Spring Cloud - *CONFIG*

Spring Cloud - Config

- Spring Cloud Config, dağıtılmış bir sistemde harici-dış yapılandırma için sunucu ve istemci tarafı desteği sağlar. Config Server ile tüm ortamlardaki uygulamalar için harici-dış özellikleri yönetmek için merkezi bir yeriniz olur.
- Hem istemci hem de sunucu üzerindeki kavramlar, Spring **Environment** ve **PropertySource** soyutlamalarıyla aynı şekilde eşleşir, bu sebeple Spring uygulamalarıyla çok iyi uyum sağlarlar, ancak herhangi bir dilde çalışan herhangi bir uygulama ile kullanılabilir.
- Bir uygulama geliştirmeden teste ve üretime dağıtım hattından geçerken, bu ortamlar arasındaki yapılandırmayı yönetebilir ve uygulamaların geçiş yaptıklarında çalıştırmak için ihtiyaç duydukları her şeye sahip olduğundan emin olabilirsiniz.
- Sunucu depolama arka ucunun varsayılan uygulaması git'i kullanır, böylece yapılandırma ortamlarının etiketli sürümlerini kolayca destekler ve içeriği yönetmek için çok çeşitli araçlara erişilebilir. Alternatif uygulamalar eklemek ve bunları Spring konfigürasyonu ile eklemek kolaydır.

Spring Cloud – Config *Server(Sunucu)* Özellikleri

- Harici-dış yapılandırma için HTTP, kaynak tabanlı API'dir (ad-değer çiftleri veya eşdeğer YAML içeriği)
- Özellik değerlerini şifreleme ve şifresini çözer (simetrik veya asimetrik)
- **@EnableConfigServer** kullanılarak Spring Boot uygulamasına kolayca yerleştirilebilir

Spring Cloud – Config *Client(İstemci)*Özellikleri (Spring uygulamaları için)

- Config Server'a bağlanır ve Spring **Environment**'ı uzak özellik kaynaklarıyla başlatır
- Özellik değerlerini şifreleme ve şifresini çözme (simetrik veya asimetrik)

- Spring Boot Actuator ve Spring Config Client «sınıf yolunda» olduğu sürece, herhangi bir Spring Boot uygulaması, **spring.cloud.config.uri**'nin varsayılan değeri olan **http://localhost:8888** üzerindeki bir yapılandırma sunucusuyla iletişim kurmaya çalışır.
- Bu varsayılanı değiştirmek isterseniz, spring.cloud.config.uri'yi bootstrap.[yml | properties]'de ayarlayabilirsiniz veya sistem özellikleri veya ortam değişkenleri aracılığıyla ayarlayabilirsiniz.

```java
@Configuration
@EnableAutoConfiguration
@RestController
public class Application {

  @Value("${config.name}")
  String name = "World";

  @RequestMapping("/")
  public String home() {
    return "Hello " + name;
  }

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

}
```

Örnekteki **config.name** değeri (veya normal Spring Boot yöntemiyle bağladığınız diğer değerler) yerel konfigürasyondan veya uzak Config Server'dan gelebilir. Config Server varsayılan olarak öncelikli olacaktır. Bunu görmek için uygulamadaki **/env** uç noktasına bakın ve **configServer** özellik kaynaklarına bakın.

Kendi serverinizi(sunucunuzu) çalıştırmak için **spring-cloud-config-server** dependency ve **@EnableConfigServer**'ı kullanın. Eğer **spring.config.name=configserver**'ı ayarlarsanız, uygulama 8888 numaralı bağlantı noktasında çalışır ve bir örnek repository'den(havuzdan) veri sunar. Kendi ihtiyaçlarınıza yönelik yapılandırma verilerini bulmak için bir **spring.cloud.config.server.git.uri**'ye ihtiyacınız var(varsayılan olarak bir git deposunun konumudur, yerel bir **file:..** URL'si olabilir).

LEARN-ÖĞREN

Hızlı Başlangıç

Hızlı başlangıçta, Spring Cloud Config Server'in hem server(sunucu) hem de client'i(alıcı) kullanma süreci açıklanmaktadır.

İlk önce server'i aşağıdaki gibi başlatın:

```
$ cd spring-cloud-config-server
$ ../mvnw spring-boot:run
```

Sunucu bir Spring Boot uygulamasıdır, yani isterseniz onu IDE'nizden çalıştırabilirsiniz (ana sınıf **ConfigServerApplication**'dır).

Ardından, bir client'i aşağıdaki gibi deneyin:

```
$ curl localhost:8888/foo/development
{
   "name": "foo",
   "profiles": [
     "development"
  ]
....
   "propertySources": [
      {
       "name": "https://github.com/spring-cloud-samples/config-repo/foo-development.properties",
       "source": {
          "bar": "spam",
          "foo": "from foo development"
       }
   },
   {
       "name": "https://github.com/spring-cloud-samples/config-repo/foo.properties",
       "source": {
           "foo": "from foo props",
           "democonfigclient.message": "hello spring io"
    }
},
....
```

Özellik kaynaklarını bulmak için varsayılan strateji, bir git deposunu klonlamak (**spring.cloud.config.server.git.uri**'de) ve bunu mini bir SpringApplication başlatmak için kullanmaktır. Mini uygulamanın Environment'i, özellik kaynaklarını numaralandırmak ve bunları bir JSON uç noktasında yayınlamak için kullanılır.

HTTP hizmeti aşağıdaki biçimde kaynaklara sahiptir:

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

Örneğin:

```
curl localhost:8888/foo/development
curl localhost:8888/foo/development/master
curl localhost:8888/foo/development,db/master
curl localhost:8888/foo-development.yml
curl localhost:8888/foo-db.properties
curl localhost:8888/master/foo-db.properties
```

application'ın(uygulamanın) **SpringApplication**'da **spring.config.name** olarak enjekte edildiği (normalde bir Spring Boot uygulamasında application nedir), **profil** aktif bir profildir (veya virgülle ayrılmış özellikler listesidir) ve **label** isteğe bağlı bir git etiketidir (varsayılan olarak **master**'dır).

Spring Cloud Config Server, çeşitli kaynaklardan uzak client'ler(alıcılar) için yapılandırmayı çeker. Aşağıdaki örnek, bir git deposundan (sağlanması gereken) yapılandırma alır, aşağıdaki örnekte gösterildiği gibi:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
```

Diğer kaynaklar, herhangi bir JDBC uyumlu veritabanı, Subversion, Hashicorp Vault, Credhub ve yerel dosya sistemleridir.

Client(Alıcı) Tarafı Kullanımı

<> Bu özellikleri bir uygulamada kullanmak için, onu **spring-cloud-config-client**'a bağlı bir Spring Boot uygulaması olarak oluşturabilirsiniz (örneğin, config-client veya örnek uygulama için test senaryolarına bakın).

<> Dependency'i eklemenin en uygun yolu, Spring Boot starter **org.springframework.cloud:spring-cloud-starter-config**'dir.

<> Ayrıca Maven kullanıcıları için bir parent pom ve BOM (**spring-cloud-starter-parent**) ve Gradle ve Spring CLI kullanıcıları için bir Spring IO sürüm yönetimi özellikleri dosyası vardır.

Aşağıdaki örnek, tipik bir Maven yapılandırmasını göstermektedir:

pom.xml

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>{spring-boot-docs-version}</version>
    <relativePath /> <!-- lookup parent from repository -->
 </parent>

<dependencyManagement>
                <dependencies>
                                <dependency>
                                                <groupId>org.springframework.cloud</groupId>
                                                <artifactId>spring-cloud-dependencies</artifactId>
                                                <version>{spring-cloud-version}</version>
                                                <type>pom</type>
                                                <scope>import</scope>
                                </dependency>
                </dependencies>
</dependencyManagement>

<dependencies>
<dependency>
                                <groupId>org.springframework.cloud</groupId>
                                <artifactId>spring-cloud-starter-
config</artifactId>
                </dependency>
                <dependency>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-starter-test</artifactId>
                                <scope>test</scope>
                </dependency>
</dependencies>

<build>
                <plugins>
        <plugin>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
                </plugins>
</build>

  <!-- repositories also needed for snapshots and milestones -->
```

Artık aşağıdaki HTTP sunucusu gibi standart bir Spring Boot uygulaması oluşturabilirsiniz:

```
@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello World!";
    }

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

}
```

Bu HTTP server çalıştığında, 8888 numaralı bağlantı noktasındaki varsayılan yerel config server(çalışıyorsa) harici yapılandırmayı alır. Başlangıç davranışını değiştirmek için, **application[.properties | .yml]**'i gösterildiği gibi kullanarak config server'in konumunu değiştirebilirsiniz. Aşağıdaki örnekteki gibi:

```
spring.config.import=optional:configserver:http://myconfigserver.com
```

Varsayılan olarak, herhangi bir uygulama adı ayarlanmadıysa **application** kullanılacaktır. Adı değiştirmek için **application[.properties | .yml]** dosyasına aşağıdaki özellik eklenebilir:

```
spring.application.name: myapp
```


${spring.application.name} özelliğini ayarlarken, doğru özellik kaynağını çözen sorunları önlemek için uygulama adınızın önüne ayrılmış application- kelimesi eklemeyin.

Config Server özellikleri, aşağıdaki örnekte gösterildiği gibi **/env** uç noktasında yüksek öncelikli bir özellik kaynağı olarak görünür.

```
$ curl localhost:8080/env
{
  "activeProfiles": [],
  {
    "name": "servletContextInitParams",
    "properties": {}
  },
  {
    "name": "configserver:https://github.com/spring-cloud-samples/config-repo/foo.properties",
    "properties": {
      "foo": {
        "value": "bar",
        "origin": "Config Server https://github.com/spring-cloud-samples/config-repo/foo.properties:2:12"
      }
    }
  },
  ...
}
```

configserver:<URL of remote repository>/<file name> adlı bir özellik kaynağı(property source), **bar** değerine sahip **foo** özelliğini içerir.

Özellik kaynağı(property source) adındaki URL, config server URL'si değil, git deposudur.

Spring Cloud Config Client kullanıyorsanız, Config Server'a bağlanmak için **spring.config.import** özelliğini ayarlamanız gerekir. Bununla ilgili daha fazla bilgiyi Spring Cloud Config Referans Kılavuzunda okuyabilirsiniz.

