig	Force the cookie-config, it uses a properties syntax with the keys being the web.xml tag names.
webSessionTimeout	Force the session timeout for webapps
webXml	Global web.xml

the class also provides some helper methods for programmatic use case like `randomHttpPort()` to automatically set an available port to `httpPort`.

You can also write a `Consumer<Builder>` to configure programmatically the `Builder` and make it active using `addCustomizer(Consumer<Builder>)`.

Example:

```
new Meecrowave(new Builder() {{
        randomHttpPort();
        setTomcatScanning(false);
        setTomcatAutoSetup(false);
        setRealm(new JAASRealm());
        user("admin", "secret");
     }})
    .bake()
    .await();
```

CDI SE API

CDI 2.0 introduces a "SE API" for CDI. It looks like:

```
try (final SeContainer container = SeContainerInitializer.newInstance()
        .disableDiscovery()
        .addBeanClasses(Configured.class)
        .initialize()) {
    // your main
}
```

Meecrowave inherits from OpenWebBeans SE API implementation and therefore this SE API will work out of the box.

It is implemented as a `bake()` and you can still access the `Builder` configuration or even `Meecrowave` itself if needed:

```
try (final SeContainer container = SeContainerInitializer.newInstance()
        .disableDiscovery()
        .addBeanClasses(Configured.class)
        .initialize()) {

    // use the configuration to access extensions, custom config or even server port
    Meecrowave.Builder config = container.select(Meecrowave.Builder.class).get();
    int port = config.getHttpPort();

    // default wait implementation relying on tomcat one
    container.select(Meecrowave.class).get().await(); // wait for the program to be
killed (tomcat.await() equivalent)

}
```

All the configuration of meecrowave is still available using properties:

```
try (final SeContainer container = SeContainerInitializer.newInstance()
     .addProperty("nameOfTheProperty", instanceInTheRightType)
     .initialize()) {
    container.select(Meecrowave.class).get().await();
}
```

The type should match the type expected by the `Builder` instance. Note you can also just pass directly a `Builder` instance as value (the property name is not important) if you want something preconfigured:

```
try (final SeContainer container = SeContainerInitializer.newInstance()
     .addProperty("meecrowaveConfiguration", new Meecrowave.Builder().randomPort())
     .initialize()) {
    container.select(Meecrowave.class).get().await();
}
```

Automatic configuration

The `org.apache.meecrowave.Meecrowave$Builder` class also provides `loadFromProperties(Properties)` and `loadFrom(String)`. The last one uses the parameter to locate a propertiers file (file path or at classpath) and delegate the processing to the first one.

`loadFromProperties(Propertiers)` loads the configuraton from the properties.

The matching is alsmot 1-1 with previous table excepted for these entries:

- if `httpPort` is `-1` then `randomHttpPort` is called
- `properties.x=y` will set the property (`properties` entry) `x` with the value `y`
- `users.x=y` will create the user `x` with the password `y`
- `roles.x=y` will create the role `x` with the users `y` (comma separated if multiple users)
- `cxf.servlet.params.x=y` will force the CXF servlet init parameter `x` to be `y`
- `connector.x=y` will pass the property `x` to be `y` on the connector. See the Apache Tomcat 9 Connector Documentation
- `connector.attributes.x=y` will use the property `x` with value `y` to create the connector (set a property on the instance of `org.apache.catalina.connector.Connector`) See the Connector attributes referenced in the Apache Tomcat 9 Connector Documentation
- `valves.*` will be used to create valves. This prefix must be followed by a valve identifier then you can use the built-in virtual attributes. These ones are `_order` to sort the valves (natural order) and `_className` to specify the class to instantiate. Finally you can use any dotted attribute to configure the valve (see example after this list).
- `realm=y` will create an instance of `y` (qualified name of the class) as `realm`
- `realm.x=y` will set `x` property to `y` - needs previous property to be set

- `login=` will create a custom `org.apache.meecrowave.Meecrowave$LoginConfigBuilder`
- `login.x=y` will customize previous instance with `x` property
- `securityConstraint=` will create a custom `org.apache.meecrowave.Meecrowave$SecurityConstaintBuilder`
- `securityConstraint.x=y` will customize previous instance with `x` property
- `configurationCustomizer=y` will create an instance of `y` to customize the configuration
- `configurationCustomizer.x=y` will set `x` to `y` for the customizer

Out of the box, any `Builder` instance will read `meecrowave.properties`. `meecrowave.properties` uses CLI names (without the leading `--`). See CLI page for the list.

Valve configuration

Here is an example to configure the `RemoteIpValve` and `LoadBalancerDrainingValve` using the `meecrowave.properties` syntax (which means it uses the `properties.` prefix to specify properties, drop it if you use the CLI options):

```
properties.valves.remote-ip._order = 1
properties.valves.remote-ip._className = org.apache.catalina.valves.RemoteIpValve
properties.valves.remote-ip.internalProxies = 192\\.168\\.0\\.10\|192\\.168\\.0\\.11
properties.valves.remote-ip.remoteIpHeader = x-forwarded-for
properties.valves.remote-ip.proxiesHeader = x-forwarded-by
properties.valves.remote-ip.trustedProxies = proxy1|proxy2

properties.valves.draining._order = 2
properties.valves.draining._className =
org.apache.catalina.valves.LoadBalancerDrainingValve
properties.valves.draining.redirectStatusCode = 307
properties.valves.draining.ignoreCookieName = draining-action
properties.valves.draining.ignoreCookieValue = skip
```

This will define the `remote-ip` and `draining` valves in this order with the configuration defined thanks to the properties not having an underscore at the beginning of their name.

Logging

Meecrowave relies by default on Log4j2 (see http://logging.apache.org/log4j/2.x/). By default it uses an internal configuration which is overridden by standard log4j mechanism.

Passwords/Secrets

For the configuration requiring to be ciphered you can implement `org.apache.meecrowave.service.ValueTransformer`:

```
public class MyTransformer implements ValueTransformer {
    @Override
    public String name() {
        return "mine";
    }

    @Override
    public String apply(final String encodedPassword) {
        return ....;
    }
}
```

this code being executed before the container starts you can't use CDI there.

To register your implementation just put the fully qualified name of your transformer in `META-INF/services/org.apache.meecrowave.service.ValueTransformer`.

Then to use it set the value to `decode:mine:encodedvalue`. General pattern is: `decode:<transformer name>:<value before decryption>`.

Note that by default the same ciphering algorithm than in TomEE is available (Static3DES).

This syntax is usable on the command line and in `meecrowave.properties`.

Programmatic customization

`org.apache.meecrowave.Meecrowave$ConfigurationCustomizer` can be used to customize the configuration programmatically before startup. It will take the `Builder` as parameter and you can change it at that moment.

`org.apache.meecrowave.Meecrowave$InstanceCustomizer` can be used to customize the configuration programmatically before startup. It will take the `Tomcat` as parameter and you can change it at that moment. This is very useful to automatically add valves and things like that.