Еще 3 статьи по Spring Cloud Config Server

**Настраиваем сервер:**

Мы должны использовать аннотацию [@EnableConfigServer](http://twitter.com/EnableConfigServer) в основном классе

[*@EnableConfigServer*](http://twitter.com/EnableConfigServer)[*@SpringBootApplicatio*](http://twitter.com/SpringBootApplicatio)*n  
public class ServerApplication {*

*public static void main(String[] args) {  
SpringApplication.run(ServerApplication.class, args);  
}*

*}*

В файле **application.yml** нам нужно выбрать порт, имя и локальный URL-адрес git для сервера конфигурации Spring Cloud.

Файл **application.yml**

*spring.application.name: config\_server  
  
server:  
port: 8888  
, где spring.application.name: config\_server — это имя вашего сервиса, который будет работать на порту 8888.*

Чтобы брать данные например из удаленного git репозитория, в файле **applicatiom.properties** необходимо добавить

*spring.cloud.config.server.git.uri = https://*[*github.com/lalala*](http://github.com/simplyi/PhotoAppConfigData)

*spring.cloud.config.server.git.username = USER NAME HERE*

*spring.cloud.config.server.git.password = PASSWORD HERE*

*spring.cloud.config.server.git.clone-on-start=true*

Файл **applicatiom.properties**является файлом свойст по умолчанию.

**Настройка клиента:**

Для этого в основном классе поставим аннотацию **@EnableConfigClient**

*@EnableConfigClient*[*@SpringBootApplicatio*](http://twitter.com/SpringBootApplicatio)*n  
public class ServerApplication {*

*public static void main(String[] args) {  
SpringApplication.run(ServerApplication.class, args);  
}*

*}*

Создадим простейший REST контроллер

[*@Component*](http://twitter.com/Component)[*@RestController*](http://twitter.com/RestController)[*@RefreshScope*](http://twitter.com/RefreshScope) *public class ServiceInstanceRestController {*

[*@Autowired*](http://twitter.com/Autowired) *private DiscoveryClient discoveryClient;*

[*@RequestMapping*](http://twitter.com/RequestMapping)*(“/service-instances/{applicationName}”)  
public List<ServiceInstance> serviceInstancesByApplicationName(*[*@PathVariable*](http://twitter.com/PathVariable)*String applicationName) {  
return this.discoveryClient.getInstances(applicationName);  
}*

[*@RequestMapping*](http://twitter.com/RequestMapping)*(“/hello”)  
public String getHelloWorld() {  
return “Hello world!”;  
}*

*}*

или вот такой (на ваше усмотрение)

[*@Component*](http://twitter.com/Component)[*@RefreshScope*](http://twitter.com/RefreshScope)[*@RestController*](http://twitter.com/RestController) *public class ScheduleTaskController {*

[*@Value*](http://twitter.com/Value)*(“${myname}”)  
private String name;*

[*@RequestMapping*](http://twitter.com/RequestMapping)*(“/name”)  
String getValue() {  
return this.name;  
}*

*}*

В файле **application.yml** вам нужно указать порт, имя:

*spring:  
application:  
name: config\_client  
cloud:  
config:  
label: latest*

*server:  
port: 8083*

Нам также понадобиться для примера создать локальный гит репозиторий.

Но вам ничто не мешает сделать удаленный git репозиторий и хранить там какие-то данные. Но для интереса мы сделаем локальный репозиторий.

В консоли (все действия я производил на Ubuntu)

*git init*

Создайте в этом репозитории файл **application.properties** и в нем эту строку:

*myname=helloWorld*

Следующий шаг (в консоли):

*git commit -m “firstcommit”*

Также настройте файл **bootstrap.properties** на клиенте (если его нет, создайте):

*# N.B. this is the default:****spring.cloud.config.uri=***[***http://localhost:8888***](http://localhost:8888/) *# For local GIT repo:****spring.cloud.config.server.git.uri=/home/tmp/serverrepo*** *# For File System:  
# spring.profiles.active=native  
# spring.cloud.config.server.native.searchLocations=/tmp/config-server*

Мы обычно указываем детали конфигурации в файле **application.properties** или **application.yml**. На этот раз, в дополнение к уже существующему файлу свойств нам нужно создать новый файл **bootstrap.properties**. Этот файл будет загружен **до** загрузки файла **application.properties** и **до** создания **Java-бинов** при запуске приложения.

