Ratpacked

Experience Ratpack through code snippets

Hubert A. Klein Ikkink

Ratpacked Notebook

Experience Ratpack with code snippets

Hubert Klein Ikkink

This book is for sale at http://leanpub.com/ratpacked-notebook

This version was published on 2023-04-19

Tweet This Book!

Please help Hubert Klein Ikkink by spreading the word about this book on Twitter!

The suggested hashtag for this book is #ratpackednotebook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#ratpackednotebook

Also By Hubert Klein Ikkink

Groovy Goodness Notebook

Grails Goodness Notebook

Gradle Goodness Notebook

Spocklight Notebook

Awesome Asciidoctor Notebook

This book is dedicated to my lovely family. I love you.

Contents

Configuration

Default Port Is 5050

When we use all the defaults for a Ratpack application the default port that is used for listening to incoming requests is 5050. This is something to remember, because we don't see it when we start the application. If we want to show it, for example in the console, we must add a SLF4J Logging API implementation. Ratpack uses SLF4J API for logging and the port number and address that is used for listening to incoming requests are logged at INFO level. We must add a runtime dependency to our project with a SLF4J API implementation. We provide the necessary logging configuration if needed and then when we start our Ratpack application we can see the port number that is used.

In the following example we use the Logback library as a SLF4J API implementation. We add a runtime dependency `ch.qos.logback:logback-classic:1.1.3` to our Gradle build file. If we would use another build tool, we can still use the same dependency. And also we add a simple Logback configuration file in the `src/main/resources` directory.

```
// File: build.gradle
plugins {
    id 'io.ratpack.ratpack-java' version '1.0.0'
}

mainClassName = 'com.mrhaki.ratpack.Main'

repositories {
    jcenter()
}

dependencies {
    // Here we add the Logback classic
    // dependency, so we can configure the
    // logging in the Ratpack application.
    runtime 'ch.qos.logback:logback-classic:1.1.3'
}
```

We create a Logback XML configuration file in `src/main/resources`:

```
<!-- File: src/main/resources/logback.xml -->
<configuration>

    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%msg%n</pattern>
        </encoder>
    </appender>

    <!-- Log all INFO level messages from RatpackServer -->
    <logger name="ratpack.server.RatpackServer" level="INFO"/>

    <root level="ERROR">
        <appender-ref ref="STDOUT"/>
    </root>

</configuration>
```

When we start the application with Gradle we can see in our console the logging messages from the `RatpackServer`. The last line shows the port number of our running Ratpack application:

```
$ gradle -q run
Starting server...
Building registry...
Ratpack started (development) for http://localhost:5050
```

Written with Ratpack 1.0.0.

Original blog post written on September 30, 2015.

Change Server Port With Environment Variable

When we define a Ratpack application we can set a server port in the server configuration code. When we do not define the port number in our code and use the default server configuration we can also set the server port with the environment variables `PORT` or `RATPACK_PORT`.

In the following example we use Gradle as build tool to run our Ratpack application. Gradle will pass on environment variables to the `run` task. We use the environment variable `RATPACK_PORT` to change the port to `9000`:

```
$ RATPACK_PORT gradle run
Starting server...
Building registry...
Ratpack started (development) for http://localhost:9000
```

Alternatively we can define the port number in our configuration, but also add an option to support environment variables to set or override configuration properties. This support is simply added by invoking the `env` method on the `ConfigDataBuilder` interface. With this method Ratpack looks for environment variables that start with `RATPACK_`. The rest

of the environment variable is then parsed and transformed to a configuration property. A double underscore is used as separator between sections, a single underscore is used as boundary for camel case fields. For example the environment variable `RATPACK_SERVER_-_PORT` transforms to `server.port`.

```
import static ratpack.groovy.Groovy.ratpack

ratpack {
    serverConfig {
        // Instruct Ratpack to look for
        // environment variables that
        // start with RATPACK_ as
        // configuration properties.
        env()
    }
}
```

