Vert.x configuration

How to configure a Vert.x application

Last update: May 24, 2020. Tested using Vert.x 3.9.0.

Project: https://github.com/gualtierotesta/blog-projects/tree/master/vertx-configuration

Introduction

Configuration is usually required to customize the application behaviour based on the execution context. For example, the user credentials to access the DB are different in the development and production environments. External web services URLs can be also different in all environments.

The configuration can be saved in:

* A property file in the project resources, built inside the application package and read via the application class-path
* A property file outside the project and stored in the runtime server filesystem which can be read by the application at start-up.
* One or more JVM properties set using the -d flag of the java command
* One or more environment variables defined in the invoking shell

A Vert.x application can read its configuration using one or more (also all) of these alternative ways and we can also define a proper relative order among them, as we will see looking at the code.

In the examples below, we will consider a configuration parameter named "user" which will be defined using the different configuration solutions.

Vert.x configuration

Inside the Vert.x verticles, the configuration is available as JsonObject returned by the ( [https://vertx.io/docs/apidocs/io/vertx/core/AbstractVerticle.html#config](https://vertx.io/docs/apidocs/io/vertx/core/AbstractVerticle.html" \l "config) ) AbstractVerticle.config() function:

public class AVerticle extends AbstractVerticle {

@Override

public void start(final Promise<Void> startPromise) {

JsonObject configuration = config();

String user = configuration.getString("user");

//...

}

}

In the main application verticle, we can:

1. Read the configuration data

2. Pass the configuration to the deployed verticles

public class MainVerticle extends AbstractVerticle {

@Override

public void start(final Promise<Void> startPromise) {

Configurator.readConfiguration(vertx)

.onSuccess(configuration -> {

vertx.deployVerticle(new AVerticle(),

new DeploymentOptions()

.setConfig(configuration));

startPromise.complete();

})

.onFailure(startPromise::fail);

}

}

Line 5: we call the method Configurator.readConfiguration which returns the configuration data as Future. We will describe the class Configurator in the next section.

Lines 6-8: if configuration is available, we can deploy the other verticles (here just AVerticle) passing the configuration as deployment option.

The Configurator class

This class is a custom class, not a Vert.x class, which implements our configuration strategy:

1. where to get configuration data

2. which is the name, position and syntax (JSON, YAML..) of the configuration files

3. which is the relative order among the configuration sources (if more than one or, in other words, which configuration source wins over the others.

As already anticipated, we can define more than one configuration "source": the Configurator will read all of them in order and process the configuration properties in the same order.

What order does mean? If the same configuration property (same key) is defined in more than one configuration sources, the resulting configuration will have the property value of the <strong>last</strong> read source.

For example, let's assume we have a internal (class-path) resource file which defines the following property:

db.hostname=dev.example.com

the db.hostname key in the configuration JsonObject will have "dev.example.com" as value.

But, if the Configurator is configured to read the internal resource file AND the system properties (in this order), we can override the db.hostname key by invoking the application like the following:

java -Ddb.hostname=prod.example.com ....

the db.hostname key in the configuration JsonObject will have the "prod.example.com" as value.

Before looking the the Configurator source code, we should first introduce the Vert.x classes related to configuration:

1. ConfigRetriever is responsible of reading configurations data from all defined configuration sources and to generate a JsonObject which is the ordered merge of all configurations.

2. ConfigRetrieverOptions customizes the ConfigRetriever behaviour

3. ConfigStoreOptions defines a configuration source.

Configurator source

Configurator class:

class Configurator {

public static Future<JsonObject> readConfiguration(Vertx pVertx) {

Promise<JsonObject> promise = Promise.promise();

createConfigurationRetriever(pVertx).getConfig(promise::handle);

return promise.future();

}

// ..

}

The private createConfigurationRetriever method:

private static ConfigRetriever createConfigurationRetriever(Vertx pVertx) {

ConfigStoreOptions fileStoreExternal =

new ConfigStoreOptions()

.setType("file")

.setOptional(true)

.setConfig(new JsonObject()

.put("path", "/opt/config/config.json"));

ConfigStoreOptions fileStoreClasspath =

new ConfigStoreOptions()

.setType("file")

.setOptional(false)

.setConfig(new JsonObject()

.put("path", "config.json"));

ConfigStoreOptions sysPropsStore =

new ConfigStoreOptions().setType("sys");

ConfigStoreOptions envPropsStore =

new ConfigStoreOptions().setType("env");

ConfigRetrieverOptions options = new ConfigRetrieverOptions()

.addStore(fileStoreClasspath)

.addStore(fileStoreExternal)

.addStore(sysPropsStore)

.addStore(envPropsStore);

return ConfigRetriever.create(pVertx, options);

}

Line 4-9: we define a ConfigStoreOptions for an external file located in /opt/config/config.json. It is marked as optional so the file is not required to exists.

Line 11-16: we define a ConfigStoreOptions for a file named config.json e located at the root of the class-path. It's not optional.

Line 18:19: we define a ConfigStoreOptions for the system (JVM) properties.

Line 21:22: finally, we define a ConfigStoreOptions for the environment variables.

Line 24:28: we create an instance of the ConfigRetrieverOptions class add all the store previously defined.

Line 30: at the end we create the ConfigRetriever instance.

The addStore method calls order when creating the ConfigRetrieverOptions instance is very important: <em>the last added wins over the others.</em>

In our example we have the following order:

1. Environment variables

2. System properties

3. The external file (if exists)

4. The file in the class-path

Any environment variable will override configuration properties defined in the class-path file.

NOTE: there is no need to define and use all possible configuration sources.

Which configuration to use

The configuration file in the class-path should be always present:

1. to document all possible application configuration parameters, their names and possible values

2. to easy the application development. The parameters should possibly have a value which allows each single developer to run the application locally. For example the 'db.hostname’ key can have 'localhost' as value so the application can be run against a locally running database (by docker-compose, for example).

The parameters in the class-path file can then be overridden by the other configuration sources once the application is running on the execution environment (server, cloud, Kubernetes..).

For example, when deploying in Kubernetes, we can have a config map which maps the configuration into a file in the container or into environment variables.

On the contrary, in a server based traditional deployment, environment dependent configuration can be stored in a file on the server file system or as system properties in the application server JVM configuration.

Final comments

The ConfigRetriever supports also periodic re-reading of the configuration files and properties and it can generate events accordingly. See the ConfigRetrieverOptions.setScanPeriod method and the ConfigRetriever.listen method. These solution allows you to implement a dynamic reconfiguration of the application.