Spring Cloud Config Server

Spring Cloud Config Server, harici yapılandırma (name-value çiftleri veya eşdeğer YAML içeriği) için HTTP kaynağı tabanlı bir API sağlar. Sunucu, **@EnableConfigServer** anatasyonu kullanılarak Spring Boot uygulamasına yerleştirilebilir. Sonuç olarak, aşağıdaki uygulama bir yapılandırma sunucusudur:

ConfigServer.java

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServer {
  public static void main(String[] args) {
    SpringApplication.run(ConfigServer.class, args);
  }
}
```

Tüm Spring Boot uygulamaları gibi, varsayılan olarak 8080 numaralı bağlantı noktasında çalışır, ancak onu çeşitli şekillerde daha geleneksel bağlantı noktası 8888'e değiştirebilirsiniz. Varsayılan bir yapılandırma repository'si (havuzu) de ayarlamanın en kolayı, onu **spring.config.name=configserver** ile başlatmaktır (Config Server jar'da bir configserver.yml vardır).

Git deponuz için yerel dosya sistemini kullanmak yalnızca test amaçlıdır. Yapılandırma havuzlarınızı üretimde barındırmak için bir server kullanmalısınız.

Yalnızca metin dosyalarını içinde tutarsanız, yapılandırma repository'inizin ilk klonu hızlı ve verimli olabilir. İkili dosyaları, özellikle büyük dosyaları depolarsanız, ilk yapılandırma isteğinde gecikmeler yaşayabilir veya serverde yetersiz bellek hatalarıyla karşılaşabilirsiniz.

Bir diğeri, aşağıdaki örnekte gösterildiği gibi kendi application.properties'inizi kullanmaktır:

application.properties

```
server.port: 8888
spring.cloud.config.server.git.uri: file://${user.home}/config-repo
```

Burada ${user.home}/config-repo, YAML ve özellikler dosyalarını içeren bir git repository'sidir(deposudur).

Windows'ta, bir sürücü önekiyle (örneğin, file:///${user.home}/config-repo) mutlak ise dosya URL'sinde fazladan bir "**/**" işaretine ihtiyacınız vardır.

Aşağıdaki liste, önceki örnekte git deposu oluşturmak için bir tarif gösterir:

```
$ cd $HOME
$ mkdir config-repo
$ cd config-repo
$ git init .
$ echo info.foo: bar > application.properties
$ git add -A .
$ git commit -m "Add application.properties"
```

Environment Repository(Ortam Deposu)

Config Server için yapılandırma verilerini nerede saklamanız gerekir? Bu davranışı yöneten strateji, **Environment** nesnelerine hizmet veren **EnvironmentRepository**'dir. Bu Environment, Spring Environmentteki alanının sığ bir kopyasıdır (ana özellik olarak propertySources dahil). **Environment** kaynakları üç değişkenle parametrelendirilir:

- **{application}**, client(alıcı) tarafında **spring.application.name** ile eşleşen.
- **{profile}**, client'te **spring.profiles.active** ile eşleşen(virgülle ayrılmış liste).
- **{label}**, "sürümlendirilmiş" bir yapılandırma dosyası kümesini etiketleyen bir server tarafı özelliğidir.

Repository implementasyonları genellikle bir Spring Boot uygulaması gibi davranır, konfigürasyon dosyalarını **{application}** parametresine eşit bir **spring.config.name** ve **{profiles}** parametresine eşit **spring.profiles.active**'den yükler. Profiller için öncelik kuralları da normal Spring Boot uygulamasındakilerle aynıdır: Aktif profiller varsayılanlara göre önceliklidir ve birden fazla profil varsa sonuncusu kazanır (Bir **map**' e giriş eklemeye benzer şekilde).

Aşağıdaki örnek client application'u bu bootsrap yapılandırmasına sahiptir:

```
spring:
  application:
    name: foo
  profiles:
    active: dev,mysql
```

(Bir Spring Boot uygulamasında her zaman olduğu gibi, bu özellikler ayrıca ortam değişkenleri veya komut satırı argümanları tarafından da ayarlanabilir).

Repository dosya tabanlıysa(file-based), server **application.yml**'den (tüm client'ler arasında paylaşılan) ve **foo.yml**'den (öncelik foo.yml olmak üzere) bir **Environment**(ortam) oluşturur. YAML dosyalarının içinde Spring profillerine işaret eden belgeler varsa, bunlar daha yüksek önceliğe sahip olarak uygulanır (listelenen profillerin sırasına göre). Profile özgü YAML (veya özellikler) dosyaları varsa, bunlar da varsayılanlardan daha yüksek önceliğe sahip olarak uygulanır. Daha yüksek öncelik, Environment'de(ortamda) daha önce listelenen bir PropertySource'a dönüşür. (Aynı kurallar bağımsız bir Spring Boot uygulamasında da geçerlidir.)

Spring.cloud.config.server.accept-empty ayarını **false** olarak ayarlayabilirsiniz, böylece uygulama bulunamazsa Sunucu bir HTTP 404 durumu döndürür. Varsayılan olarak, bu bayrak ***true*** olarak ayarlanır.

Git Backend

EnvironmentRepository'nin varsayılan uygulaması, yükseltmeleri ve fiziksel ortamları yönetmek ve değişiklikleri denetlemek için çok uygun olan bir Git backend kullanır. Repository'nin konumunu değiştirmek için, Config Server'da (örneğin application.yml'de) **spring.cloud.config.server.git.uri** konfigürasyon özelliğini ayarlayabilirsiniz. Bir **file:** öneki ile ayarlarsanız, sunucu olmadan hızlı ve kolay bir şekilde başlayabilmeniz için yerel bir repository'den(depo) çalışması gerekir. Ancak, bu durumda, server yerel depoyu klonlamadan doğrudan yerel depoda çalışır (açık olup olmaması önemli değildir, çünkü Config Server "uzak" depoda-repository'de hiçbir zaman değişiklik yapmaz). Config Server'ı ölçeklendirmek ve yüksek düzeyde kullanılabilir hale getirmek için, sunucunun tüm örneklerinin aynı depoya işaret etmesi gerekir, böylece yalnızca paylaşılan bir dosya sistemi çalışır. Bu durumda bile, paylaşılan bir dosya sistemi deposu için **ssh:** protokolünü kullanmak daha iyidir, böylece sunucu onu klonlayabilir ve bir yerel çalışma kopyasını önbellek olarak kullanabilir.

Bu repository implementasyonu, HTTP kaynağının **{label}** parametresini bir git etiketine (taahhüt kimliği, dal adı veya etiket) eşler. Git dalı veya etiket adı bir eğik çizgi (**/**) içeriyorsa, HTTP URL'sindeki etiket bunun yerine özel dize (**_**) ile belirtilmelidir (diğer URL yollarıyla belirsizliği önlemek için).
Örneğin, etiket **foo/bar** ise, eğik çizginin değiştirilmesi şu etiketle sonuçlanır: **foo(_)bar**. Özel dizenin (_) eklenmesi **{application}** parametresine de uygulanabilir. curl gibi bir komut satırı client'i kullanıyorsanız, URL'deki parantezlere dikkat edin — bunları kabuktan tek tırnak (**''**) ile kaçırmalısınız.

Skipping SSL Certificate Validation (SSL Sertifika Doğrulamasının Atlanması)

Configuration server'in Git serverin SSL sertifikasını doğrulaması, **git.skipSslValidation** özelliği **true** olarak ayarlanarak devre dışı bırakılabilir (varsayılan **false**).

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://example.com/my/repo
          skipSslValidation: true
```

Setting HTTP Connection Timeout (HTTP Bağlantı Zaman Aşımını Ayarlama)

Yapılandırma sunucusunun bir HTTP bağlantısı almak için bekleyeceği süreyi saniye cinsinden yapılandırabilirsiniz.

git.timeout özelliğini kullanın.

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://example.com/my/repo
          timeout: 4
```

Placeholders in Git URI (Git URI'deki yer tutucular)

Spring Cloud Config Server, **{application}** ve **{profile}** (ve gerekirse **{label}**) için yer tutuculara sahip bir git deposu URL'sini destekler, ancak etiketin yine de git etiketi olarak uygulandığını unutmayın. Böylece, aşağıdakine benzer bir yapı kullanarak "***uygulama başına bir repository***" politikasını destekleyebilirsiniz:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/myorg/{application}
```

Benzer bir kalıp kullanarak **{profile}** ile "***profil başına bir repository***" politikasını da destekleyebilirsiniz.

Ek olarak, **{application}** parametrelerinizde "(_)" özel dizesini kullanmak, aşağıdaki örnekte gösterildiği gibi birden çok kuruluş için desteği etkinleştirebilir:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/{application}
```

{application}, istek sırasında şu biçimde sağlanır:
organizasyon(_)application

Pattern Matching and Multiple Repositories (Model Eşleştirme ve Çoklu Depolar)

Spring Cloud Config, uygulama ve profil adında kalıp eşleştirme ile daha karmaşık gereksinimler için destek de içerir. Kalıp formatı, aşağıdaki örnekte gösterildiği gibi, joker karakterli **{application}/{profile}** adlarının virgülle ayrılmış bir listesidir (joker karakterle başlayan bir kalıbın alıntılanması gerekebileceğini unutmayın):

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          repos:
            simple: https://github.com/simple/config-repo
            special:
              pattern: special*/dev*,*special*/dev*
              uri: https://github.com/special/config-repo
            local:
              pattern: local*
              uri: file:/home/configsvc/config-repo
```

Eğer **{application}/{profile}** herhangi bir modelle eşleşmiyorsa, **spring.cloud.config.server.git.uri** altında tanımlanan varsayılan URI'yi kullanır. Yukarıdaki örnekte, "simple" veri repository için kalıp **simple/***'dir (tüm profillerde yalnızca **simple** adlı bir uygulamayla eşleşir). " local" repository, tüm profillerde **local** ile başlayan tüm uygulama adlarıyla eşleşir (**/*** soneki, profil eşleştiricisi olmayan herhangi bir kalıba otomatik olarak eklenir).

"Simple" örnekte kullanılan "tek satırlı" kısayol, yalnızca ayarlanacak tek özellik URI ise kullanılabilir. Başka bir şey (kimlik bilgileri, kalıp-model vb.) ayarlamanız gerekirse, tam formu kullanmanız gerekir.

Repository’deki **pattern**(desen-model-kalıp) özelliği aslında bir dizidir, bu nedenle birden çok desene bağlanmak için bir YAML dizisi (veya özellikler dosyalarındaki **[0]**, **[1]** vb. sonekleri) kullanabilirsiniz.
Aşağıdaki örnekte gösterildiği gibi uygulamaları birden çok profille çalıştıracaksanız bunu yapmanız gerekebilir:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          repos:
            development:
              pattern:
                - '*/development'
                - '*/staging'
              uri: https://github.com/development/config-repo
            staging:
              pattern:
                - '*/qa'
                - '*/production'
              uri: https://github.com/staging/config-repo
```

Spring Cloud, * ile bitmeyen bir profil içeren bir kalıbın, aslında bu kalıpla başlayan bir profil listesiyle eşleştirmek istediğinizi ima ettiğini tahmin eder (bu nedenle ***/staging**, **["*/staging", "*/staging*"]** için bir kısayoldur).
Örneğin, " development" profilindeki uygulamaları yerel olarak ve aynı zamanda «cloud" profilini uzaktan çalıştırmanız gerektiğinde bu durum yaygındır.

Her repository ayrıca isteğe bağlı olarak yapılandırma dosyalarını alt dizinlerde saklayabilir ve bu dizinleri aramak için pattern'ler **search-paths** olarak belirtilebilir. Aşağıdaki örnek, en üst düzeyde bir yapılandırma dosyasını göstermektedir:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          search-paths:
            - foo
            - bar*
```

Yukarıdaki örnekte, sunucu üst düzeydeki ve **foo/** alt dizinindeki yapılandırma dosyalarını ve ayrıca adı **bar** ile başlayan herhangi bir alt dizini arar.
Varsayılan olarak, yapılandırma ilk istendiğinde server uzak repository'leri klonlar.
Server, yandadaki üst düzey örnekte gösterildiği gibi, başlangıçta repository'leri klonlayacak şekilde yapılandırılabilir:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://git/common/config-repo.git
          repos:
            team-a:
              pattern: team-a-*
              cloneOnStart: true
              uri: https://git/team-a/config-repo.git
            team-b:
              pattern: team-b-*
              cloneOnStart: false
              uri: https://git/team-b/config-repo.git
            team-c:
              pattern: team-c-*
              uri: https://git/team-a/config-repo.git
```

Config Server başlatıldığında klonlanacak bir repository ayarlamak, Config Server başlatılırken yanlış yapılandırılmış bir yapılandırma kaynağının (geçersiz bir repository URI'si gibi) hızlı bir şekilde belirlenmesine yardımcı olabilir. Bir yapılandırma kaynağı için **clonOnStart** etkinleştirilmemişse, Config Server yanlış yapılandırılmış veya geçersiz bir yapılandırma kaynağıyla başarılı bir şekilde başlayabilir ve bir uygulama bu yapılandırma kaynağından yapılandırma talep edene kadar bir hata algılamayabilir.

Yukarıdaki örnekte, sunucu, herhangi bir isteği kabul etmeden önce, team-a'nın yapılandırma deposunu başlangıçta klonlar. Diğer tüm depolar, depodan yapılandırma talep edilene kadar klonlanmaz.

Authentication (Kimlik doğrulama)

Uzak repository'de HTTP temel kimlik doğrulamasını kullanmak için, aşağıdaki örnekte gösterildiği gibi **username** ve **password** özelliklerini (URL'de değil) ayrı olarak ekleyin:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          username: trolley
          password: strongpassword
```