Let's run our application and use the environment variable `RATPACK_SERVER__PORT` to change the port number:

```
$ RATPACK_SERVER__PORT=9000 gradle run
Starting server...
Building registry...
Ratpack started (development) for http://localhost:9000
```

We can alter the default prefix `RATPACK` for environment variables. We still use the method `env`, but this time we specify an argument as the prefix for the environment variables. In the following code we see how to use the prefix `COCKTAILS_` for environment variables:

```
package learning.ratpack;

import ratpack.server.RatpackServer;

public class Main {
    public static void main(String[] args) throws Exception {
        RatpackServer.start(server ->
            server
                .serverConfig(serverConfig ->
                    serverConfig
                        // Define prefix for environment
                        // variables that Ratpack uses.
                        .env("COCKTAILS_")));
    }
}
```

Let's run our application and use the environment variable `COCKTAILS_SERVER__PORT` to change the port number:

```
$ COCKTAILS_SERVER__PORT=9000 gradle run
Starting server...
Building registry...
Ratpack started (development) for http://localhost:9000
```

Written with Ratpack 1.0.0.

Original blog post written on October 02, 2015.

Start Ratpack With a Random Port Number

To start our Ratpack application with a random port number we must use the value `0` as port number. The value `0` tells Ratpack to use a random port number (in a safe port range).

In the following example Ratpack application we configure the server port to be random:

```
import static ratpack.groovy.Groovy.ratpack

ratpack {
    serverConfig {
        // Tell Ratpack to use a
        // random port number.
        port 0
    }
}
```

Written with Ratpack 1.0.0.

Original blog post written on October 02, 2015.

Externalized Application Configuration

Ratpack has very useful methods to apply application configuration in our application. We can read configuration properties from files in different formats, like JSON, YAML and Java properties, and the files can be read from different locations, like class path or file system. We can also set configuration properties via Java system properties on the command line or use environment variables.

We use the `ratpack.config.ConfigData` class with the static `of` method to add configuration properties to our application. We provide a lambda expression or Groovy closure to the `of` method to build our configuration. Here we specify external files, locations and other configuration options we want to include for our application. If the same configuration property is defined in multiple configuration sources Ratpack will apply the value that is last discovered. This way we can for example provide default values and allow them to be overridden with environment variables if we apply the environment variables last.

To use the values that are gathered we use the `get` method of the `ConfigData` instance. We can apply the configuration properties to the properties of a configuration class that is then automatically instantiated. We add this to the registry so we can use the configuration properties further down in our application.

In the following example Ratpack application we use different ways to apply configuration properties. It is inspired by how Spring Boot reads and applies externalized configuration properties. First we use a simple `Map` with default values, then the class path is scanned for files with names `application.yml`, `application.json` and `application.properties` in either the root of the class path or a `config` package. Next the same file names are searched on the file system relative from where the application is started. Next Java system properties starting with `sample.` are applied to the configuration. And finally environment variables starting with `SAMPLE_` are interpreted as configuration properties.

Let's start with the simple configuration class and properties we want to set via Ratpack's configuration ability:

```
// File: src/main/groovy/com/mrhaki/SampleConfig.groovy
package com.mrhaki

/**
 * Configuration properties for our application.
 */
class SampleConfig {

    /**
     * URL for external service to invoke with HTTP client.
     */
    String externalServiceUrl

    /**
     * URI to access the Mongo database.
     */
    String mongoUri

    /**
     * Indicate if we need to use a HTTP proxy.
     */
    boolean useProxy

    /**
     * Simple message
     */
    String message

}
```

