Building microservices with Spring Boot – part 1

Posted on April 1, 2014

This article introduces the concept of a microservice architecture and the motivations for using this architectural approach. It then shows how Spring Boot, a relatively new project in the Spring ecosystem can be used to significantly simplify the development and deployment of a microservice. You can find the example code on github.

What are microservices?

Since the earliest days of Enterprise Java, the most common way of deploying an application has been to package all the application's server-side components as a single war or ear file. This so-called monolithic architecture has a number of benefits. Monolithic applications are simple to develop since IDEs and other tools are oriented around developing a single application. They are also simple to deploy since you just have to deploy the one war/ear file on the appropriate container.

However, the monolithic approach becomes unwieldy for complex applications. A large monolithic application can be difficult for developers to understand and maintain. It is also an obstacle to frequent deployments. To deploy changes to one application component you have to build and deploy the entire monolith, which can be complex, risky, time consuming, require the coordination of many developers and result in long test cycles. A monolithic architecture can also make it difficult to trial and adopt new technologies and so you are often stuck with the technology choices that you made at the start of the project.

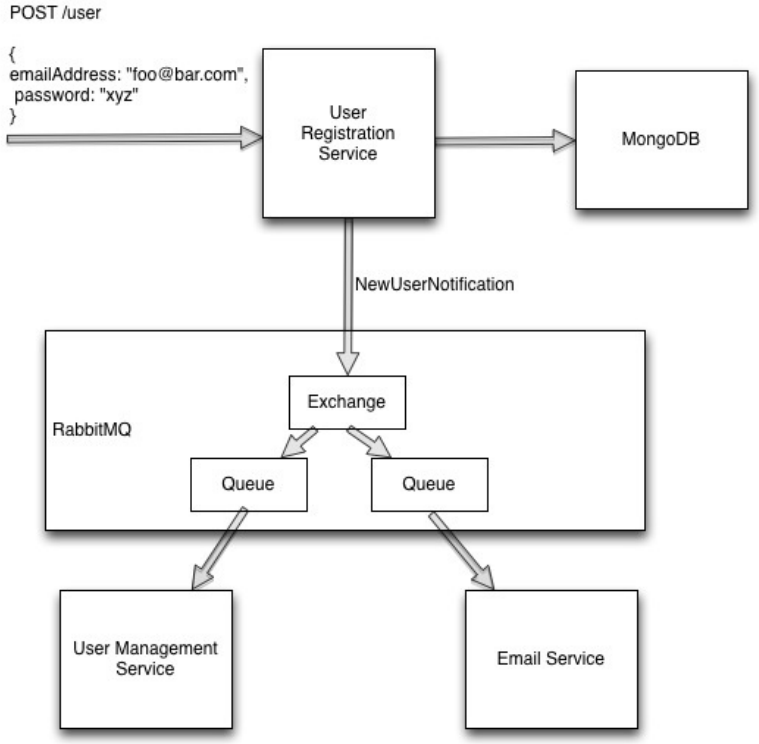
To avoid these problems, a growing number of organizations are using a microservice architecture. The application is functionally decomposed into a set of services. Each service has a narrow, focused set of responsibilities, and is, in some cases, quite small. For example, an application might consist of services such as the order management service, the customer management service etc.

Microservices have a number of benefits and drawbacks. A key benefit is that services are developed and deployed independently of one another. Another key benefit is that different services can use different technologies. Moreover, since each service is typically quite small, it's practical to rewrite it using a different technology. As a result, microservices make it easier to trial and adopt new, emerging technologies. One major drawback of microservices is the additional complexity – development, and deployment – of distributed systems. For most large applications, however, the benefits outweigh the drawbacks.

You can learn more about microservices by visiting microservices.io.

Developing with micro-services

Let's imagine that you are implementing user registration as part of an application that has a micro-service architecture. Users register by entering their email address and a password. The system then initiates an account creation workflow that includes creating the account in the database and sending an email to confirm their address. We could deploy the user registration components (controllers, services, repositories, ... etc.) as part of some other service. However, user registration is a sufficiently isolated chunk of functionality and so it makes sense, to deploy it as a standalone micro-service. In a later blog post, we will look at the web UI part of user registration but for now we will focus on the backend service. The following diagram shows the user registration service and how it fits into the overall system architecture.