Eğer HTTPS ve kullanıcı kimlik bilgilerini kullanmazsanız, anahtarları varsayılan dizinlerde (**~/.ssh**) sakladığınızda ve URI **git@github.com:configuration/cloud-configuration** gibi bir SSH konumuna işaret ettiğinde SSH kutudan çıktığı gibi çalışmalıdır. Git sunucusu için bir girdinin **~/.ssh/known_hosts** dosyasında bulunması ve bunun **ssh-rsa** biçiminde olması önemlidir. Diğer biçimler (**ecdsa-sha2-nistp256** gibi) desteklenmez. Sürprizlerden kaçınmak için Git sunucusu için **known_hosts** dosyasında yalnızca bir girişin bulunduğundan ve yapılandırma sunucusuna sağladığınız URL ile eşleştiğinden emin olmalısınız. URL'de bir ana bilgisayar adı kullanıyorsanız, **known_hosts** dosyasında tam olarak buna (IP'ye değil) sahip olmak istersiniz. Depoya JGit kullanılarak erişilir, bu nedenle bulduğunuz tüm belgeler uygulanabilir olmalıdır. HTTPS proxy ayarları, sistem özellikleriyle (**-Dhttps.proxyHost** ve **-Dhttps.proxyPort**) **~/.git/config** veya (diğer JVM işlemleriyle aynı şekilde) içinde ayarlanabilir.

~/.git dizininizin nerede olduğunu bilmiyorsanız, ayarları değiştirmek için **git config --global** kullanın (örneğin, **git config --global http.sslVerify false**).

JGit, PEM formatında RSA anahtarları gerektirir. Aşağıda, corect biçiminde bir anahtar oluşturacak örnek bir ssh-keygen (openssh'den) komutu verilmiştir:

```
ssh-keygen -m PEM -t rsa -b 4096 -f ~/config_server_deploy_key.rsa
```

Uyarı: SSH anahtarlarıyla çalışırken, beklenen ssh özel anahtarı **-----BEGIN RSA PRIVATE KEY----** ile başlamalıdır. Anahtar **-----BEGIN OPENSSH PRIVATE KEY-----** ile başlıyorsa, spring-cloud-config server başlatıldığında RSA anahtarı yüklenmeyecektir. Hata şuna benziyor:

```
- Error in object 'spring.cloud.config.server.git': codes [PrivateKeyIsValid.spring.cloud.config.server.git,PrivateKeyIsValid]; arguments
[org.springframework.context.support.DefaultMessageSourceResolvable: codes [spring.cloud.config.server.git.,]; arguments []; default message []]; default message [Property
'spring.cloud.config.server.git.privateKey' is not a valid private key]
```

Yukarıdaki hatayı düzeltmek için RSA anahtarının PEM formatına dönüştürülmesi gerekir. Uygun biçimde yeni bir anahtar oluşturmak için yukarıda openssh kullanımına bir örnek verilmiştir.

Authentication with AWS CodeCommit (AWS CodeCommit ile Kimlik Doğrulama)

Spring Cloud Config Server, AWS CodeCommit kimlik doğrulamasını da destekler. AWS CodeCommit, Git'i komut satırından kullanırken bir kimlik doğrulama yardımcısı kullanır. Bu yardımcı JGit kitaplığıyla kullanılmaz, bu nedenle Git URI'si AWS CodeCommit modeliyle eşleşirse AWS CodeCommit için bir JGit CredentialProvider oluşturulur. AWS CodeCommit URI'leri bu pattern'i (modeli-deseni-kalıbı) takip eder:

```
https//git-codecommit.${AWS_REGION}.amazonaws.com/v1/repos/${repo}.
```

Bir AWS CodeCommit URI ile bir kullanıcı adı ve parola sağlarsanız, bunlar havuza erişim sağlayan AWS accessKeyId ve secretAccessKey olmalıdır. Bir kullanıcı adı ve parola belirtmezseniz, accessKeyId ve secretAccessKey, AWS Default Credential Provider Chain. kullanılarak alınır.

Git URI'niz CodeCommit URI modeliyle (daha önce gösterilen) eşleşiyorsa, kullanıcı adı ve parolada veya varsayılan kimlik bilgisi sağlayıcı zinciri tarafından desteklenen konumlardan birinde geçerli AWS kimlik bilgilerini sağlamanız gerekir. AWS EC2 bulut serverleri, IAM Roles for EC2 Instances kullanabilir.

aws-java-sdk-core jar isteğe bağlı bir bağımlılıktır. **aws-java-sdk-core** jar, sınıf yolunuzda değilse, git serveri URI'sinden bağımsız olarak AWS Code Commit kimlik bilgisi sağlayıcısı oluşturulmaz.

Authentication with Google Cloud Source (Google Cloud Source ile Kimlik Doğrulama)

Spring Cloud Config Server, Google Cloud Source depolarına karşı kimlik doğrulamayı da destekler.

Git URI'niz **http** veya **https** protokolünü kullanıyorsa ve alan adı **source.developers.google.com** ise, Google Cloud Source kimlik bilgileri sağlayıcısı kullanılır. Bir Google Cloud Source veri havuzu URI'si, **https://source.developers.google.com/p/${GCP_PROJECT}/r/${REPO}** biçimindedir. Deponuz için URI'yi almak için Google Cloud Source kullanıcı arayüzünde "Klonla"yı tıklayın ve "Manuel olarak oluşturulan kimlik bilgileri"ni seçin. Herhangi bir kimlik bilgisi oluşturmayın, görüntülenen URI'yi kopyalamanız yeterlidir.

Google Cloud Source kimlik bilgileri sağlayıcısı, Google Cloud Platform uygulamasının varsayılan kimlik bilgilerini kullanır. Bir sistem için uygulama varsayılan kimlik bilgilerinin nasıl oluşturulacağına ilişkin Google Cloud SDK belgelerine bakın. Bu yaklaşım, geliştirme ortamlarındaki kullanıcı hesapları ve üretim ortamlarındaki hizmet hesapları için çalışacaktır.

com.google.auth:google-auth-library-oauth2-http isteğe bağlı bir bağımlılıktır. **google-auth-library-oauth2-http** jar, sınıf yolunuzda değilse, git sunucusu URI'sinden bağımsız olarak Google Cloud Source kimlik bilgisi sağlayıcısı oluşturulmaz.

Git SSH configuration using properties (Özellikleri kullanarak Git SSH yapılandırması)

Spring Cloud Config Server tarafından kullanılan JGit kitaplığı varsayılan olarak Git depolarına bir SSH URI kullanarak bağlanırken **~/.ssh/known_hosts** ve **/etc/ssh/ssh_config** gibi SSH yapılandırma dosyalarını kullanır. Cloud Foundry gibi bulut ortamlarında yerel dosya sistemi geçici olabilir veya kolayca erişilebilir olmayabilir. Bu durumlarda SSH yapılandırması Java özellikleri kullanılarak ayarlanabilir. Özellik tabanlı SSH yapılandırmasını etkinleştirmek için, aşağıdaki örnekte gösterildiği gibi, **spring.cloud.config.server.git.ignoreLocalSshSettings** özelliği **true** olarak ayarlanmalıdır:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: git@gitserver.com:team/repo1.git
          ignoreLocalSshSettings: true
          hostKey: someHostKey
          hostKeyAlgorithm: ssh-rsa
          privateKey: |
                -----BEGIN RSA PRIVATE KEY-----
                MIIEpgIBAAKCAQEAx4UbaDzY5xjW6hc9jwN0mX33XpTDVW9WqHp5AKaRbtAC3DqX
                IXFMPgw3K45jxRb93f8tv9vL3rD9CUG1Gv4FM+o7ds7FRES5RTjv2RT/JVNJCoqF
                ol8+ngLqRZCyBtQN7zYByWMRirPGoDUqdPYrj2yq+ObBBNhg5N+hOwKjjpzdj2Ud
                1l7R+wxIqmJo1IYyy16xS8WsjyQuyC0lL456qkd5BDZ0Ag8j2X9H9D5220Ln7s9i
                oezTipXipS7p7Jekf3Ywx6abJwOmB0rX79dV4qiNcGgzATnG1PkXxqt76VhcGa0W
                DDVHEEYGbSQ6hIGSh0I7BQun0aLRZojfE3gqHQIDAQABAoIBAQCZmGrk8BK6tXCd
                fY6yTiKxFzwb38IQP0ojIUWNrq0+9Xt+NsypviLHkXfXCKKU4zUHeIGVRq5MN9b
                BO56/RrcQHHOoJdUWuOV2qMqJvPUtC0CpGkD+valhfD75MxoXU7s3FK7yjxy3rsG
                EmfA6tHV8/4a5umo5TqSd2YTm5B19AhRqiuUVI1wTB41DjULUGiMYrnYrhzQlVvj
                5MjnKTlYu3V8PoYDfv1GmxPPh6vlpafXEeEYN8VB97e5x3DGHjZ5UrurAmTLTdO8
                +AahyoKsIY612TkkQthJlt7FJAwnCGMgY6podzzvzICLFmmTXYiZ/28I4BX/mOSe
                pZVnfRixAoGBAO6Uiwt40/PKs53mCEWngslSCsh9oGAaLTf/XdvMns5VmuyyAyKG
                ti8Ol5wqBMi4GIUzjbgUvSUt+IowIrG3f5tN85wpjQ1UGVcpTnl5Qo9xaS1PFScQ
                xrtWZ9eNj2TsIAMp/svJsyGG3OibxfnuAIpSXNQiJPwRlW3irzpGgVx/AoGBANYW
                dnhshUcEHMJi3aXwR12OTDnaLoanVGLwLnkqLSYUZA7ZegpKq90UAuBdcEfgdpyi
                PhKpeaeIiAaNnFo8m9aoTKr+7I6/uMTlwrVnfrsVTZv3orxjwQV20YIBCVRKD1uX
                VhE0ozPZxwwKSPAFocpyWpGHGreGF1AIYBE9UBtjAoGBAI8bfPgJpyFyMiGBjO6z
                FwlJc/xlFqDusrcHL7abW5qq0L4v3R+FrJw3ZYufzLTVcKfdj6GelwJJO+8wBm+R
                gTKYJItEhT48duLIfTDyIpHGVm9+I1MGhh5zKuCqIhxIYr9jHloBB7kRm0rPvYY4
                VAykcNgyDvtAVODP+4m6JvhjAoGBALbtTqErKN47V0+JJpapLnF0KxGrqeGIjIRV
                cYA6V4WYGr7NeIfesecfOC356PyhgPfpcVyEztwlvwTKb3RzIT1TZN8fH4YBr6Ee
                KTbTjefRFhVUjQqnucAvfGi29f+9oE3Ei9f7wA+H35ocF6JvTYUsHNMIO/3gZ38N
                CPjyCMa9AoGBAMhsITNe3QcbsXAbdUR00dDsIFVROzyFJ2m40i4KCRM35bC/BIBs
                q0TY3we+ERB40U8Z2BvU61QuwaunJ2+uGadHo58VSVdggqAo0BSkH58innKKt96J
                69pcVH/4rmLbXdcmNYGm6iu+MlPQk4BUZknHSmVHIFdJ0EPupVaQ8RHT
                -----END RSA PRIVATE KEY-----
```

Aşağıdaki tablo, SSH yapılandırma özelliklerini açıklar.

Property Name(Özellik İsmi)	Remarks(Açıklama-Notlar)
ignoreLocalSshSettings	Eğer **true** ise, dosya tabanlı SSH yapılandırması yerine özellik tabanlı kullanın. Bir repository tanımı içinde değil, **spring.cloud.config.server.git.ignoreLocalSshSettings** olarak ayarlanmalıdır.
privateKey	Geçerli SSH private key. **ignoreLocalSshSettings** true ise ve Git URI'si SSH biçimindeyse ayarlanmalıdır.
hostKey	Geçerli SSH host key. **hostKeyAlgorithm** de ayarlanmışsa ayarlanmalıdır.
hostKeyAlgorithm	**ssh-dss, ssh-rsa, ecdsa-sha2-nistp256, ecdsa-sha2-nistp384 veya ecdsa-sha2-nistp521**'den biri. **HostKey** de ayarlanmışsa ayarlanmalıdır.
strictHostKeyChecking	**True** ya da **false**. Eğer false ise, ana bilgisayar anahtarıyla ilgili hataları yoksayın.
knownHostsFile	Özel **.known_hosts** dosyasının konumu.
preferredAuthentications	Sunucu kimlik doğrulama yöntemi sırasını geçersiz kıl. Bu, sunucunun **publickey** yönteminden önce klavye etkileşimli kimlik doğrulaması varsa, oturum açma istemlerinden kaçınmaya izin vermelidir.

Placeholders in Git Search Paths (Git Arama Yollarındaki yer tutucular)

Spring Cloud Config Server ayrıca aşağıdaki örnekte gösterildiği gibi **{application}** ve **{profile}** (ve gerekirse **{label}**) için yer tutucuları olan bir arama yolunu da destekler:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          search-paths: '{application}'
```