Next we have a very simple Ratpack application. Here we use `ConfigData.of` with a lot of helper methods to read in configuration properties from different sources:

```
// File: src/ratpack/Ratpack.groovy
import com.google.common.io.Resources
import com.mrhaki.SampleConfig
import ratpack.config.ConfigData
import ratpack.config.ConfigDataBuilder

import java.nio.file.Files
import java.nio.file.Path
import java.nio.file.Paths

import static groovy.json.JsonOutput.prettyPrint
import static groovy.json.JsonOutput.toJson
import static ratpack.groovy.Groovy.ratpack

ratpack {

    bindings {
        final ConfigData configData = ConfigData.of { builder ->
            // Set default value, can be overridden by
            // configuration further down the chain.
            // The map must have String values.
            builder.props(['app.useProxy': Boolean.TRUE.toString()])

            loadExternalConfiguration(builder)

            // Look for system properties starting with
            // sample. to set or override configuration properties.
            builder.sysProps('sample.')

            // Look for environment variables starting
            // with SAMPLE_ to set or override configuration properties.
            builder.env('SAMPLE_')

            builder.build()
        }

        // Assign all configuration properties from the /app node
        // to the properties in the SampleConfig class.
        bindInstance(SampleConfig, configData.get('/app', SampleConfig))
    }

    handlers {
        get('configprops') { SampleConfig config ->
            render(prettyPrint(toJson(config)))
        }
    }

}

private void loadExternalConfiguration(final ConfigDataBuilder configDataBuilder) {

    final List<String> configurationLocations =
            ['application.yml',
             'application.json',
             'application.properties',
             'config/application.yml',
```

```
             'config/application.json',
             'config/application.properties']

    configurationLocations.each { configurationLocation ->
        loadClasspathConfiguration(configDataBuilder, configurationLocation)
    }

    configurationLocations.each { configurationLocation ->
        loadFileSystemConfiguration(configDataBuilder, configurationLocation)
    }
}

private void loadClasspathConfiguration(
        final ConfigDataBuilder configDataBuilder,
        final String configurationName) {

    try {
        final URL configurationResource = Resources.getResource(configurationName)
        switch (configurationName) {
            case yaml():
                configDataBuilder.yaml(configurationResource)
                break
            case json():
                configDataBuilder.json(configurationResource)
                break
            case properties():
                configDataBuilder.props(configurationResource)
                break
            default:
                break
        }
    } catch (IllegalArgumentException ignore) {
        // Configuration not found.
    }

}

private void loadFileSystemConfiguration(
        final ConfigDataBuilder configDataBuilder,
        final String configurationFilename) {

    final Path configurationPath = Paths.get(configurationFilename)
    if (Files.exists(configurationPath)) {
        switch (configurationFilename) {
            case yaml():
                configDataBuilder.yaml(configurationPath)
                break
            case json():
                configDataBuilder.json(configurationPath)
                break
            case properties():
                configDataBuilder.props(configurationPath)
                break
            default:
                break
        }
```

```
    }
}

private def yaml() {
    return hasExtension('yml')
}

private def json() {
    return hasExtension('json')
}

private def properties() {
    return hasExtension('properties')
}

private def hasExtension(final String extension) {
    return { filename -> filename ==~ /.*\.${extension}$/ }
}
```

Next we create some external configuration files:

```
# File: src/ratpack/application.yml
---
app:
  mongoUri: mongodb://mongo:27017/test


# File: src/ratpack/application.properties
app.externalServiceUrl = http://remote:9000/api
app.message = Ratpack rules!
```

Let's run the application and see the output of the `configprops` endpoint:

```
$ http localhost:5050/configprops
...
{
    "externalServiceUrl": "http://remote:9000/api",
    "useProxy": true,
    "message": "Ratpack rules!",
    "mongoUri": "mongodb://mongo:27017/test"
}
```

Next we stop the application and start it with the Java system property `-Dsample.app.useProxy=false` and the environment variable `SAMPLE_APP__MESSAGE='Ratpack rocks!'`. We check the `configprops` endpoint again:

```
$ http localhost:5050/configprops
...
{
    "externalServiceUrl": "http://remote:9000/api",
    "useProxy": false,
    "message": "Ratpack rocks!",
    "mongoUri": "mongodb://mongo:27017/test"
}
```

Written with Ratpack 1.0.0.

Original blog post written on November 03, 2015.