The backend user registration service exposes a single RESTful endpoint for registering users. A registration request contains the user's email address and password. The service verifies that a user with that email address has not previously registered and publishes a message notifying the rest of the system that a new user has registered. The notification is consumed by various other services including the user management service, which maintains user accounts, and the email service, which sends a confirmation email to the user.

It's quite straightforward to implement the user registration backend using various projects in the Spring ecosystem. Here is the Spring framework controller, which is written in Scala, that implements the RESTful endpoint.

```
@RestController
class UserRegistrationController @Autowired()(…) {

@RequestMapping(value = Array("/user"),
            method = Array(RequestMethod.POST))
def registerUser(@RequestBody request: RegistrationRequest) = {

  val registeredUser =
    new RegisteredUser(null,
        request.emailAddress, request.password)

  registeredUserRepository.save(registeredUser)

  rabbitTemplate.convertAndSend(exchangeName, routingKey,
          NewRegistrationNotification(registeredUser.id,
```

```
                            request.emailAddress, request.password)
  RegistrationResponse(registeredUser.id, request.emailAddress)
}

@ResponseStatus(value = HttpStatus.CONFLICT,
        reason = "duplicate email address")
@ExceptionHandler(Array(classOf[DuplicateKeyException]))
def duplicateEmailAddress() {}
}
```

The @RestController annotation specifies that Spring MVC should assume that controller methods have an @ResponseBody annotation by default.

The registerUser() method records the registration in a database and then publishes a notification announcing that a user has registered. It calls the RegisteredUserRepository.save() method to persist a new registered user. Here is the RegisteredUserRepository, which provides access to the database of user registrations.

```
trait RegisteredUserRepository extends MongoRepository[RegisteredUser, String]

case class RegisteredUser(
    id : String,
    @(Indexed@field)(unique = true) emailAddress : String,
    password : String)
```

Notice that we do not need to supply an implementation of this interface. Instead, Spring Data for Mongo creates one automatically. Moreover, Spring Data for Mongo notices the @Indexed annotation on the emailAddress parameter and creates a unique index. If save() is called with an already existing email address it throws a DuplicateKeyException, which is translated by the duplicateEmailAddress() exception handler into an HTTP response with a status code of 409 (Conflict).

The UserRegistrationController also uses Spring AMQP to notify the rest of the application that a user has registered:

```
class UserRegistrationController @Autowired()(…) {
…
rabbitTemplate.convertAndSend(exchangeName, routingKey,
     NewRegistrationNotification(registeredUser.id,
                request.emailAddress, request.password))
…
}

case class NewRegistrationNotification(
  id: String, emailAddress: String, password: String)
```

The convertAndSend() method converts the NewRegistrationNotification to JSON and sends a message to the user-registrations exchange.

So far, so good! With just a few lines of code we have implemented the desired functionality. But in order to have a complete deployable application there are a few more things we need to take care of.

- Configure Spring dependency injection to instantiate and assemble these components along with the needed infrastructure components (RabbitTemplate, MongoTemplate, etc and their dependencies) into an application.

- Externalize message broker and MongoDB connection configuration so that we can build the war file once and run it in different environments: e.g. CI, QA, staging, production, etc.
- Configure logging etc.
- Decide how we are going to package and deploy the application.

And, oh yes, we had better write some tests.

Towards a deployable application

The Spring framework provide three main ways of configuring dependency injection: XML, annotations, and Java-based configuration. My preferred approach is to use a combination of annotations and Java-based configuration. I avoid XML-based configuration as much as possible unless it is absolutely necessary.

We could just launch an IDE, annotate the classes, and write the Java configuration classes and before long we would have a correctly configured application. The trouble with this old-style approach of manually crafting the each application's configuration is that we regularly create new microservices. It would become quite tedious to create very similar configurations over and over again even if we did just copy and paste from one service to another.

Similarly, to deploy the service, we could install and configure Tomcat or Jetty to run this service. But once again, in the course of building many microservices, this is something we would have to do repeatedly. There needs to be better way of dealing with both application and web container configuration that avoids all this duplication. We need an approach that lets us focus on getting things done for both web and non-web (e.g. message-based) applications.