Yukarıdaki liste, dizinle aynı ada sahip dosyalar için (üst seviyenin yanı sıra) havuzun aranmasına neden olur. Joker karakterler, yer tutucuları olan bir arama yolunda da geçerlidir (eşleşen herhangi bir dizin aramaya dahil edilir).

Force pull in Git Repositories (Git Depolarında çekmeyi zorla)

Daha önce belirtildiği gibi, Spring Cloud Config Server, yerel kopyanın kirlenmesi (örneğin, bir işletim sistemi işlemi tarafından klasör içeriğinin değişmesi) durumunda, Spring Cloud Config Server'ın yerel kopyayı uzak repository'den güncelleyemediği durumlarda uzak git deposunun bir klonunu yapar.

Bu sorunu çözmek için, aşağıdaki örnekte gösterildiği gibi, yerel kopya kirliyse Spring Cloud Config Server'ı uzak repository'den çekmeye zorlayan bir **force-pull** özelliği vardır:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          force-pull: true
```

Birden çok repository yapılandırmanız varsa, aşağıdaki örnekte gösterildiği gibi repository başına **force-pull** özelliğini yapılandırabilirsiniz:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://git/common/config-repo.git
          force-pull: true
          repos:
            team-a:
              pattern: team-a-*
              uri: https://git/team-a/config-repo.git
              force-pull: true
            team-b:
              pattern: team-b-*
              uri: https://git/team-b/config-repo.git
              force-pull: true
            team-c:
              pattern: team-c-*
              uri: https://git/team-a/config-repo.git
```

Force-pull özelliği için varsayılan değer **false**'tur.

Deleting untracked branches in Git Repositories (Git Depolarında izlenmeyen dalları silme)

Spring Cloud Config Server, şubeyi yerel depoya teslim ettikten sonra uzak git repository'nin bir klonuna sahip olduğundan (örn. Bu nedenle, uzak dalın silindiği bir durum olabilir, ancak bunun yerel kopyası hala getirilmeye hazırdır. Ve Spring Cloud Config Server istemci hizmeti **--spring.cloud.config.label=deletedRemoteBranch,master** ile başlarsa, özellikleri **deleteRemoteBranch** yerel şubesinden alır, ancak **master**'dan almaz.

Yerel repository dallarını temiz ve uzak tutmak için - **deleteUntrackedBranches** özelliği ayarlanabilir. Spring Cloud Config Server'ın izlenmeyen dalları yerel repository'den silmeye **zorlar**.

Örnek:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          deleteUntrackedBranches: true
```

deleteUntrackedBranches özelliği için varsayılan değer **false** şeklindedir.

Git Refresh Rate (Git Yenileme Hızı)

Spring.cloud.config.server.git.refreshRate kullanarak, yapılandırma sunucusunun Git arka ucunuzdan güncellenmiş yapılandırma verilerini ne sıklıkta alacağını kontrol edebilirsiniz. Bu özelliğin değeri saniye cinsinden belirtilir. Varsayılan olarak değer 0'dır, yani yapılandırma sunucusu her istendiğinde Git deposundan güncellenmiş yapılandırmayı alır.

Default Label (Varsayılan Etiket)

Git için kullanılan varsayılan etiket **main**'dir. **spring.cloud.config.server.git.defaultLabel**'i ayarlamazsanız ve **main** adında bir dal yoksa, yapılandırma sunucusu varsayılan olarak **master** adlı bir dalı kullanıma almayı dener. Geri dönüş dal davranışını devre dışı bırakmak isterseniz, **spring.cloud.config.server.git.tryMasterBranch**'ı **false** olarak ayarlayabilirsiniz.

Version Control Backend Filesystem Use (Sürüm Kontrolü Arka Uç Dosya Sistemi Kullanımı)

VCS tabanlı arka uçlar (git, svn) ile dosyalar teslim alınır veya yerel dosya sistemine klonlanır. Varsayılan olarak, **config-repo-** öneki ile sistem geçici dizinine yerleştirilirler. Örneğin linux'ta **/tmp/config-repo-<randomid>** olabilir. Bazı işletim sistemleri rutin olarak geçici dizinleri temizler. Bu, eksik özellikler gibi beklenmeyen davranışlara yol açabilir. Bu sorunu önlemek için, **spring.cloud.config.server.git.basedir** veya **spring.cloud.config.server.svn.basedir**'i sistem geçici yapısında bulunmayan bir dizine ayarlayarak Config Server'ın kullandığı dizini değiştirin.

File System Backend (Dosya Sistemi Arka Ucu)

Ayrıca, Yapılandırma Sunucusunda Git'i kullanmayan, ancak yerel sınıf yolundan veya dosya sisteminden yapılandırma dosyalarını yükleyen bir "yerel" profil vardır (**spring.cloud.config.server.native.searchLocations** ile işaret etmek istediğiniz herhangi bir statik URL). Yerel profili kullanmak için, Config Server'ı **spring.profiles.active=native** ile başlatın.

Dosya kaynakları için **file:** önekini kullanmayı unutmayın (öneksiz varsayılan genellikle sınıf yoludur). Herhangi bir Spring Boot yapılandırmasında olduğu gibi, **${}** -tarzı ortam yer tutucularını gömebilirsiniz, ancak Windows'taki mutlak yolların fazladan bir **/** (örneğin, **file:///${user.home}/config-repo**) gerektirdiğini unutmayın.

searchLocations'ın varsayılan değeri, yerel Spring Boot uygulamasıyla aynıdır (yani, **[classpath:/, classpath:/config, file:./, file:./config]**). Bu, serverdeki **application.yml**'i tüm client'lere göstermez, çünkü serverde bulunan tüm özellik kaynakları client'e(alıcıya) gönderilmeden önce kaldırılır.

Bir dosya sistemi arka ucu, hızlı bir şekilde başlamak ve test etmek için harikadır. Üretimde kullanmak için dosya sisteminin güvenilir olduğundan ve Yapılandırma Sunucusunun tüm örnekleri arasında paylaşıldığından emin olmanız gerekir.

Arama konumları **{application}**, **{profile}** ve **{label}** için yer tutucular içerebilir. Bu şekilde, yoldaki dizinleri ayırabilir ve sizin için anlamlı olan bir strateji seçebilirsiniz (uygulama başına alt dizin veya profil başına alt dizin gibi).
Eğer arama konumlarında yer tutucular kullanmazsanız, bu repository ayrıca HTTP kaynağının **{label}** parametresini arama yolundaki bir son eke ekler, böylece özellikler dosyaları her arama konumundan ve aynı ada sahip bir alt dizinden yüklenir. etiket (etiketli özellikler Bahar Ortamında önceliklidir). Bu nedenle, yer tutucu içermeyen varsayılan davranış, **/{etiket}/** ile biten bir arama konumu eklemekle aynıdır. Örneğin, **file:/tmp/config**, **file:/tmp/config,file:/tmp/config/{label}** ile aynıdır. Bu davranış, **spring.cloud.config.server.native.addLabelLocations=false** ayarlanarak devre dışı bırakılabilir.

Vault Backend

Spring Cloud Config Server ayrıca Vault'u arka uç olarak destekler

Apps Kasası, gizli dizilere güvenli bir şekilde erişmek için bir araçtır. Gizli, API anahtarları, parolalar, sertifikalar ve diğer hassas bilgiler gibi erişimi sıkı bir şekilde kontrol etmek istediğiniz her şeydir. Apps Kasası, sıkı erişim kontrolü sağlarken ve ayrıntılı bir denetim günlüğü kaydederken herhangi bir sır için birleşik bir arabirim sağlar.

Yapılandırma sunucusunun bir vault backend kullanmasını sağlamak için config server'inizin **vault** profiliyle çalıştırabilirsiniz. Örneğin, config server'inizin **application.yml** dosyasına **spring.profiles.active=vault** ekleyebilirsiniz.

Spring Cloud Config Server, varsayılan olarak Vault'tan yapılandırmayı almak için Token tabanlı Kimlik Doğrulamayı kullanır. Vault ayrıca AppRole, LDAP, JWT, CloudFoundry, Kubernetes Auth gibi ek kimlik doğrulama yöntemlerini de destekler. TOKEN veya X-Config-Token başlığı dışında herhangi bir kimlik doğrulama yöntemini kullanmak için, Config Server'ın bu kitaplığa kimlik doğrulamasını devredebilmesi için sınıf yolunda Spring Vault Core'a ihtiyacımız var. Lütfen aşağıdaki bağımlılıkları Config Server Uygulamanıza ekleyin.

Maven (pom.xml)

```
<dependencies>
        <dependency>
                <groupId>org.springframework.vault</groupId>
                <artifactId>spring-vault-core</artifactId>
        </dependency>
</dependencies>
```

Gradle (build.gradle)