Указание как минимум двух первых свойств в файле **bootstrap.properties** позволит нашим компонентам bean инициализироваться со свойствами, загруженными из файла (локальный git репозиторий).

**Но** если вы хотите сделать наоборот (указать файл свойств на стороне сервера, а не на стороне клиента, как это сделали мы сейчас), то на стороне клиента вам надо будет указать такие свойства:

*spring.cloud.config.uri=http://localhost:8012*

*spring.cloud.config.name=config\_server*

Указание этих свойств на стороне клиента в файле **bootstrap.properties** позволит нашим компонентам Bean инициализироваться со свойствами, загруженными с сервера Spring Cloud Config.

А на стороне сервера указать нечто подобное:

*server.port=8010*

*spring.application.name=config\_server*

*spring.cloud.config.server.git.uri = https://github.com/lalala*

*spring.cloud.config.server.git.username = YOUR USER NAME*

*spring.cloud.config.server.git.password = YOUR PASSWORD*

*spring.cloud.config.server.git.clone-on-start=true*

В этом случае все файлы свойств будут браться из сервера из удаленного git репозитория

**Тестируем:**

Запустите сервер и клиент.  
Запустите эту команду из консоли (172.17.0.1 — это IP-адрес моего ПК):

*curl 172.17.0.1:8888/config\_server/myname*

Затем создайте того же второго клиента и попробуйте получить доступ ко второму клиенту с первого клиента (из консоли):

*curl localhost:8888/service-instances/config\_client  
или  
curl localhost:8888/service-instances/config\_client\_2  
  
или*

*curl localhost:8888/name*

Вы получите информацию из файла в локальном репозитории:

*/home/tmp/serverrepo*

а именно то, что мы указали в файле:

*helloWorld*

Краткое введение в конфигурацию Spring Cloud

* [Spring Security](https://www.codeflow.site/ru/tag/Spring%20Security) [Spring Cloud](https://www.codeflow.site/ru/tag/Spring%20Cloud)

[Facebook](https://www.codeflow.site/#facebook)[Tumblr](https://www.codeflow.site/#tumblr)[Pinterest](https://www.codeflow.site/#pinterest)[Pocket](https://www.codeflow.site/#pocket)[Evernote](https://www.codeflow.site/#evernote)[Twitter](https://www.codeflow.site/#twitter)[Line](https://www.codeflow.site/#line)[Email](https://www.codeflow.site/#email)[Reddit](https://www.codeflow.site/#reddit)[Digg](https://www.codeflow.site/#digg)[VK](https://www.codeflow.site/#vk)[Ресурс](https://www.addtoany.com/share#url=https%3A%2F%2Fwww.codeflow.site%2Fru%2Farticle%2Fspring-cloud-configuration&title=%D0%9A%D1%80%D0%B0%D1%82%D0%BA%D0%BE%D0%B5%20%D0%B2%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5%20%D0%B2%20%D0%BA%D0%BE%D0%BD%D1%84%D0%B8%D0%B3%D1%83%D1%80%D0%B0%D1%86%D0%)

**1. Обзор**

* *Spring Cloud Config*\*\* - это клиент-серверный подход Spring для хранения и обслуживания распределенных конфигураций в различных приложениях и средах.

Это хранилище конфигурации идеально версионировано под управлением версией *Git*и может быть изменено во время выполнения приложения. Хотя он очень хорошо подходит для приложений Spring, использующих все поддерживаемые форматы файлов конфигурации вместе с такими конструкциями, как *Environment*, *[PropertySource или @Value](https://www.codeflow.site/properties-with-spring)*, он может использоваться в любой среде, в которой работает любой язык программирования.

В этой статье мы рассмотрим пример настройки сервера конфигурации с поддержкой *Git*, его использования на простом сервере приложений *REST*и настройки безопасной среды, включая зашифрованные значения свойств.