About Spring Boot

One technology that lets you focus on getting things done is one of the newer members of the Spring ecosystem: the Spring Boot project. This project has two main benefits. The first benefit is that Spring Boot dramatically simplifies application configuration by taking Convention over Configuration (CoC) in Spring applications to a whole new level. Spring Boot has a feature called auto-configuration that intelligently provides a set of default behaviors that are driven by what jars are on the classpath. For example, if you include database jars on the classpath then Spring Boot will define DataSource and JdbcTemplate beans unless you have already defined them. As a result, it's remarkably easy to get a new micro-service up and running with little or no configuration while preserving the ability to customize your application.

The second benefit of Spring Boot is that it simplifies deployment by letting you package your application as an executable jar containing a pre-configured embedded web container (Tomcat or Jetty). This eliminates the need to install and configure Tomcat or Jetty on your servers. Instead, to run your micro-service you simply need to have Java installed. Moreover, the executable jar format provides uniform and self-contained way of packaging and running JVM applications regardless of type, which simplifies operations. If necessary, you can, however, configure Spring Boot to build a war file. Let's illustrate these features by developing a Spring Boot version of the user registration microservice.

Using Spring Boot to implement user registration

The Spring Boot part of the application consists of four pieces: a build.gradle (or Maven pom.xml), one or more Java Configuration classes, a configuration properties file, which defines connection settings for the

message broker and Mongo database, and a main() method class. Let's look at each one in turn.

build.gradle

The build.gradle file configures the Spring Boot build plugin, which creates the executable jar file. The build.gradle file also declares dependencies on Spring Boot artifacts. Here is the file.

```
buildscript {
  repositories {
    maven { url "http://repo.spring.io/libs-snapshot" }
    mavenCentral()
  }
  dependencies {
    classpath("org.springframework.boot:spring-boot-gradle-plugin:1.0.0.RC5")
  }
}

apply plugin: 'scala'
apply plugin: 'spring-boot'

dependencies {
  compile "org.scala-lang:scala-library:2.10.2"
  compile 'com.fasterxml.jackson.module:jackson-module-scala_2.10:2.3.1'

  compile "org.springframework.boot:spring-boot-starter-web"
  compile "org.springframework.boot:spring-boot-starter-data-mongodb"
  compile "org.springframework.boot:spring-boot-starter-amqp"

  testCompile "org.springframework.boot:spring-boot-starter-test"
  testCompile "org.scalatest:scalatest_2.10:2.0"
}

repositories {
  mavenCentral()
  maven { url 'http://repo.spring.io/milestone' }
}
```

The Spring Boot build plugin builds and configures the executable war file to execute the main() method defined in the project.

What's particularly interesting about build.gradle is that it defines dependencies on Spring Boot starter artifacts. Starter artifacts (aka. starters) use the naming convention spring-boot-starter-X, which X is the type of application that you are building. By depending on a starter you get a consistent set of dependencies for building applications of type X along with the appropriate auto-configuration behavior.

Since this service is a web application that uses MongoDB and AMQP, it defines the dependencies on the following starters:

- spring-boot-starter-web – includes the jars required by a web application such as Tomcat and Spring MVC
- spring-boot-starter-data-mongodb – includes the jars required by a MongoDB application including the MongoDB driver and Spring Data for Mongo.
- spring-boot-starter-amqp – includes the jars required by an AMQP application including Spring Rabbit

All of these starters also depend on spring-boot-starter, which provides auto-configuration, logging, and YAML configuration file supports.

Java configuration class(es)

A typical Spring Boot application needs at least one Spring bean annotated with @EnableAutoConfiguration, which enables auto-configuration. For example, the Spring Boot Hello World consists of a single class that's annotated with both @Controller and @EnableAutoConfiguration. Since the user registration service is more complex it has a separate Java Configuration class.

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
class UserRegistrationConfiguration {

  import MessagingNames._

  @Bean
  @Primary
  def scalaObjectMapper() = new ScalaObjectMapper

  @Bean
  def rabbitTemplate(connectionFactory : ConnectionFactory) = {
    val template = new RabbitTemplate(connectionFactory)
    val jsonConverter = new Jackson2JsonMessageConverter
    jsonConverter.setJsonObjectMapper(scalaObjectMapper())
    template.setMessageConverter(jsonConverter)
    template
  }

  @Bean
  def userRegistrationsExchange() = new TopicExchange("user-registrations")