Use Command Line Arguments For Configuration

Ratpack 1.1 introduced a feature to use command line arguments for our application configuration. We must use the `args` method of the `ConfigDataBuilder` class. We can define a common prefix for the arguments and the separator between the configuration property and value. If we don't specify any arguments then Ratpack assumes there is no prefix and the separator is the equal sign (=).

In the following example Ratpack application we use the `args` method and rely on all the default settings:

```
// File: src/ratpack/Ratpack.groovy
import ratpack.config.ConfigData
import ratpack.config.ConfigDataBuilder

import static groovy.json.JsonOutput.prettyPrint
import static groovy.json.JsonOutput.toJson
import static ratpack.groovy.Groovy.ratpack

class SimpleConfig {
    String message
}

ratpack {

    bindings {
        final ConfigData configData = ConfigData.of { ConfigDataBuilder builder ->
            // We use the args method to use
            // command line arguments to configure
            // our application. The arguments
            // must have a key and value separated by
            // an equal sign (=).
            // We pass the argument list (args) which is
            // available in our Ratpack.groovy context.
            builder.args(args)

            builder.build()
        }

        // Assign all configuration properties from the /simple node
```

```
        // to the properties in the SimpleConfig class.
        bindInstance(SimpleConfig, configData.get('/simple', SimpleConfig))
    }

    handlers {
        get('configprops') { SimpleConfig config ->
            render(prettyPrint(toJson(config)))
        }
    }

}
```

If we use Gradle to run our application we can reconfigure the `run` task to pass command line arguments:

```
// File: build.gradle
...
run {
    args 'simple.message=Sample message'
}
...
```

If we run the application with our Gradle build file we can request the `configprops` path and get the following result:

```
$ http localhost:5050/configprops
...
{
    "message": "Sample message"
}

$
```

Written with Ratpack 1.1.1.

Original blog post written on November 16, 2015.

Using Database As Custom Configuration Source

We learned about externalised configuration in a previous blog post. Ratpack provides support out of the box for several formats and configuration sources. For example we can use files in YAML, properties or JSON format, arguments passed to the application, system properties and environment variables. We can add our own configuration source by implementing the `ratpack.config.ConfigSource` interface. We must override the method `loadConfigData` to load configuration data from a custom source and convert it to a format that can be handled by Ratpack.

We are going to write a custom `ConfigSource` implementation that will get configuration data from a database. We assume the data is in a table with the name `CONFIGURATION` and has the columns `KEY` and `VALUE`. The format of the key is the same as for Java properties files.

```
package com.mrhaki.config

import groovy.sql.Sql
import ratpack.config.ConfigSource
import ratpack.config.internal.source.AbstractPropertiesConfigSource

/**
 * {@link ConfigSource} implementation to read configuration
 * data from a database table. The database must have a table
 * CONFIGURATION with the VARCHAR columns KEY and VALUE.
 * The format of the key matches that of regular Java properties.
 *
 * E.g. we can insert the configuration key 'app.message' like this:
 * INSERT INTO CONFIGURATION(KEY, VALUE) VALUES('app.message', 'Ratpack rocks');
 *
 * This class extends {@link AbstractPropertiesConfigSource}, because it supports
 * property key formats like we use in our database.
 */
class JdbcConfigSource extends AbstractPropertiesConfigSource {

    /**
     * Database JDBC url, username, password and driver.
     */
    final Map<String, String> sqlProperties

    JdbcConfigSource(final Map<String, String> sqlProperties) {
        super(Optional.empty())
        this.sqlProperties = sqlProperties
    }

    @Override
    protected Properties loadProperties() throws Exception {
        final Properties properties = new Properties()

        Sql.withInstance(sqlProperties) { sql->
            // Get key/value pairs from database.
            sql.eachRow("SELECT KEY, VALUE FROM CONFIGURATION") { row ->
                // Save found data in Properties object.
                properties.setProperty(row.key, row.value)
            }
        }

        return properties
    }
}
```