**2. Настройка проекта и зависимости**

Чтобы подготовиться к написанию кода, мы сначала создадим два новых проекта *Maven*. Проект сервера основан на [*https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-cloud-config-*](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-cloud-config-)*модуль сервера% 22[spring-cloud-config-server]*, а также [*https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.boot%22%*](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.boot%22%)*20AND% 20a% 3A% 22spring-boot-starter-security% 22[spring-boot-starter-security]*и ​​[*https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org*](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org)*.springframework.boot% 22% 20AND% 20a% 3A% 22spring-boot-starter-web% 22[spring-boot-starter-web]*начальные пакеты:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

<version>1.1.2.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

<version>1.4.0.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<version>1.4.0.RELEASE</version>

</dependency>

Однако для клиентского проекта нам понадобится только [*https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%*](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%)*22spring-cloud-starter-config% 22[spring-cloud-starter-config]*и ​​[*https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.boot%*](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.boot%)*22% 20AND% 20a% 3A% 22spring-boot-starter-web% 22[spring-boot-starter-web modules]*:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

<version>1.1.2.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<version>1.4.0.RELEASE</version>

</dependency>

**3. Реализация сервера конфигурации**

Основная часть приложения - это класс конфигурации, в частности ссылка:/spring-boot-application-configuration[ *@ SpringBootApplication*], которая включает все необходимые настройки через *auto-configure*annotation *@ EnableConfigServer:*

@SpringBootApplication

@EnableConfigServer

public class ConfigServer {

public static void main(String[]arguments) {

SpringApplication.run(ConfigServer.class, arguments);

}

}

Теперь нам нужно настроить сервер *port*, на котором слушает наш сервер, и *Git*-url, который предоставляет содержимое конфигурации с управлением версиями. Последний может использоваться с такими протоколами, как *http*, *ssh*или простой *file*в локальной файловой системе.

* Совет: \*\* Если вы планируете использовать несколько экземпляров сервера конфигурации, указывающих на один и тот же репозиторий конфигурации, вы можете настроить сервер для клонирования вашего хранилища во временную локальную папку. Но имейте в виду частные репозитории с двухфакторной аутентификацией, с ними трудно обращаться! В таком случае их проще клонировать в локальной файловой системе и работать с копией.

Для настройки *repository-url*также доступны некоторые переменные *placeholder и шаблоны поиска*; но это выходит за рамки нашей статьи. Если вам интересно, официальная документация - хорошее место для начала.

Нам также нужно установить имя пользователя и пароль для *Basic-Authentication*в нашем *application.properties*, чтобы избежать автоматически сгенерированного пароля при каждом перезапуске приложения:

server.port=8888

spring.cloud.config.server.git.uri=ssh://localhost/config-repo

spring.cloud.config.server.git.clone-on-start=true

security.user.name=root

security.user.password=s3cr3t

**4. Git-репозиторий как хранилище настроек**

Чтобы завершить работу нашего сервера, мы должны инициализировать репозиторий *Git*под настроенным URL, создать несколько новых файлов свойств и популяризировать их с некоторыми значениями.

Имя файла конфигурации составлено как обычный Spring *application.properties*, но вместо слова «application» настроенное имя, например используется значение свойства *‘spring.application.name’*клиента, за которым следует тире и активный профиль. Например:

$> git init

$> echo 'user.role=Developer' > config-client-development.properties

$> echo 'user.role=User' > config-client-production.properties

$> git add .

$> git commit -m 'Initial config-client properties'

* Устранение неполадок: \*\* Если вы столкнулись с проблемами аутентификации, связанными с *ssh*, дважды проверьте *~/.ssh/known*hosts *и*~/.ssh/authorized *keys*на вашем сервере ssh!

**5. Запрос конфигурации**

Теперь мы можем запустить наш сервер. API-интерфейс с поддержкой *Git*, предоставляемый нашим сервером, можно запросить по следующим путям:

----/{application}/{profile}[/{label}]/{application}-{profile}.yml/{label}/{application}-{profile}.yml/{application}-{profile}.properties/{label}/{application}-{profile}.properties

----

В котором заполнитель *\ {label}*относится к ветви Git, *\ {application}*- к имени клиентского приложения, а *\ {profile}*- к текущему активному профилю приложения клиента.

Таким образом, мы можем получить конфигурацию для нашего запланированного клиента конфигурации, работающего в профиле разработки в ветви *master*, с помощью:

$> curl http://root:[email protected]:8888/config-client/development/master

**6. Реализация клиента**

Далее, давайте позаботимся о клиенте. Это будет очень простое клиентское приложение, состоящее из контроллера *REST*с одним методом *GET*.