}
```

The UserRegistrationConfiguration class has three annotations: @Configuration, which identifies the class as a Java Configuration class, @EnableAutoConfiguration, which was discussed above, along with @ComponentScan, which enables component scanning for the controller.

The UserRegistrationConfiguration class defines three custom beans:

- scalaObjectMapper – A Jackson JSON ObjectMapper that registers the DefaultScalaModule, which provides support for Scala objects. The ObjectMapper is used by the RabbitTemplate for serializing outgoing messaging and by Spring MVC for request/response serialization/deserialization
- rabbitTemplate – configures a RabbitTemplate that uses the ScalaObjectMapper so that NewRegistrationNotification messages are sent in JSON format
- userRegistrationsExchange– ensures via RabbitAdmin that there is an AMQP Topic Exchange called user-registrations

There is remarkably little configuration for this kind of application. That's because Spring Boot's auto-configuration creates several beans for you:

- Spring MVC – Dispatcher servlet and the HttpMessageConverters that are configured to use Jackson JSON and the ScalaObjectMapper
- AMQP – RabbitAdmin and ConnectionFactory
- Mongo – Mongo driver and MongoTemplate

UserRegistrationMain

This class defines the main() method that runs the application. It's a one liner that calls the SpringApplication.run() method passing in the configuration class and the args parameter to main().

```
object UserRegistrationMain {

  def main(args: Array[String]) : Unit =
    SpringApplication.run(classOf[UserRegistrationConfiguration], args :_ *)

}
```

The SpringApplication class is provided by Spring Boot. It's run() method creates and starts the web container that runs the application.

application.properties

This file contains property settings that define how the application connects to the RabbitMQ server and the MongoDB database. It currently defines one property:

```
spring.data.mongodb.uri=mongodb://localhost/userregistration
```

This property specifies that the application should connect to the Mongo host running locally on the default port and use the userregistration database rather than the default test database.

This default configuration can be overridden in a couple of different ways. One option is to specify properties values on the command line when running the application. The other option is to supply additional application.properties files, which override all or some of the properties. This is done using either system properties or by putting the files in the current directory or on the classpath. See the documentation for the exact details on how Spring Boot locates properties files.

Putting it all together

With these two files and two classes, we can now build the application. Running ./gradlew build compiles the application, builds the executable jar and runs the tests. You can then execute the jar file to start the application:

```
$ java -jar build/libs/spring-boot-restful-service.jar
…
2014-03-28 09:20:13.423 INFO 57472 --- [ main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat start
2014-03-28 09:20:13.426 INFO 57472 --- [ main] n.c.m.r.main.UserRegistrationMain$ : Started UserRegist
```

Once built, this jar can move through the deployment pipeline to production. You can, for example, change the MongoDB connection URL by specifying the property on the command line:

```
$ java -jar build/libs/spring-boot-restful-service.jar \
--spring.data.mongodb.uri=mongodb://productionMongo/userregistration
```

Quite remarkable, given how little effort was required! Don't forget to look at the code on github.

Summary

As you can see, Spring Boot lets you focus on developing your microservices. It dramatically reduces the amount of application and server configuration that you would normally need to write. Furthermore, it's extremely easy to build an executable jar file that can be run on any machine with Java installed – no need to install and configure an application server. In later posts, we will look at other aspects of developing microservices with Spring Boot including web application development (see part 2 of this series), and automated testing, as well as look at how Spring Boot simplifies monitoring and management.

Next steps

- Read up on the Spring Boot project
- Learn more about microservices at Microservices.io
- Look at the user registration service code on github

This entry was posted in architecture, microservices and tagged architecture, microservices, REST, spring boot. Bookmark the permalink.

12 Responses to *Building microservices with Spring Boot – part 1*

Pingback: *The Baeldung Weekly Review 13*

Pingback: *Building microservices with Spring Boot – part 2 | plain old objects*

Pingback: *Article: Building microservices with Spring Boot | Braveterry*

Pingback: *Deploying Spring Boot-based microservices with Docker | plain old objects*

Tai Truong *says:*
December 20, 2014 at 5:08 am

This is exactly what I've been looking for. I am trying to set this up. All I did is to import the build.gradle into IntelliJ. I can start it but the tests fails with these errors:

org.springframework.amqp.AmqpConnectException: java.net.ConnectException: Connection refused: connect
…
com.mongodb.MongoTimeoutException: Timed out after 10000 ms while waiting for a server that matches AnyServerSelector{}. Client view of cluster state is {type=Unknown, servers=[{address=localhost:27017, type=Unknown, state=Connecting, exception={com.mongodb.MongoException$Network: Exception opening the socket}, caused by {java.net.ConnectException: Connection refused: connect}}]
…

The latter MongoTimeoutException I can imagine I just need to install MongoDB – but for now I try to switch to HSQL.

But what about Amqp/Docker? I am not familliar with this. Do I need to install it and run it separately from Spring Boot?

Thanks, Tai

Reply

ceracm *says:*
December 20, 2014 at 9:08 am

Yes. To run the application you need MongoDB and RabbitMQ.

These days I find it extremely convenience to run MongoDB and RabbitMQ as Docker containers. Rather than having to figure out how to install them (and other middleware your apps might need) you install Docker and run pre-packaged Docker containers containing the middleware.

Assuming that you aren't running on a Linux machine, the first thing to do is install boot2docker – https://docs.docker.com/installation/#installation

The directory https://github.com/cer/microservices-examples/tree/master/spring-boot-restful-service/docker has some useful scripts:
* run-dependencies.sh – runs MongoDB and RabbitMQ as docker containers
* set-env.sh – sets environment variables so that the Spring Boot application can connect to MongoDB and RabbitMQ

I hope this helps.

Reply

Pingback: *Tools for Crushing High Availability and Scalability, Part I*

Tim Goeke *says:*
March 2, 2015 at 7:41 pm

Thx again Chris, your articles are great. I was having trouble with the unique index and found that for recent Spring Data versions, there must be an @Document leading the case class or the index won't be created.

@Document
case class RegisteredUser(id : String, @(Indexed@field)(unique = true) emailAddress : String,

Reply

ceracm *says:*
March 3, 2015 at 1:30 pm

Yes. I encountered that problem too. Thanks for reminding me to push the change.

Reply

Kevin Andres *says:*
February 22, 2016 at 6:16 pm

This is a very useful walkthrough, and I really like what Spring boot can do for developing these microservices. However, I am having one issue, which is preventing me from running this application as an executable, and it seems to be related to Mongo. I followed your instructions on running both MongoDB and RabbitMQ as docker containers (as I don't have mongo actually installed on my machine), as I was running into the same issue that the previous commenter had with the tests failing when invoking './gradlew build' due to:
com.mongodb.MongoTimeoutException: Timed out after 10000 ms while waiting for a server that matches AnyServerSelector{}. Client view of cluster state is {type=Unknown, servers=[{address=localhost:27017, type=Unknown, state=Connecting, exception={com.mongodb.MongoException$Network: Exception opening the socket}, caused by {java.net.ConnectException: Connection refused}}]

I found this strange, as 'docker ps' tells me that I have both microservicesexamples_rabbitmq_1 and microservicesexamples_mongodb_1 running as containers on boot2docker, so I ran the build-and-test-all.sh script (which succeeded, along with the tests), and then I tried the './gradlew build' again from the spring-boot-restful-service directory, and this time around, the tests passed and the build succeeded, generating the jar file. However, when I try to execute the jar file with 'java -jar build/libs/spring-boot-restful-service.jar', i receive the same "MongoTimeoutException: timed out after 1000 ms…" stack trace once again

DEBUG 5376 — [ost-startStop-1] o.s.data.mongodb.core.MongoDbUtils : Getting Mongo Database name=[userregistration]
ERROR 5376 — [ost-startStop-1] o.s.b.c.embedded.tomcat.TomcatStarter : Error starting Tomcat context:
org.springframework.beans.factory.BeanCreationException
WARN 5376 — [main] ationConfigEmbeddedWebApplicationContext : Exception encountered during context initialization – cancelling refresh attempt

So it seems like MongoDb is still timing out, even though I have run the docker/run-dependencies.sh and docker/set-env.sh scripts beforehand? Any advice on where I might have something missing, or how MongoDB should be started/ran to avoid the timeout? Thanks!

-Kevin

Reply

ceracm *says:*
February 28, 2016 at 3:10 am

My guess is that the application doesn't have the correct IP address and port for MongoDB. Look at spring-boot-restful-service/docker/set-env.sh . You need to set SPRING_DATA_MONGODB_URI

Reply

russomi *says:*
April 2, 2016 at 4:03 pm

Reblogged this on russomi.

Reply