To use this `ConfigSource` implementation in our Ratpack application we use the `add` method of `ConfigDataBuilder`:

```
import com.mrhaki.config.JdbcConfigSource
import org.slf4j.Logger
import org.slf4j.LoggerFactory
import ratpack.config.ConfigData

import static groovy.json.JsonOutput.prettyPrint
import static groovy.json.JsonOutput.toJson
import static ratpack.groovy.Groovy.ratpack

class SampleConfig {
    String message
}

ratpack {

    bindings {
        final ConfigData configData = ConfigData.of { builder ->
            // Add our custom JdbcConfigSource as
            // configuration source.
            builder.add(
                        new JdbcConfigSource(
                            driver: 'org.postgresql.Driver',
                            url: 'jdbc:postgresql://192.168.99.100:32768/',
                            user: 'postgres',
                            password: 'secret'))
                    .build()
        }

        // Assign all configuration properties from the /app node
        // to the properties in the SampleConfig class.
        bindInstance(SimpleConfig, configData.get('/app', SampleConfig))
    }

    handlers {
        get('configprops') { SampleConfig config ->
            render(prettyPrint(toJson(config)))
        }
    }

}
```

When we request `configprops` we get the following result:

```
$ http -b localhost:5050/configprops
{
    "message": "Ratpack rocks!"
}

$
```

Written with Ratpack 1.1.1.

Original blog post written on January 14, 2016.

Using Groovy Configuration Scripts As Configuration Source

Ratpack has a lot of options to add configuration data to our application. We can use for example YAML and JSON files, properties, environment variables and Java system properties. Groovy has the `ConfigSlurper` class to parse Groovy script with configuration data. It even supports an `environments` block to set configuration value for a specific environment. If we want to support Groovy scripts as configuration definition we write a class that implements the `ratpack.config.ConfigSource` interface.

We create a new class `ConfigSlurperConfigSource` and implement the `ConfigSource` interface. We must implement the `loadConfigData` method in which we read the Groovy configuration and transform it to a `ObjectNode` so Ratpack can use it:

```
// File: src/main/groovy/mrhaki/ratpack/config/ConfigSlurperConfigSource.groovy
package mrhaki.ratpack.config

import com.fasterxml.jackson.databind.ObjectMapper
import com.fasterxml.jackson.databind.node.ArrayNode
import com.fasterxml.jackson.databind.node.ObjectNode
import groovy.transform.CompileDynamic
import groovy.transform.CompileStatic
import ratpack.config.ConfigSource
import ratpack.file.FileSystemBinding

import java.nio.file.Path

@CompileStatic
class ConfigSlurperConfigSource implements ConfigSource {

    private final String configScript

    private final URL scriptUrl

    private final String environment

    ConfigSlurperConfigSource(final String configScript) {
        this(configScript, '')
    }

    ConfigSlurperConfigSource(final String configScript, final String environment) {
        this.configScript = configScript
        this.environment = environment
    }

    ConfigSlurperConfigSource(final Path configPath) {
        this(configPath, '')
    }

    ConfigSlurperConfigSource(final Path configPath, final String environment) {
        this(configPath.toUri(), environment)
    }

    ConfigSlurperConfigSource(final URI configUri) {
        this(configUri, '')
```

```
    }

    ConfigSlurperConfigSource(final URI configUri, final String environment) {
        this(configUri.toURL(), environment)
    }

    ConfigSlurperConfigSource(final URL configUrl) {
        this(configUrl, '')
    }

    ConfigSlurperConfigSource(final URL configUrl, final String environment) {
        this.scriptUrl = configUrl
        this.environment = environment
    }

    @Override
    ObjectNode loadConfigData(
            final ObjectMapper objectMapper,
            final FileSystemBinding fileSystemBinding) throws Exception {

        // Create ConfigSlurper for given environment.
        final ConfigSlurper configSlurper = new ConfigSlurper(environment)

        // Read configuration.
        final ConfigObject configObject =
                configScript ?
                        configSlurper.parse(configScript) :
                        configSlurper.parse(scriptUrl)

        // Transform configuration to node tree
        final ObjectNode rootNode = objectMapper.createObjectNode()
        populate(rootNode, configObject)
        return rootNode
    }

    @CompileDynamic
    private populate(final ObjectNode node, final ConfigObject config) {
        // Loop through configuration.
        // ConfigObject also implements Map interface,
        // so we can loop through key/value pairs.
        config.each { key, value ->
            // Value is another configuration,
            // this means the nested configuration
            // block.
            if (value instanceof Map) {
                populate(node.putObject(key), value)
            } else {
                // If value is a List we convert it to
                // an array node.
                if (value instanceof List) {
                    final ArrayNode listNode = node.putArray(key)
                    value.each { listValue ->
                        listNode.add(listValue)
                    }
                } else {
                    // Put key/value pair in node.
```

```
                    node.put(key, value)
                }
            }
        }
    }
}
```