Конфигурация, чтобы получить наш сервер, должна быть помещена в файл ресурсов с именем *bootstrap.application*, потому что этот файл (как следует из названия) будет загружен очень рано, пока приложение запускается:

@SpringBootApplication

@RestController

public class ConfigClient {

@Value("${user.role}")

private String role;

public static void main(String[]args) {

SpringApplication.run(ConfigClient.class, args);

}

@RequestMapping(

value = "/whoami/{username}",

method = RequestMethod.GET,

produces = MediaType.TEXT\_\_PLAIN\_\_VALUE)

public String whoami(@PathVariable("username") String username) {

return String.format("Hello!

You're %s and you'll become a(n) %s...\n", username, role);

}

}

В дополнение к имени приложения мы также помещаем активный профиль и сведения о соединении в наш b *\_ootstrap.properties \_*:

spring.application.name=config-client

spring.profiles.active=development

spring.cloud.config.uri=http://localhost:8888

spring.cloud.config.username=root

spring.cloud.config.password=s3cr3t

Чтобы проверить, правильно ли получена конфигурация с нашего сервера и в методе контроллера введено *role value*, мы просто свернем ее после загрузки клиента:

$> curl http://localhost:8080/whoami/Mr\_\_Pink

Если ответ следующий, наш *Spring Cloud Config Server*и его клиент работают нормально:

Hello! You're Mr\_\_Pink and you'll become a(n) Developer...

**7. Шифрование и дешифрование**

* Требование \*\* : Чтобы использовать криптографически надежные ключи вместе с функциями шифрования и дешифрования Spring, вам понадобятся *'Java Cryptography Extension (JCE) Файлы политики неограниченной юрисдикции'*, установленные в вашем *JVM.*Их можно загрузить, например, с [Oracle](http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html).

Для установки следуйте инструкциям, включенным в загрузку. Некоторые дистрибутивы Linux также предоставляют устанавливаемый пакет через своих менеджеров пакетов.

Поскольку сервер конфигурации поддерживает шифрование и дешифрование значений свойств, вы можете использовать общедоступные репозитории в качестве хранилища для конфиденциальных данных, таких как имена пользователей и пароли. Зашифрованные значения имеют префикс строки *\ {cipher}*и могут быть сгенерированы с помощью REST-вызова пути *‘/encrypt’*, если сервер настроен на использование симметричного ключа или пары ключей.

Также доступна конечная точка для расшифровки. Обе конечные точки принимают путь, содержащий заполнители для имени приложения и его текущего профиля: *‘/\*\* /\ {name}/\ {profile}’*. Это особенно полезно для управления криптографией для каждого клиента. Однако, прежде чем они станут полезными, вы должны настроить криптографический ключ, который мы сделаем в следующем разделе.

* Совет: \*\* Если вы используете curl для вызова API en-/decryption, лучше использовать параметр *– data-urlencode*(вместо *– data/-d*) или установить заголовок Content-Type явно в *'text/plain'*. Это обеспечивает правильную обработку специальных символов, таких как in ’, в зашифрованных значениях.

Если значение не может быть дешифровано автоматически при извлечении из клиента, его *key*переименовывается с самим именем, с префиксом слова «invalid» Это должно предотвратить, например, использование зашифрованного значения в качестве пароля.

* Совет: \*\* При настройке репозитория, содержащего файлы YAML, вы должны заключать в зашифрованные и префиксные значения одинарные кавычки! Со свойствами это не так.

**7.1. Ключевой менеджмент**

По умолчанию сервер конфигурации включен для шифрования значений свойств симметричным или асимметричным способом.

* Чтобы использовать симметричную криптографию \*\* , вам просто нужно установить свойство *‘encrypt.key’*в вашем *application.properties*на выбранный вами секрет *\_\_. В качестве альтернативы вы можете передать переменную среды*ENCRYPT *KEY*.
* Для асимметричной криптографии \*\* вы можете установить для *‘encrypt.key’*строковое значение в кодировке *PEM*или настроить использование *keystore*.

Поскольку для нашего демонстрационного сервера нам нужна среда с высокой степенью защиты, мы выбрали последний вариант и создали новое хранилище ключей, включая пару ключей *RSA*, с Java *keytool*first:

$> keytool -genkeypair -alias config-server-key \

-keyalg RSA -keysize 4096 -sigalg SHA512withRSA \

-dname 'CN=Config Server,OU=Spring Cloud,O=Baeldung' \

-keypass my-k34-s3cr3t -keystore config-server.jks \

-storepass my-s70r3-s3cr3t

После этого мы добавляем созданное хранилище ключей в *application.properties*нашего сервера и перезапускаем его:

encrypt.key-store.location=classpath:/config-server.jks

encrypt.key-store.password=my-s70r3-s3cr3t

encrypt.key-store.alias=config-server-key

encrypt.key-store.secret=my-k34-s3cr3t

В качестве следующего шага мы можем запросить конечную точку шифрования и добавить ответ в качестве значения к конфигурации в нашем репозитории:

$> export PASSWORD=$(curl -X POST --data-urlencode d3v3L \

http://root:[email protected]:8888/encrypt)

$> echo "user.password=$PASSWORD" >> config-client-development.properties

$> git commit -am 'Added encrypted password'

$> curl -X POST http://root:[email protected]:8888/refresh

Чтобы проверить, правильно ли работает наша установка, мы модифицируем класс *ConfigClient*и перезапустим наш клиент:

@SpringBootApplication

@RestController

public class ConfigClient {

...

@Value("${user.password}")

private String password;

...

public String whoami(@PathVariable("username") String username) {

return String.format("Hello!

You're %s and you'll become a(n) %s, " +

"but only if your password is '%s'!\n",

username, role, password);

}

}

Последний запрос к нашему клиенту покажет нам, если значение нашей конфигурации правильно расшифровано:

$> curl http://localhost:8080/whoami/Mr\_\_Pink

Hello! You're Mr\_\_Pink and you'll become a(n) Developer, \

but only if your password is 'd3v3L'!

**7.2. Использование нескольких ключей**

Если вы хотите использовать несколько ключей для шифрования и дешифрования, например: выделенный для каждого обслуживаемого приложения, вы можете добавить другой префикс в форме *\ {name: value}*между префиксом *\ {cipher}*и *BASE64*-закодированное значение свойства.

Конфигурационный сервер распознает префиксы, такие как *\ {secret: my-crypto-secret}*или *\ {key: my-key-alias}*, практически готовые к использованию. Последний вариант требует настроенного хранилища ключей в вашем *application.properties*. В этом хранилище ключей выполняется поиск подходящего псевдонима ключа.

user.password={cipher}{secret:my-499-s3cr3t}AgAMirj1DkQC0WjRv...

user.password={cipher}{key:config-client-key}AgAMirj1DkQC0WjRv...

Для сценариев без хранилища ключей вы должны реализовать *@ Bean*типа *TextEncryptorLocator*, который обрабатывает поиск и возвращает *TextEncryptor*-Object для каждого ключа.

**7.3. Обслуживание зашифрованных свойств**

Если вы хотите отключить криптографию на стороне сервера и локально обрабатывать дешифрование значений свойств, вы можете поместить следующее в *application.properties*вашего сервера:

spring.cloud.config.server.encrypt.enabled=false

Кроме того, вы можете удалить все другие свойства «encrypt. \*\* », Чтобы отключить конечные точки *REST*.

**8. Заключение**

Теперь мы можем создать сервер конфигурации для предоставления набора файлов конфигурации из репозитория *Git*клиентским приложениям.

Есть несколько других вещей, которые вы можете сделать с таким сервером.

Например:

* Служите конфигурации в формате *YAML*или *Properties*вместо \_\_JSON

– *также с разрешенными заполнителями. Что может быть полезно при использовании его в средах, отличных от Spring, где конфигурация напрямую не сопоставлена ​​с*PropertySource\_\_.

* Служите простыми текстовыми файлами конфигурации - в свою очередь, опционально с

решенные заполнители. Это может быть полезно, например, для предоставления зависящей от среды конфигурации регистрации.

* Вставить сервер конфигурации в приложение, где он настраивает

сам из репозитория *Git*, вместо того, чтобы работать как отдельное приложение, обслуживающее клиентов. Поэтому некоторые свойства начальной загрузки должны быть установлены и/или аннотация *@ EnableConfigServer*должна быть удалена, что зависит от варианта использования.

* Сделать сервер конфигурации доступным в сервисе Spring Netflix Eureka

обнаружение и включение автоматического обнаружения сервера в клиентах конфигурации. Это становится важным, если у сервера нет фиксированного местоположения или он перемещается в своем местоположении.