```
dependencies {
   implementation "org.springframework.vault:spring-vault-core"
}
```

Varsayılan olarak, config server, vault serverinizin **http://127.0.0.1:8200** adresinde çalıştığını varsayar. Ayrıca backend adının **secret** olduğunu ve anahtarın **application** olduğunu varsayar. Bu varsayılanların tümü, config server'inizin **application.yml** dosyasında yapılandırılabilir. Aşağıdaki tabloda, yapılandırılabilir Vault özellikleri açıklanmaktadır:

Name (İsim)	Default Value (Varsayılan Değer)
host	127.0.0.1
port	8200
scheme	http
backend	secret
defaultKey	application
profileSeparator	,
kvVersion	1
skipSslValidation	false
timeout	5
namespace	null

Vault 0.10.0, önceki sürümlerden farklı bir API ortaya çıkaran, sürümlü bir key-value backend (k/v backend version 2) sundu, artık bağlama yolu ile gerçek bağlam yolu arasında bir **data/** gerektiriyor ve gizli dizileri bir **data** nesnesine sarıyor . **spring.cloud.config.server.vault.kv-version=2** ayarının yapılması bunu hesaba katacaktır.

Yukarıdaki tablodaki tüm özelliklerin önüne **spring.cloud.config.server.vault** eklenmeli veya bileşik konfigürasyonun doğru vault bölümüne yerleştirilmelidir.

Tüm yapılandırılabilir özellikler **org.springframework.cloud.config.server.environment.VaultEnvironmentProperties** içinde bulunabilir.

İsteğe bağlı olarak, Vault Enterprise **X-Vault-Namespace** başlığı için destek vardır. Vault'a gönderilmesini sağlamak için **namespace** özelliğini ayarlayın.
Config server çalışırken, Vault backend'den değerleri almak için servere HTTP istekleri gönderebilirsiniz. Bunu yapmak için Vault serveriniz için bir jetona ihtiyacınız var.
İlk olarak, aşağıdaki örnekte gösterildiği gibi, Vault'unuza bazı veriler yerleştirin:

```
$ vault kv put secret/application foo=bar baz=bam
$ vault kv put secret/myapp foo=myappsbar
```

İkinci olarak, aşağıdaki örnekte gösterildiği gibi, değerleri almak için Config Server'inize bir HTTP isteği yapın:
$ curl -X "GET" "http://localhost:8888/myapp/default" -H "X-Config-Token: yourtoken"
Aşağıdakine benzer bir yanıt görmelisiniz:

```
{
  "name":"myapp",
  "profiles":[
    "default"
  ],
  "label":null,
  "version":null,
  "state":null,
  "propertySources":[
    {
      "name":"vault:myapp",
      "source":{
        "foo":"myappsbar"
      }
    },
    {
      "name":"vault:application",
      "source":{
        "baz":"bam",
        "foo":"bar"
      }
    }
  ]
}
```

İstemcinin, Config Server'ın Vault ile konuşmasına izin vermek için gerekli kimlik doğrulamasını sağlamasının varsayılan yolu, X-Config-Token başlığını ayarlamaktır. Ancak, bunun yerine, Spring Cloud Vault ile aynı yapılandırma özelliklerini ayarlayarak, başlığı atlayabilir ve serverdeki kimlik doğrulamasını yapılandırabilirsiniz. Ayarlanacak özellik, **spring.cloud.config.server.vault.authentication**'dır. Desteklenen kimlik doğrulama yöntemlerinden birine ayarlanmalıdır. Ayrıca, **spring.cloud.vault** için belgelenenle aynı özellik adlarını kullanarak, ancak bunun yerine **spring.cloud.config.server.vault** önekini kullanarak, kullandığınız kimlik doğrulama yöntemine özgü diğer özellikleri ayarlamanız gerekebilir. Daha fazla ayrıntı için Spring Cloud Vault Reference Guide'a bakın.

X-Config-Token başlığını atlarsanız ve kimlik doğrulamasını ayarlamak için bir sunucu özelliği kullanırsanız, ek kimlik doğrulama seçeneklerini etkinleştirmek için Config Server uygulamasının Spring Vault'a ek bir bağımlılığa ihtiyacı vardır. Bu bağımlılığın nasıl ekleneceğini öğrenmek için Spring Vault Reference Guide'a bakın.

Multiple Properties Sources (Çoklu Mülk Kaynakları)

Vault'ı kullanırken uygulamalarınıza birden çok özellik kaynağı sağlayabilirsiniz. Örneğin, Vault'da aşağıdaki yollara veri yazdığınızı varsayalım:

```
secret/myApp,dev
secret/myApp
secret/application,dev
secret/application
```

secret/application'a yazılan özellikler, Config Server kullanan tüm uygulamalar tarafından kullanılabilir. myApp adlı bir uygulama, **secret/myApp** ve **secret/application** için yazılmış tüm özelliklere sahip olacaktır. **myApp**, **dev** profilini etkinleştirdiğinde, yukarıdaki yolların tümüne yazılan özellikler, listedeki ilk yoldaki özellikler diğerlerine göre öncelikli olacak şekilde onun için kullanılabilir olacaktır.

Accessing Backends Through a Proxy (Bir Proxy Aracılığıyla Arka Uçlara Erişme)

Config Server, bir HTTP veya HTTPS proxy aracılığıyla Git veya Vault backend'ine erişebilir. Bu davranış, Git veya Vault için **proxy.http** ve **proxy.https** altındaki ayarlar tarafından kontrol edilir. Bu ayarlar repository başınadır, bu nedenle composite environment repository(bileşik ortam deposu) kullanıyorsanız bileşikteki her arka uç için ayrı ayrı proxy ayarlarını yapılandırmanız gerekir. HTTP ve HTTPS URL'leri için ayrı proxy sunucuları gerektiren bir ağ kullanıyorsanız, tek bir backend için hem HTTP hem de HTTPS proxy ayarlarını yapılandırabilirsiniz: bu durumda **http** erişimi **http** proxy'sini kullanır ve **https**, **https**'ye erişir. Ayrıca, uygulama ve proxy arasındaki proxy tanım protokolünü kullanarak her iki protokol için kullanılacak tek bir proxy belirleyebilirsiniz.

Aşağıdaki tablo, hem HTTP hem de HTTPS proxy'leri için proxy yapılandırma özelliklerini açıklar. Bu özelliklerin tümüne **proxy.http** veya **proxy.https** ön eki getirilmelidir.

Proxy Yapılandırma Özellikleri	
Property Name(Özellik İsmi)	**Remarks(Açıklama-Notlar)**
host	Proxy'nin ev sahibi.
port	Proxy'ye erişilecek bağlantı noktası.
nonProxyHosts	Yapılandırma sunucusunun proxy dışında erişmesi gereken ana bilgisayarlar. Hem **proxy.http.nonProxyHosts** hem de **proxy.https.nonProxyHosts** için değerler sağlanırsa, **proxy.http** değeri kullanılır.
username	Proxy'de kimlik doğrulaması yapılacak kullanıcı adı. Hem **proxy.http.username** hem de **proxy.https.username** için değerler sağlanırsa, **proxy.http** değeri kullanılır.
password	Proxy'de kimlik doğrulaması yapılacak parola. Hem **proxy.http.password** hem de **proxy.https.password** için değerler sağlanırsa, **proxy.http** değeri kullanılır.

Aşağıdaki yapılandırma, Git deposuna erişmek için bir HTTPS proxy'si kullanır.

```
spring:
  profiles:
    active: git
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          proxy:
            https:
              host: my-proxy.host.io
              password: myproxypassword
              port: '3128'
              username: myproxyusername
              nonProxyHosts: example.com
```

Sharing Configuration With All Applications (Yapılandırmayı Tüm Uygulamalarla Paylaşma)

Tüm uygulamalar arasında yapılandırmayı paylaşma, aşağıdaki konularda açıklandığı gibi, hangi yaklaşıma sahip olduğunuza göre değişir:

- Dosya Tabanlı Depolar
- Kasa Sunucusu

File Based Repositories (Dosya Tabanlı Depolar-Repository'ler)

Dosya tabanlı (git, svn ve yerel) depolarda, **application*** içindeki dosya adlarına sahip kaynaklar (**application.properties**, **application.yml**, **application-*.properties** vb.) tüm client(alıcı) uygulamaları arasında paylaşılır. Genel varsayılanları yapılandırmak ve gerektiğinde uygulamaya özel dosyalar tarafından geçersiz kılınmasını sağlamak için bu dosya adlarına sahip kaynakları kullanabilirsiniz.

Özellik geçersiz kılma özelliği, yer tutucu uygulamalarının bunları yerel olarak geçersiz kılmasına izin vererek, genel varsayılanları ayarlamak için de kullanılabilir.

" native" profille (yerel dosya sistemi backend), sunucunun kendi yapılandırmasının parçası olmayan açık bir arama konumu kullanmalısınız. Aksi takdirde, varsayılan arama konumlarındaki **application*** kaynakları, sunucunun parçası oldukları için kaldırılır.

Vault Server

Vault'ı backend olarak kullanırken, yapılandırmayı **secret/application** içine yerleştirerek yapılandırmayı tüm uygulamalarla paylaşabilirsiniz. Örneğin, aşağıdaki vault komutunu çalıştırırsanız, config server kullanan tüm uygulamalar, kendilerine sunulan **foo** ve **baz** özelliklerine sahip olacaktır:

```
$ vault write secret/application foo=bar baz=bam
```

CredHub Server

CredHub'ı backend olarak kullanırken, yapılandırmayı **/application/** içine yerleştirerek veya uygulamanın **default** profiline yerleştirerek yapılandırmayı tüm uygulamalarla paylaşabilirsiniz. Örneğin, aşağıdaki CredHub komutunu çalıştırırsanız, config server kullanan tüm uygulamalar, kendilerine sunulan **share.color1** ve **shared.color2** özelliklerine sahip olacaktır:

```
credhub set --name "/application/profile/master/shared" --type=json
değer: {"shared.color1": "blue", "shared.color": "red"}
```

```
credhub set --name "/my-app/default/master/more-shared" --type=json
value: {"shared.word1": "hello", "shared.word2": "world"}
```

AWS Secrets Manager (AWS Secrets Yöneticisi)

AWS Secrets Manager'ı backend olarak kullanırken, yapılandırmayı **/application/** içine yerleştirerek veya uygulamanın **default** profiline yerleştirerek yapılandırmayı tüm uygulamalarla paylaşabilirsiniz. Örneğin, aşağıdaki anahtarlarla secrets eklerseniz, config serveri kullanan tüm uygulamalar, **shared.foo** ve **shared.bar** özelliklerine sahip olur:

```
secret name = /secret/application-default/
```

```
secret value =
{
 shared.foo: foo,
 shared.bar: bar
}
```

veya

```
secret name = /secret/application/
```

```
secret value =
{
 shared.foo: foo,
 shared.bar: bar
}
```

AWS Parameter Store

AWS Parameter Store'u backend olarak kullanırken, özellikleri **/application** hiyerarşisine yerleştirerek tüm uygulamalarla yapılandırmayı paylaşabilirsiniz.

Örneğin, aşağıdaki adlara sahip parametreler eklerseniz, yapılandırma sunucusunu kullanan tüm uygulamalar, **foo.bar** ve **fred.baz** özelliklerine sahip olur:

```
/config/application/foo.bar
/config/application-default/fred.baz
```

JDBC Backend

Spring Cloud Config Server, yapılandırma özellikleri için bir arka uç olarak JDBC'yi (ilişkisel veritabanı) destekler. Bu özelliği, sınıf yoluna **spring-jdbc** ekleyerek ve **jdbc** profilini kullanarak veya **JdbcEnvironmentRepository** türünde bir bean ekleyerek etkinleştirebilirsiniz. Sınıf yoluna doğru bağımlılıkları eklerseniz (bununla ilgili daha fazla ayrıntı için kullanım kılavuzuna bakın), Spring Boot bir veri kaynağı yapılandırır.

spring.cloud.config.server.jdbc.enabled özelliğini **false** olarak ayarlayarak **JdbcEnvironmentRepository** için otomatik yapılandırmayı devre dışı bırakabilirsiniz.

Veritabanının, **APPLICATION**, **PROFILE** ve **LABEL** (her zamanki **Environment** anlamnında) adlı sütunları olan **PROPERTIES** adlı bir tabloya ve **Properties** stilindeki anahtar(key) ve değer(value) çiftleri için **KEY** ve **VALUE**'ye sahip olması gerekir. Java'da tüm alanlar String türündedir, böylece onları ihtiyacınız olan uzunlukta **VARCHAR** yapabilirsiniz. Özellik değerleri, işlem sonrası adımlar olarak uygulanacak tüm şifreleme ve şifre çözme dahil **{application}-{profile}.properties** adlı Spring Boot özellik dosyalarından geldiyse aynı şekilde davranır doğrudan repository uygulamasında).

Redis Backend

Spring Cloud Config Server, yapılandırma özellikleri için bir backend olarak Redis'i destekler. Spring Data Redis'e bir bağımlılık ekleyerek bu özelliği etkinleştirebilirsiniz.

pom.xml

```
<dependencies>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-data-redis</artifactId>
        </dependency>
</dependencies>
```

Aşağıdaki yapılandırma, bir Redis'e erişmek için Spring Data **RedisTemplate**'i kullanır.
spring.redis.* özelliklerini varsayılan bağlantı ayarlarını geçersiz kılmak için kullanabiliriz.