We have several options to pass the Groovy configuration to the `ConfigSlurperConfigSource` class. We can use a `String`, `URI`, `URL` or `Path` reference. Let's create a file with some configuration data.

```
// File: src/ratpack/application.groovy
app {
    serverPort = 9000
}

environments {
    development {
        app {
            serverName = 'local'
        }
    }
    production {
        app {
            serverName = 'cloud'
            serverPort = 80
        }
    }
}
```

Next we create a Ratpack application using the Groovy DSL. In the `serverConfig` section we use our new `ConfigSlurperConfigSource`:

```
// File: src/ratpack/ratpack.groovy
import com.google.common.io.Resources
import com.mrhaki.config.ConfigSlurperConfigSource

import static groovy.json.JsonOutput.prettyPrint
import static groovy.json.JsonOutput.toJson
import static ratpack.groovy.Groovy.ratpack

//final Logger log = LoggerFactory.getLogger('ratpack')

ratpack {

    serverConfig {
        // Use Groovy configuration.
        add new ConfigSlurperConfigSource('''\
            app {
                message = 'Ratpack swings!'
            }''')

        // Use external Groovy configuration script file.
        add new ConfigSlurperConfigSource(
```

```
                Resources.getResource('application.groovy'), 'development')

        require '/app', SimpleConfig
    }

    handlers {
        get('configprops') { SimpleConfig config ->
            render(prettyPrint(toJson(config)))
        }
    }

}

// Simple configuration.
class SimpleConfig {
    String message
    String serverName
    Integer serverPort
}
```

Let's check the output of the `configprops` endpoint:

```
$ http -b localhost:5050/configprops
{
    "message": "Ratpack swings!",
    "serverName": "local",
    "serverPort": 9000
}
```

Now we set the environment to `production` in our Ratpack application:

```
// File: src/ratpack/ratpack.groovy
...

ratpack {

    serverConfig {
        ...

        // Use external Groovy configuration script file.
        add new ConfigSlurperConfigSource(
                Resources.getResource('application.groovy'), 'production')

       ...
    }

    ...
}
```

If we check `configprops` again we see different configuration values:

```
$ http -b localhost:5050/configprops
{
    "message": "Ratpack swings!",
    "serverName": "cloud",
    "serverPort": 80
}
```

Written with Ratpack 1.3.3.

Original blog post written on June 27, 2016.

Handling Exceptions When Reading Configuration Sources

To define configuration sources for our Ratpack application we have several options. We can set default properties, look at environment variables or Java system properties, load JSON or YAML formatted configuration files or implement our own configuration source. When something goes wrong using one of these methods we want to be able to handle that situation. For example if an optional configuration file is not found, we want to inform the user, but the application must still start. The default exception handling will throw the exception and the application is stopped. We want to customise this so we have more flexibility on how to handle exceptions.

We provide the configuration source in the `serverConfig` configuration block of our Ratpack application. We must add the `onError` method and provide an error handler implementation before we load any configuration source. This error handler will be passed to each configuration source and is execute when an exception occurs when the configuration source is invoked. The error handler implements the `Action` interface with the type `Throwable`. In our implementation we can for example check for the type of `Throwable` and show a correct status message to the user.