Введение в Spring Cloud Config Server

[I.SUKACH](https://springgears.com/author/i-sukach/)[22 АПРЕЛЯ, 2019](https://springgears.com/cloud-config-server-introduction/)[ДЕМО](https://springgears.com/category/%d0%b4%d0%b5%d0%bc%d0%be/)[0 КОММЕНТАРИЕВ](https://springgears.com/cloud-config-server-introduction/#respond)

После длительного перерыва мне, наконец, удалось вернуться к разбору деталей Spring. Эта статья станет первой в новом цикле, посвящённом проекту Spring Cloud. Постепенно мы изучим его основные компоненты. А начнём погружение с разбора технологии, тесно связанной с темой [прошлой статьи](https://springgears.com/spring-boot-external-configuration/) — **Spring Cloud Config Server**.

Что такое Spring Cloud?

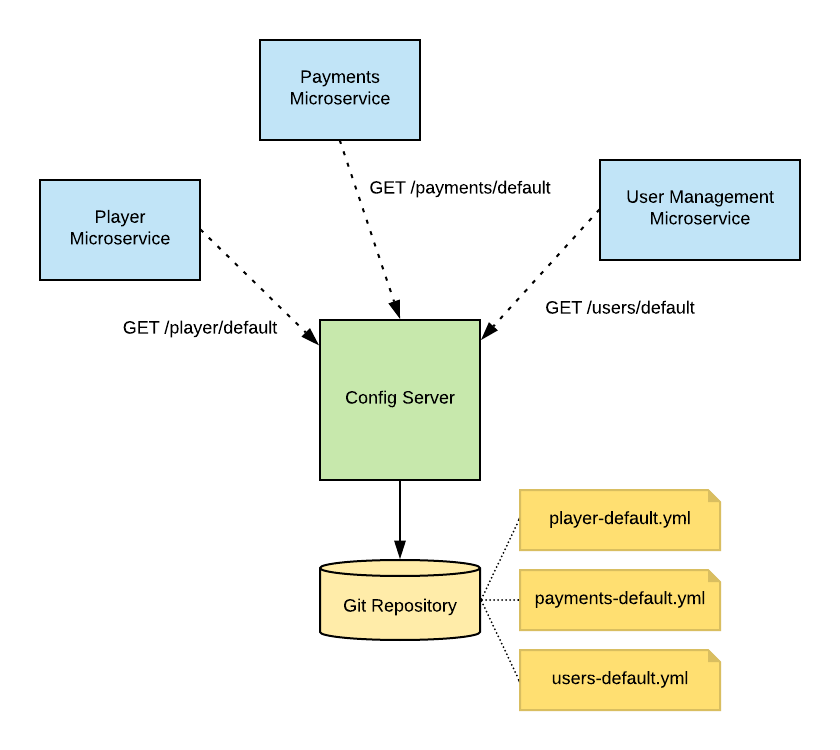
Проект Spring Cloud представляет собой набор технологий, призванных помочь разработчику в создании распределенных систем. Каждая из технологий позволяет решать конкретную задачу. К таким задачам относятся:

* Распределённая конфигурация
* Обнаружение сервисов
* Маршрутизация запросов
* Организация вызовов между сервисами
* Балансировка нагрузки
* и многие другие

Начнём с первого пункта списка и поговорим о распределенной конфигурации.

Отправляем конфигурацию в облако

Основная идея такая: мы заводим отдельный сервер, который будет выдавать настройки для всех приложений в нашей архитектуре. Каждое приложение будет обращаться к этому серверу за своими настройками на этапе создания собственного контекста. Сами же настройки хранятся либо в файловой системе, либо, что предпочтительнее, в системе контроля версий.



Этот подход — альтернатива разобранному в предыдущей статье способу инициализации с помощью параметров командной строки и переменных среды. Плюсом является то, что конфигурация для многочисленных сервисов будет хранится в одном месте. Недостаток же в усложненной конфигурации и необходимости обеспечения безопасного хранения настроек.

Настраиваем сервер

Для создания конфигурационного сервера добавляем в обычное Spring Boot приложение зависимости Spring Cloud:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | dependencies {     implementation 'org.springframework.cloud:spring-cloud-config-server'  }    dependencyManagement {     imports {        mavenBom "org.springframework.cloud:spring-cloud-dependencies:${springCloudVersion}"     }  } |

Затем добавляем конфигурацию системы контроля версий, где будут храниться настройки других приложений, в application.yml:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | server:    port: 8888    spring:    cloud:      config:        server:          git:            uri: https://github.com/isukach/spring-cloud-config-file |

В этом репозитории будет лежать файл **player-default.yml**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | db:    url: jdbc:postgresql://localhost:5432/player    player:    max-playlists: 100 |