In the following example application we rely on external configuration source files that are optional. If the file is present it must be loaded, otherwise a message must be shown to indicate the file is missing, but the application still starts:

```
// File: src/ratpack/ratpack.groovy
import org.slf4j.Logger
import org.slf4j.LoggerFactory

import java.nio.file.NoSuchFileException
import java.nio.file.Paths

import static ratpack.groovy.Groovy.ratpack

final Logger log = LoggerFactory.getLogger('ratpack.server')

ratpack {
    serverConfig {
        // Use custom error handler, when
        // exceptions happen during loading
        // of configuration sources.
        onError { throwable ->
            if (throwable in NoSuchFileException) {
```

```
                final String file = throwable.file
                log.info "Cannot load optional configuration file '{}'", file
            } else {
                throw throwable
            }
        }

        yaml('application.yml')

        // Optional configuration files
        // to override values in
        // 'application.yml'. This could
        // potentially give an exception if
        // the files don't exist.
        yaml(Paths.get('conf/application.yml'))
        json(Paths.get('conf/application.json'))

        args(args)
        sysProps()
        env()

        ...
    }

    ...
}
```

Next we start the application with the absence of the optional configuration files `conf/application.yml` and `conf/application.json`:

```
$ gradle run
...
:run

12:28:38.887 [main]            INFO  ratpack.server.RatpackServer - Starting server...
12:28:39.871 [main]            INFO  ratpack.server - Cannot load optional configuration file 'conf/ap\
plication.yml'
12:28:39.873 [main]            INFO  ratpack.server - Cannot load optional configuration file 'conf/ap\
plication.json'
12:28:40.006 [main]            INFO  ratpack.server.RatpackServer - Building registry...
12:28:40.494 [main]            INFO  ratpack.server.RatpackServer - Ratpack started (development) for \
http://localhost:5050
```

Notice that application is started and in the logging we have nice status messages that tell us the files are not found.

Written with Ratpack 1.3.3.

Original blog post written on June 24, 2016.

Apply Configuration To Configurable Module

In Ratpack we can use Guice modules to organise code to provide objects to the registry. Ratpack adds configurable modules that take an extra configuration object. Values from

the configuration object are used to create new objects that are provided to our application. Using the Groovy DSL we have several ways to make a configuration object available to a configurable module.

Let's create a sample configurable module and look at the different ways to give it a configuration object. We create a module, `GreetingModule`, that provides a `Greeting` object. The configuration is defined in the class `GreetingConfig`.

```
// File: src/main/groovy/com/mrhaki/ratpack/GreetingModule.groovy
package com.mrhaki.ratpack

import com.google.inject.Provides
import groovy.transform.CompileStatic
import ratpack.guice.ConfigurableModule

@CompileStatic
class GreetingModule extends ConfigurableModule<GreetingConfig> {

    @Override
    protected void configure() {}

    /**
     * Method to create a new Greeting object with values
     * from the GreetingConfig configuration object.
     */
    @Provides
    Greeting greeting(final GreetingConfig config) {
        new Greeting(message: "${config.salutation}, ${config.message}")
    }

}



// File: src/main/groovy/com/mrhaki/ratpack/GreetingConfig.groovy
package com.mrhaki.ratpack

import groovy.transform.CompileStatic

/**
 * Configuration properties for creating
 * a {@link Greeting} object using the
 * configurable module {@link GreetingModule}.
 */
@CompileStatic
class GreetingConfig {
    String message
    String salutation
}
```

```
// File: src/main/groovy/com/mrhaki/ratpack/Greeting.groovy
package com.mrhaki.ratpack

import groovy.transform.CompileStatic

/**
 * Simple class with a greeting message.
 * The {@link GreetingModule} module creates
 * an instance of this class.
 */
@CompileStatic
class Greeting {
    String message
}
```

We have our configurable module and the supporting classes. Now we define the configuration and module in the `bindings` block of our Ratpack Groovy application. We use the `ConfigData` class to set configuration properties we later bind to the `GreetingConfig` object. With `ConfigData` we can define configuration properties from different sources, like files in Yaml, JSON or Java properties format. In our sample we simply define them using a `Map`. In the first sample we use the `module` method with a `Closure` argument to configure the `GreetingModule`

```
// File: src/ratpack/Ratpack.groovy
import com.mrhaki.ratpack.Greeting
import com.mrhaki.ratpack.GreetingConfig
import com.mrhaki.ratpack.GreetingModule
import ratpack.config.ConfigData
import ratpack.config.ConfigDataBuilder

import static ratpack.groovy.Groovy.ratpack

ratpack {
    bindings {
        final ConfigData configData = ConfigData.of { ConfigDataBuilder builder ->
            builder.props(
                    ['greeting.salutation': 'Hi',
                     'greeting.message'   : 'how are you doing?'])
            // We can also use external files,
            // system properties and environment
            // variables to set configuration properties
            // inside this block.
            builder.build()
        }

        // The module methods allows a Closure argument to
        // set values for the supported configuration class.
        module GreetingModule, { GreetingConfig config ->
            config.salutation = configData.get('/greeting/salutation', String)
            config.message = configData.get('/greeting/message', String)
        }
    }

    handlers {
        // Simple handler using the created Greeting object.
```

```
        get { Greeting greeting ->
            render greeting.message
        }
    }
}
```

In the following sample we create an instance of `GreetingConfig` using the `bindInstance` method. The `GreetingModule` will pick up this instance and use it automatically:

```
// File: src/ratpack/Ratpack.groovy
import com.mrhaki.ratpack.Greeting
import com.mrhaki.ratpack.GreetingConfig
import com.mrhaki.ratpack.GreetingModule
import ratpack.config.ConfigData
import ratpack.config.ConfigDataBuilder

import static ratpack.groovy.Groovy.ratpack

ratpack {
    bindings {
        final ConfigData configData = ConfigData.of { ConfigDataBuilder builder ->
            builder.props(
                    ['greeting.salutation': 'Hi',
                     'greeting.message'   : 'how are you doing?'])
            // We can also use external files,
            // system properties and environment
            // variables to set configuration properties
            // inside this block.
            builder.build()
        }

        // First create an instance of the GreetingConfig class.
        bindInstance GreetingConfig, configData.get('/greeting', GreetingConfig)
        // With the module method we use the GreetingModule and
        // because there is a GreetingConfig instance available
        // it is used to configure the module.
        module GreetingModule
    }

    handlers {
        // Simple handler using the created Greeting object.
        get { Greeting greeting ->
            render greeting.message
        }
    }
}
```

Finally we can use the `moduleConfig` method. This methods accepts a configuration object as an argument directly. So we can use an instance of `GreetingConfig` as method argument. We can also combine it with a `Closure` argument to override configuration properties:

```
// File: src/ratpack/Ratpack.groovy
import com.mrhaki.ratpack.Greeting
import com.mrhaki.ratpack.GreetingConfig
import com.mrhaki.ratpack.GreetingModule
import ratpack.config.ConfigData
import ratpack.config.ConfigDataBuilder

import static ratpack.groovy.Groovy.ratpack

ratpack {
    bindings {
        final ConfigData configData = ConfigData.of { ConfigDataBuilder builder ->
            builder.props(
                    ['greeting.salutation': 'Hi',
                     'greeting.message'   : 'how are you doing?'])
            // We can also use external files,
            // system properties and environment
            // variables to set configuration properties
            // inside this block.
            builder.build()
        }

        // With the moduleConfig method we can pass an instance
        // of GreetingConfig directly.
        moduleConfig GreetingModule, configData.get('/greeting', GreetingConfig)

        // Or we can even combine it with a Closure argument to override a
        // configuration property.
        //moduleConfig(
        //        GreetingModule,
        //        configData.get('/greeting', GreetingConfig)) { GreetingConfig config ->
        //    config.message = 'Ratpack rocks!'
        //}
    }

    handlers {
        // Simple handler using the created Greeting object.
        get { Greeting greeting ->
            render greeting.message
        }
    }
}
```

Written with Ratpack 1.1.1.

Original blog post written on November 11, 2015.