И обязательно добавляем аннотацию @EnableConfigServer:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @EnableConfigServer  @SpringBootApplication  public class ConfigServerApplication {    public static void main(String[] args) {  SpringApplication.run(ConfigServerApplication.class, args);  }    } |

Теперь запускаем сервер и проверяем выдачу. Обратите внимание, что в пути фигурирует названия приложения (**player**), для которого получаются настройки, а также его профиль (**default**):

*GET http://localhost:8888/****player****/****default***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | {       "name": "player",       "profiles": [           "default"       ],       "label": null,       "version": "0166ffd5cc09226dae5d5ccd98f4417f9336b9aa",       "state": null,       "propertySources": [           {               "name": "https://github.com/isukach/spring-cloud-config-file/player-default.yml",               "source": {                   "db.url": "jdbc:postgresql://localhost:5432/player",                   "player.max-playlists": 100               }           }       ]  } |

Настраиваем клиент

Для того чтобы клиент начал обращаться к конфигурационному серверу за настройками во время создания контекста, нам достаточно добавить в проект зависимость:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | dependencies {  implementation 'org.springframework.cloud:spring-cloud-starter-config'  }    dependencyManagement {  imports {  mavenBom "org.springframework.cloud:spring-cloud-dependencies:${springCloudVersion}"  }  } |

А также конфигурацию имени приложения и пути к серверу в application.yml:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | spring:    application:      name: player    cloud:      config:        uri: http://localhost:8888 |

Всё! Теперь в контексте клиентского приложения автоматически появятся настройки из файла, хранящегося в Git-репозитории. Проверяем с помощью ApplicationListener:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | @Slf4j  @Component  public class ApplicationBootstrapListener implements ApplicationListener<ContextRefreshedEvent> {        @Value("${db.url}")      private String dbUrl;        @Value("${player.max-playlists}")      private String maxPlaylists;        @Override      public void onApplicationEvent(ContextRefreshedEvent event) {          log.info("Database URL: {}", dbUrl);            log.info("Max playlists: {}", maxPlaylists);      }  } |

Database URL: jdbc:postgresql://localhost:5432/player  
Max playlists: 100

Разделяем настройки окружений

Так как настройки для разных окружений (local, dev, prod) будут отличаться, в конфигурации мы будем разделять их на уровне файлов в Git. Spring Cloud Config Server использует механизм профилей (Spring Profiles) чтобы понять, из какого файла брать настройки.

Например, если при запуске приложения player не указать профиль, то сервер будет искать файл player-**default**.yml (как в примере выше). Если же указать профиль, скажем, **dev**, то и настройки нужно класть в файл с именем player-**dev**.yml.

Добавим в Git файл player-dev.yml:

|  |  |
| --- | --- |
| 1  2  3  4  5 | db:    url: jdbc:postgresql://dev-host:5432/player    player:    max-playlists: 999999 |

Отправляем запрос на конфигурационный сервер

*GET http://localhost:8888/****player****/****dev***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | {      "name": "player",      "profiles": [          "dev"      ],      "label": null,      "version": "85fe0679976f44a69931d5db9c47b7644bafcd6a",      "state": null,      "propertySources": [          {              "name": "https://github.com/isukach/spring-cloud-config-file/player-dev.yml",              "source": {                  "db.url": "jdbc:postgresql://dev-host:5432/player",                  "player.max-playlists": 999999              }          }      ]  } |

Также, если запустить клиентское приложение с параметром  
-Dspring.profiles.active=dev

Database URL: jdbc:postgresql://dev-host:5432/player  
Max playlists: 999999

Что дальше?

Итак, у нас есть работающий конфиг-сервер. Однако, в этом виде его конфигурация не совсем подходит для боевого развертывания, так как не хватает некоторых настроек, выходящих за рамки этой статьи:

1. Конфигурация приватного git-репозитория и его защита с помощью логина/пароля или пары public/private ключей. Добавление настроек приватности в конфиг-сервер. Почитать можно [здесь](https://cloud.spring.io/spring-cloud-config/multi/multi__spring_cloud_config_server.html#_authentication).
2. Защита эндпойнтов сервера с помощью одного из механизмов аутентификации (Basic, Oauth2 и др.). Этот момент описан по [ссылке](https://cloud.spring.io/spring-cloud-config/multi/multi__spring_cloud_config_server.html#_security).