```
spring:
  profiles:
    active: redis
  redis:
    host: redis
    port: 16379
```

Özellikler, bir hash içindeki alanlar olarak saklanmalıdır. Hash adı, **spring.application.name** özelliği ile aynı olmalıdır veya **spring.application.name** ve **spring.profiles.active[n]**'nin birleşimi.

```
HMSET sample-app server.port "8100" sample.topic.name "test" test.property1 "property1"
```

Yukarıda görünen komutu çalıştırdıktan sonra, bir hash değeri olan aşağıdaki anahtarları içermelidir:

```
HGETALL sample-app
{
  "server.port": "8100",
  "sample.topic.name": "test",
  "test.property1": "property1"
}
```

Herhangi bir profil belirlenmediğinde **default** kullanılacaktır.

AWS S3 Backend

Spring Cloud Config Server, yapılandırma özellikleri için backend olarak AWS S3'ü destekler. AWS Java SDK For Amazon S3'e bir bağımlılık ekleyerek bu özelliği etkinleştirebilirsiniz.

pom.xml

```
<dependencies>
        <dependency>
                <groupId>com.amazonaws</groupId>
                <artifactId>aws-java-sdk-s3</artifactId>
        </dependency>
</dependencies>
```

Aşağıdaki yapılandırma, yapılandırma dosyalarına erişmek için AWS S3 istemcisini kullanır. Yapılandırmanızın depolandığı paketi seçmek için **spring.cloud.config.server.awss3.*** özelliklerini kullanabiliriz.

```
spring:
 profiles:
  active: awss3
 cloud:
  config:
   server:
    awss3:
     region: us-east-1
     bucket: bucket1
```

Spring.cloud.config.server.awss3.endpoint ile S3 hizmetinizin standart uç noktasını geçersiz kılmak için bir AWS URL'si belirtmek de mümkündür. Bu, S3'ün beta bölgeleri ve diğer S3 uyumlu depolama API'leri için destek sağlar.
Kimlik bilgileri, Varsayılan AWS Kimlik Bilgisi Sağlayıcı Zinciri kullanılarak bulunur. Sürümlü ve şifreli paketler, daha fazla yapılandırma olmadan desteklenir.
Yapılandırma dosyaları paketinizde **{application}-{profile}.properties**, **{application}-{profile}.yml** veya **{application}-{profile}.json** olarak saklanır. Dosyaya bir dizin yolu belirtmek için isteğe bağlı bir etiket sağlanabilir.

Herhangi bir profil belirlenmediğinde **default** kullanılacaktır.

AWS Parameter Store Backend

Spring Cloud Config Server, yapılandırma özellikleri için bir arka uç olarak AWS Parameter Store'u destekler. AWS Java SDK for SSM'ye bir bağımlılık ekleyerek bu özelliği etkinleştirebilirsiniz.

pom.xml

```
<dependency>
   <groupId>com.amazonaws</groupId>
   <artifactId>aws-java-sdk-ssm</artifactId>
</dependency>
```

Aşağıdaki yapılandırma, parametrelere erişmek için AWS SSM istemcisini kullanır.

```
spring:
  profiles:
    active: awsparamstore
  cloud:
    config:
      server:
        awsparamstore:
          region: eu-west-2
          endpoint: https://ssm.eu-west-2.amazonaws.com
          origin: aws:parameter:
          prefix: /config/service
          profile-separator: _
          recursive: true
          decrypt-values: true
          max-results: 5
```

Hiçbir uygulama belirtilmediğinde **application** varsayılandır ve hiçbir profil belirtilmediğinde **default** kullanılır.
awsparamstore.prefix için geçerli değerler, bir eğik çizgi ile başlamalı ve ardından bir veya daha fazla geçerli yol segmenti gelmeli veya boş olmalıdır.
awsparamstore.profile-separator için geçerli değerler yalnızca nokta, tire ve alt çizgi içerebilir.
awsparamstore.max-results için geçerli değerler **[1, 10]** aralığında olmalıdır.

Aşağıdaki tablo, AWS Parameter Store yapılandırma özelliklerini açıklar.

Tablo 3. AWS Parametre Deposu Yapılandırma Özellikleri

Property Name	Required	Default Value	Remarks(Açıklama)
region	no		AWS Parameter Store istemcisi tarafından kullanılacak bölge. Açıkça ayarlanmadıysa SDK, Varsayılan Bölge Sağlayıcı Zincirini kullanarak kullanılacak bölgeyi belirlemeye çalışır.
endpoint	no		AWS SSM istemcisi için giriş noktasının URL'si. Bu, API istekleri için alternatif bir uç nokta belirtmek için kullanılabilir.
origin	no	aws:ssm:parameter:	Menşeini göstermek için mülk kaynağının adına eklenen önek.
prefix	no	/config	AWS Parameter Store'dan yüklenen her özellik için parametre hiyerarşisinde L1 düzeyini gösteren önek.
profile-separator	no	-	Eklenen profili bağlam adından ayıran dize.
recursive	no	true	Bir hiyerarşi içindeki tüm AWS parametrelerinin alındığını belirtmek için bayrak.
decrypt-values	no	true	Tüm AWS parametrelerinin değerlerinin şifresi çözülmüş olarak alındığını gösteren işaret.
max-results	no	10	Bir AWS Parameter Store API çağrısı için döndürülecek maksimum öğe sayısı.

AWS Parameter Store API kimlik bilgileri, Varsayılan Kimlik Bilgisi Sağlayıcı Zinciri kullanılarak belirlenir. Sürümlü parametreler, en son sürümü döndürme varsayılan davranışıyla zaten desteklenmektedir.

AWS Secrets Manager Backend

Spring Cloud Config Server, yapılandırma özellikleri için bir arka uç olarak AWS Secrets Manager'ı destekler. AWS Java SDK for Secrets Manager'a bir bağımlılık ekleyerek bu özelliği etkinleştirebilirsiniz.

pom.xml

```
<dependency>
   <groupId>com.amazonaws</groupId>
   <artifactId>aws-java-sdk-secretsmanager</artifactId>
</dependency>
```

Aşağıdaki yapılandırma, gizli dizilere erişmek için AWS Secrets Manager client'ini kullanır.

```
spring:
  profiles:
    active: awssecretsmanager
  cloud:
    config:
      server:
        aws-secretsmanager:
          region: us-east-1
          endpoint: https://us-east-1.console.aws.amazon.com/
          origin: aws:secrets:
          prefix: /secret/foo
          profileSeparator: _
```

AWS Secrets Manager API kimlik bilgileri, Varsayılan Kimlik Bilgisi Sağlayıcı Zinciri kullanılarak belirlenir.

Hiçbir uygulama belirtilmediğinde **application** varsayılandır ve hiçbir profil belirtilmediğinde **default** kullanılır.

CredHub Backend

Spring Cloud Config Server, konfigürasyon özellikleri için bir arka uç olarak CredHub'ı destekler. Spring CredHub'a bir bağımlılık ekleyerek bu özelliği etkinleştirebilirsiniz.

pom.xml

```
<dependencies>
        <dependency>
                <groupId>org.springframework.credhub</groupId>
                <artifactId>spring-credhub-starter</artifactId>
        </dependency>
</dependencies>
```

Aşağıdaki yapılandırma, bir CredHub'a erişmek için karşılıklı TLS kullanır:

```
spring:
  profiles:
    active: credhub
  cloud:
    config:
      server:
        credhub:
          url: https://credhub:8844
```

Özellikler, aşağıdakiler gibi JSON olarak saklanmalıdır:

```
credhub set --name "/demo-app/default/master/toggles" --type=json
value: {"toggle.button": "blue", "toggle.link": "red"}
```

```
credhub set --name "/demo-app/default/master/abs" --type=json
value: {"marketing.enabled": true, "external.enabled": false}
```

spring.cloud.config.name=demo-app adlı tüm istemci uygulamaları, kendilerine sunulan aşağıdaki özelliklere sahip olacaktır:

```
{
   toggle.button: "blue",
   toggle.link: "red",
   marketing.enabled: true,
   external.enabled: false
}
```

Profil belirlenmediğinde **default** , etiket belirtilmediğinde ise varsayılan değer olarak **master** kullanılacaktır. NOT: **application**'a eklenen değerler tüm uygulamalar tarafından paylaşılacaktır.

OAuth 2.0

UAA'yı sağlayıcı olarak kullanarak OAuth 2.0 ile kimlik doğrulaması yapabilirsiniz.

pom.xml

```
<dependencies>
		<dependency>
			<groupId>org.springframework.security</groupId>
			<artifactId>spring-security-config</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.security</groupId>
			<artifactId>spring-security-oauth2-client</artifactId>
		</dependency>
</dependencies>
```

Aşağıdaki yapılandırma, bir CredHub'a erişmek için OAuth 2.0 ve UAA'yı kullanır:

```
spring:
  profiles:
    active: credhub
  cloud:
    config:
      server:
        credhub:
          url: https://credhub:8844
          oauth2:
            registration-id: credhub-client
  security:
    oauth2:
      client:
        registration:
          credhub-client:
            provider: uaa
            client-id: credhub_config_server
            client-secret: asecret
            authorization-grant-type: client_credentials
        provider:
          uaa:
            token-uri: https://uaa:8443/oauth/token
```

Kullanılan UAA istemci kimliğinin kapsam olarak **credhub.read** olması gerekir.

Composite Environment Repositories

Bazı senaryolarda, yapılandırma verilerini birden çok ortam repository'den çekmek isteyebilirsiniz. Bunu yapmak için, config serverinizin uygulama özelliklerinde veya YAML dosyasında **composite** profili etkinleştirebilirsiniz. Örneğin, iki Git repository'sinin yanı sıra bir Subversion repository'sini de yapılandırma verilerini almak istiyorsanız, yapılandırma sunucunuz için aşağıdaki özellikleri ayarlayabilirsiniz:

```
spring:
  profiles:
    active: composite
  cloud:
    config:
      server:
        composite:
        -
          type: svn
          uri: file:///path/to/svn/repo
        -
          type: git
          uri: file:///path/to/rex/git/repo
        -
          type: git
          uri: file:///path/to/walter/git/repo
```

Bu konfigürasyon kullanılarak öncelik, repository'lerin **composite** anahtar altında listelendiği sıraya göre belirlenir. Yukarıdaki örnekte, Subversion repository'si ilk olarak listelenmiştir, bu nedenle Subversion repository'sinde bulunan bir değer, Git repository'lerinden birinde aynı özellik için bulunan değerleri geçersiz kılar. **rex** Git repository'sinde bulunan bir değer, aynı özellik için **walter** Git repository'sinde bulunan bir değerden önce kullanılacaktır.

Yapılandırma verilerini yalnızca her biri farklı türdeki havuzlardan çekmek istiyorsanız, yapılandırma sunucunuzun uygulama özelliklerinde veya YAML dosyasında **composite** profil yerine ilgili profilleri etkinleştirebilirsiniz. Örneğin, tek bir Git repository'den ve tek bir HashiCorp Vault Server'den yapılandırma verilerini almak istiyorsanız, config serveriniz için aşağıdaki özellikleri ayarlayabilirsiniz:

```
spring:
  profiles:
    active: git, vault
  cloud:
    config:
      server:
        git:
          uri: file:///path/to/git/repo
          order: 2
        vault:
          host: 127.0.0.1
          port: 8200
          order: 1
```

Bu konfigürasyonu kullanarak, bir **order** özelliği ile öncelik belirlenebilir. Tüm repository'leriniz için öncelik sırasını belirtmek için **order** özelliğini kullanabilirsiniz. **Order** özelliğinin sayısal değeri ne kadar düşükse, o kadar yüksek önceliğe sahiptir. Bir repository'inin öncelik sırası, aynı özellikler için değerler içeren repository'ler arasındaki olası çakışmaları çözmeye yardımcı olur.

composite environment'ınız önceki örnekte olduğu gibi bir vault server içeriyorsa, config server'e yapılan her isteğe bir Vault tokeni eklemelisiniz. Bkz. Vault Backend.
Bir environment repository'den değerler alınırken herhangi bir tür hata, tüm composite environment için bir hatayla sonuçlanır. Bir repository başarısız olduğunda bile composite'nin devam etmesini istiyorsanız, **spring.cloud.config.server.failOnCompositeError** öğesini **false** olarak ayarlayabilirsiniz.
Bir composite environment kullanırken, tüm repository'lerin aynı etiketleri içermesi önemlidir. Önceki örneklerdekine benzer bir ortamınız varsa ve ana etiketli yapılandırma verilerini talep ederseniz, ancak Subversion deposu **master** adlı bir dal içermiyorsa, isteğin tamamı başarısız olur.

Custom Composite Environment Repositories

Spring Cloud'daki environment repository'lerden birini kullanmaya ek olarak, composite environment'ın bir parçası olarak dahil edilmek üzere kendi **EnvironmentRepository** çekirdeğinizi de sağlayabilirsiniz. Bunu yapmak için bean'ınızın **EnvironmentRepository** arayüzünü uygulaması gerekir. Özel **EnvironmentRepository**'nizin bileşik ortam içindeki önceliğini denetlemek istiyorsanız, **Ordered** arayüzünü de uygulamanız ve **getOrdered** yöntemini geçersiz kılmanız gerekir. **Ordered** arabirimini uygulamazsanız, **EnvironmentRepository**'nize en düşük öncelik verilir.

Property Overrides (Özellik Geçersiz Kılmalar)

Yapılandırma Sunucusu, operatörün tüm uygulamalara yapılandırma özellikleri sağlamasına izin veren bir "geçersiz kılma" özelliğine sahiptir. Geçersiz kılınan özellikler, normal Spring Boot kancalarıyla uygulama tarafından yanlışlıkla değiştirilemez. Geçersiz kılmaları bildirmek için, aşağıdaki örnekte gösterildiği gibi, **spring.cloud.config.server.overrides** öğesine bir ad-değer çiftleri haritası ekleyin:

```
spring:
  cloud:
    config:
      server:
        overrides:
          foo: bar
```

Yukarıdaki örnekler, yapılandırma istemcileri olan tüm uygulamaların kendi yapılandırmalarından bağımsız olarak **foo=bar** okumasına neden olur.

Bir konfigürasyon sistemi, bir uygulamayı konfigürasyon verilerini herhangi bir şekilde kullanmaya zorlayamaz. Sonuç olarak, geçersiz kılmalar uygulanabilir değildir. Ancak, Spring Cloud Config istemcileri için yararlı varsayılan davranış sağlarlar.
Normalde, ${} içeren Spring ortamı yer tutucuları, $ veya { işaretinden kaçmak için ters eğik çizgi (\) kullanılarak kaçılabilir (ve istemcide çözülebilir). Örneğin, \${app.foo:bar}, uygulama kendi app.foo'sunu sağlamadığı sürece bar olarak çözümlenir.
YAML'de ters eğik çizginin kendisinden kaçmanız gerekmez. Ancak, özellikler dosyalarında, sunucuda geçersiz kılmaları yapılandırırken ters eğik çizgiden kaçmanız gerekir.

Spring.cloud.config.overrideNone=true bayrağını (varsayılan **false**'dur) ayarlayarak, uygulamaların ortam değişkenlerinde veya Sistem özelliklerinde kendi değerlerini sağlamasına izin vererek, istemcideki tüm geçersiz kılmaların önceliğini varsayılan değerler gibi olacak şekilde değiştirebilirsiniz.

Health Indicator (Sağlık Göstergesi)

Config Server, yapılandırılmış **EnvironmentRepository**'nin çalışıp çalışmadığını kontrol eden bir Sağlık Göstergesi ile birlikte gelir. Varsayılan olarak, **EnvironmentRepository**'den **app** adlı bir uygulama, **default** profil ve **EnvironmentRepository** uygulaması tarafından sağlanan varsayılan etiket ister.

Sağlık Göstergesini, aşağıdaki örnekte gösterildiği gibi özel profiller ve özel etiketlerle birlikte daha fazla uygulamayı kontrol edecek şekilde yapılandırabilirsiniz:

```
spring:
  cloud:
    config:
      server:
        health:
          repositories:
            myservice:
              label: mylabel
            myservice-dev:
              name: myservice
              profiles: development
```

Management.health.config.enabled=false ayarını yaparak Sağlık Göstergesini devre dışı bırakabilirsiniz.

Security (Güvenlik)

Spring Security ve Spring Boot birçok güvenlik düzenlemesi için destek sunduğundan, Config Server'ınızı size mantıklı gelen herhangi bir şekilde (fiziksel ağ güvenliğinden OAuth2 taşıyıcı belirteçlerine kadar) koruyabilirsiniz.

Varsayılan Spring Boot ile yapılandırılmış HTTP Temel güvenliğini kullanmak için, Spring Security'yi sınıf yoluna dahil edin (örneğin, **spring-boot-starter-security** aracılığıyla). Varsayılan, kullanıcının **user**'ın adı ve rastgele oluşturulmuş bir paroladır. Rastgele bir parola pratikte kullanışlı değildir, bu nedenle parolayı yapılandırmanızı (**spring.security.user.password** ayarlayarak) ve şifrelemenizi öneririz (nasıl yapılacağına ilişkin talimatlar için aşağıya bakın).

Actuator and Security

Bazı platformlar sağlık kontrollerini veya benzer bir şeyi yapılandırır ve **/actuator/health** veya diğer aktüatör uç noktalarına işaret eder. Aktüatör, yapılandırma sunucusunun bir bağımlılığı değilse, **/actuator/**'e yapılan istekler, config server API'si **/{application}/{label}** ile eşleşir ve muhtemelen güvenli bilgi sızdırır. Bu durumda **spring-boot-starter-actuator** bağımlılığını eklemeyi ve kullanıcıları **/actuator/** öğesine çağrı yapan kullanıcının **/{application}/{label}** adresindeki yapılandırma sunucusu API'sine erişimi olmayacak şekilde yapılandırmayı unutmayın.

Encryption and Decryption (Şifreleme ve Şifre Çözme)

Şifreleme ve şifre çözme özelliklerini kullanmak için JVM'nizde kurulu tam güçlü JCE'ye ihtiyacınız vardır (varsayılan olarak dahil değildir). Oracle'dan “Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files” dosyasını indirebilir ve kurulum talimatlarını takip edebilirsiniz (aslında, JRE lib/security dizinindeki iki politika dosyasını indirmiş olduklarınızla değiştirmeniz gerekir).

Eğer uzak özellik kaynakları şifrelenmiş içerik içeriyorsa (**{cipher}** ile başlayan değerler), HTTP üzerinden istemcilere gönderilmeden önce bunların şifresi çözülür. Bu kurulumun ana avantajı, özellik değerlerinin "hareketsizken" (örneğin, bir git deposunda) düz metin olarak olması gerekmemesidir. Bir değerin şifresi çözülemiyorsa, özellik kaynağından kaldırılır ve aynı anahtarla ancak **invalid** ve " not applicable" anlamına gelen bir değerle (genellikle **<n/a>**) öneki ile ek bir özellik eklenir. Bu, büyük ölçüde şifreli metnin şifre olarak kullanılmasını ve yanlışlıkla sızmasını önlemek içindir.

Eğer config client uygulamaları için bir remote config repository kurarsanız, aşağıdakine benzer bir **application.yml** içerebilir:

application.yml

```
spring:
  datasource:
    username: dbuser
    password: '{cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ'
```

application.properties dosyasındaki şifreli değerler tırnak içine alınmamalıdır. Aksi takdirde, değerin şifresi çözülmez. Aşağıdaki örnek, işe yarayacak değerleri gösterir:

application.properties

```
spring.datasource.username: dbuser
spring.datasource.password: {cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ
```

Bu düz metni güvenli bir şekilde paylaşılan bir git repository'sine gönderebilirsiniz ve gizli parola korunmaya devam eder.
Server ayrıca **/encrypt** ve **/decrypt** uç noktalarını da açığa çıkarır (bunların güvenli olduğu ve yalnızca yetkili aracılar tarafından erişildiği varsayımıyla). Bir uzak yapılandırma dosyasını düzenlerseniz, aşağıdaki örnekte gösterildiği gibi **/encrypt** uç noktasına POST göndererek değerleri şifrelemek için config serveri kullanabilirsiniz:

```
$ curl localhost:8888/encrypt -s -d mysecret
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
```

Eğer curl ile test ediyorsanız, **--data-urlencode** (**-d** yerine) kullanın ve şifrelemek için değeri **=** ile önekleyin (curl bunu gerektirir) veya özel karakterler olduğunda ('+' özellikle zor) curl'ün verileri doğru şekilde kodlamasını sağlamak için açık bir **Content-Type: text/plain** ayarlayın. Şifrelenmiş değere curl komutu istatistiklerini dahil etmediğinizden emin olun, bu nedenle örneklerde onları susturmak için **-s** seçeneği kullanılır. Değerin bir dosyaya çıktısı alınması bu sorunun önlenmesine yardımcı olabilir.

Ters işlem, aşağıdaki örnekte gösterildiği gibi **/decrypt** aracılığıyla da kullanılabilir (serverin simetrik bir anahtar veya tam bir anahtar çifti ile yapılandırılması şartıyla):

```
$ curl localhost:8888/decrypt -s -d 682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
mysecret
```

Şifrelenmiş değeri alın ve YAML veya özellikler dosyasına koymadan önce ve taahhütte bulunmadan ve uzak (potansiyel olarak güvensiz) bir mağazaya göndermeden önce **{cipher}** önekini ekleyin.
/encrypt ve **/decrypt** uç noktalarının her ikisi de **/*/{application}/{profiles}** biçimindeki yolları kabul eder; bu, alıcılar(clients) aradığında her uygulama (ad) ve profil bazında kriptografiyi kontrol etmek için kullanılabilir. ana çevre kaynağı.

Kriptografiyi bu ayrıntılı şekilde kontrol etmek için, ad ve profil başına farklı bir şifreleyici oluşturan **TextEncryptorLocator** türünde bir **@Bean** sağlamanız gerekir. Varsayılan olarak sağlanan bunu yapmaz (tüm şifrelemeler aynı anahtarı kullanır).

spring komut satırı client'ı (alıcısı) (Spring Cloud CLI uzantıları yüklenmiş olarak), aşağıdaki örnekte gösterildiği gibi şifrelemek ve şifresini çözmek için de kullanılabilir:

```
$ spring encrypt mysecret --key foo
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
$ spring decrypt --key foo
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
mysecret
```

Bir dosyada bir anahtar (şifreleme için bir RSA ortak anahtarı gibi) kullanmak için, anahtar değerinin başına "@" ekleyin ve aşağıdaki örnekte gösterildiği gibi dosya yolunu sağlayın:

```
$ spring encrypt mysecret --key @${HOME}/.ssh/id_rsa.pub
AQAjPgt3eFZQXwt8tsHAVv/QHiY5sI2dRcR+...
```

--key argümanı zorunludur (bir **--** önekine sahip olmasına rağmen).

Key Management

Config Server simetrik (paylaşılan) bir anahtar veya asimetrik bir anahtar (RSA anahtar çifti) kullanabilir. Asimetrik seçim güvenlik açısından üstündür, ancak **bootstrap.properties**'de yapılandırılacak tek bir özellik değeri olduğundan simetrik bir anahtar kullanmak genellikle daha uygundur.

Simetrik bir anahtarı yapılandırmak için **encrypt.key**'i gizli bir Dize olarak ayarlamanız (veya düz metin yapılandırma dosyalarının dışında tutmak için **ENCRYPT_KEY** ortam değişkenini kullanmanız) gerekir.

asimetrik anahtarı **encrypt.key** kullanarak yapılandıramazsınız.

Asimetrik bir anahtarı yapılandırmak için bir **keyStore** (anahtar deposu) kullanın (ör. JDK ile birlikte gelen **keytool** yardımcı programı tarafından oluşturulduğu gibi). **keyStore** (anahtar deposu) özellikleri, * ile eşit olan **encrypt.keyStore.*** şeklindedir.

Property (Özellik)	Description (Tanım)
encrypt.keyStore.location	Bir **Kaynak** konumu içerir
encrypt.keyStore.password	Anahtar deposunun kilidini açan parolayı tutar
encrypt.keyStore.alias	Mağazada hangi anahtarın kullanılacağını tanımlar
encrypt.keyStore.type	Oluşturulacak Anahtar Deposu türü. Varsayılan olarak **jks**'dir.

Şifreleme, genel anahtarla yapılır ve şifrenin çözülmesi için özel bir anahtara ihtiyaç vardır. Bu nedenle, ilke olarak, yalnızca şifrelemek istiyorsanız (ve değerleri yerel olarak özel anahtarla kendiniz çözmeye hazırsanız) serverde yalnızca genel anahtarı yapılandırabilirsiniz. Pratikte, yerel olarak şifre çözme yapmak istemeyebilirsiniz, çünkü anahtar yönetim sürecini serverde yoğunlaştırmak yerine tüm client'lara(alıcılara) yayar. Öte yandan, config serveriniz nispeten güvenli değilse ve yalnızca birkaç client'ın(alıcının) şifrelenmiş özelliklere ihtiyacı varsa, bu yararlı bir seçenek olabilir.

Creating a Key Store for Testing (Test için Anahtar Deposu Oluşturma)

Test için bir keyStore(anahtar deposu) oluşturmak için aşağıdakine benzer bir komut kullanabilirsiniz:

```
$ keytool -genkeypair -alias mytestkey -keyalg RSA \
  -dname "CN=Web Server,OU=Unit,O=Organization,L=City,S=State,C=US" \
  -keypass changeme -keystore server.jks -storepass letmein
```

JDK 11 veya üstünü kullanırken, yukarıdaki komutu kullanırken aşağıdaki uyarıyı alabilirsiniz.
Bu durumda, muhtemelen **keypass** ve **storepass** değerlerinin eşleştiğinden emin olmalısınız.

```
Warning:  Different store and key passwords not supported for PKCS12 KeyStores. Ignoring user-specified -keypass value.
```

server.jks dosyasını sınıf yoluna koyun (for instance) ve ardından Config Server için **bootstrap.yml** dosyanızda aşağıdaki ayarları oluşturun:

```
encrypt:
  keyStore:
    location: classpath:/server.jks
    password: letmein
    alias: mytestkey
    secret: changeme
```

Using Multiple Keys and Key Rotation (Birden Çok Tuş Kullanma ve Tuş Döndürme)

Şifrelenmiş özellik değerlerindeki **{cipher}** önekine ek olarak, Config Server (Base64 ile kodlanmış) şifre metninin başlangıcından önce sıfır veya daha fazla **{name:value}** öneki arar. Anahtarlar, şifre için bir **TextEncryptor** bulmak için ihtiyaç duyduğu mantığı yapabilen bir **TextEncryptorLocator**'a iletilir. Bir keyStore (**encrypt.keystore.location**) yapılandırdıysanız, varsayılan konum belirleyici, aşağıdakine benzer bir şifre metni ile **key** öneki tarafından sağlanan diğer adlara sahip key'leri(anahtarları) arar:

```
foo:
  bar: `{cipher}{key:testkey}...`
```

Konumlandırıcı, "testkey" adlı bir anahtarı arar. Bir secret, önekte **{secret:...}** değeri kullanılarak da sağlanabilir. Ancak, sağlanmazsa, varsayılan keyStore(anahtar deposu) parolasını kullanmaktır (bu, bir keyStore oluşturduğunuzda ve bir secret belirtmediğinizde elde ettiğiniz şeydir). Bir secret sağlarsanız, özel bir **SecretLocator** kullanarak secreti de şifrelemeniz gerekir.

Key'ler(anahtarlar) yalnızca birkaç baytlık yapılandırma verisini şifrelemek için kullanıldığında (yani başka bir yerde kullanılmadığında), kriptografik zeminlerde anahtar döndürme neredeyse hiç gerekli değildir. Ancak, ara sıra anahtarları değiştirmeniz gerekebilir (örneğin, bir güvenlik ihlali durumunda). Bu durumda, tüm client'ların(alıcıların) kaynak yapılandırma dosyalarını (örneğin git'te) değiştirmesi ve tüm şifrelerde yeni bir **{key:...}** öneki kullanması gerekir. Client'ların önce key(anahtar) diğer adının Config Server keyStore' de bulunup bulunmadığını kontrol etmesi gerektiğini unutmayın.

Config Server'in tüm şifrelemeyi ve şifre çözmeyi işlemesine izin vermek istiyorsanız, **{name:value}** önekleri **/encrypt** uç noktasına gönderilen düz metin olarak da eklenebilir.

Serving Encrypted Properties

Bazen client'ların(alıcıların) yapılandırmayı serverde yapmak yerine yerel olarak deşifre etmesini istersiniz. Bu durumda, bir anahtarı bulmak için **encrypt.*** yapılandırmasını sağlarsanız, yine de **/encrypt** ve **/decrypt** uç noktalarına sahip olabilirsiniz, ancak **spring.cloud.config.server.encrypt.enabled=false** 'ı **bootstrap.[yml|properties]**'e yerleştirerek giden özelliklerin şifresini çözmeyi açıkça kapatmanız gerekir. Eğer Uç noktaları(endpoints) umursamıyorsanız, anahtarı veya etkin bayrağı yapılandırmazsanız çalışması gerekir.

Serving Alternative Formats

Environment endpoint'lerden(uç noktalarından) varsayılan JSON formatı, doğrudan **Environment** soyutlaması ile eşleştiğinden, Spring uygulamaları tarafından tüketim için mükemmeldir. Dilerseniz kaynak yoluna bir sonek (".yml", ".yaml" veya ".properties") ekleyerek YAML veya Java özellikleriyle aynı verileri tüketebilirsiniz. Bu, JSON uç noktalarının yapısını veya sağladıkları ekstra meta verileri umursamayan uygulamaların tüketimi için faydalı olabilir (örneğin, Spring kullanmayan bir uygulama bu yaklaşımın basitliğinden yararlanabilir).

YAML ve özellik temsillerinde, kaynak belgelerdeki (standart Spring **${…}** formunda) yer tutucuların, mümkün olduğunda, oluşturmadan önce çıktıda çözülmesi gerektiğini belirtmek için ek bir bayrak (**resolvePlaceholders** adı verilen bir boole sorgu parametresi olarak sağlanır) bulunur. . Bu, Spring yer tutucu kuralları hakkında bilgisi olmayan tüketiciler için faydalı bir özelliktir.

YAML veya özellik biçimlerinin kullanılmasında, özellikle meta veri kaybıyla ilgili olarak sınırlamalar vardır. Örneğin, JSON, kaynakla ilişkili adlarla sıralı bir özellik kaynakları listesi olarak yapılandırılmıştır. YAML ve özellikler formları, değerlerin kökeni birden fazla kaynağa sahip olsa ve orijinal kaynak dosyalarının adları kaybolsa bile tek bir haritada birleştirilir. Ayrıca, YAML temsili, bir destek repository'sinde da YAML kaynağının aslına uygun bir temsili olmak zorunda değildir. Düz özellik kaynaklarının bir listesinden oluşturulmuştur ve anahtarların şekli hakkında varsayımlar yapılmalıdır.

Serving Plain Text (Düz Metin Sunma)

Uygulamalarınız, **Environment** soyutlamasını (veya bunun YAML veya özellikler biçimindeki alternatif temsillerinden birini) kullanmak yerine, environment'lerine(ortamlarına) göre uyarlanmış genel düz metin yapılandırma dosyalarına ihtiyaç duyabilir. Config Server bunları **/{application}/{profile}/{label}/{path}** konumundaki ek bir uç nokta aracılığıyla sağlar; burada **application**, **profile** ve **label** normal environment bitiş noktasıyla aynı anlama gelir, ancak **path** bir dosya adının bir yoludur(**log.xml** gibi). Bu uç noktanın kaynak dosyaları, environment endpoints'iyle(uç noktalarıyla) aynı şekilde bulunur. Aynı arama yolu, özellikler ve YAML dosyaları için kullanılır. Ancak, eşleşen tüm kaynakları toplamak yerine, yalnızca eşleşen ilk kaynak döndürülür.

Bir kaynak bulunduktan sonra normal biçimdeki (**${…}**) yer tutucular, sağlanan uygulama adı, profili ve etiketi için etkin **Environment** kullanılarak çözümlenir. Bu şekilde, kaynak uç noktası, environment uç noktalarıyla sıkı bir şekilde tümleştirilir.

Environment(Ortam) yapılandırması için kaynak dosyalarında olduğu gibi, dosya adını çözümlemek için **profile** kullanılır. Dolayısıyla, profile-specific bir dosya istiyorsanız, **/*/development/*/logback.xml**, **logback-development.xml** (tercihen **logback.xml**) adlı bir dosya ile çözülebilir.

Eğer **label**'i sağlamak istemiyorsanız ve serverin varsayılan label'i(etiketi) kullanmasına izin verirseniz, bir **useDefaultLabel** istek parametresi sağlayabilirsiniz. Sonuç olarak, **default** profil için önceki örnek **/sample/default/nginx.conf?useDefaultLabel** olabilir.

Şu anda Spring Cloud Config, git, SVN, yerel arka uçlar ve AWS S3 için düz metin sunabilir. Git, SVN ve yerel arka uçlar için destek aynıdır. AWS S3 biraz farklı çalışır. Aşağıdaki bölümler her birinin nasıl çalıştığını gösterir:

- Git, SVN ve Native Backends
- AWS S3

Git, SVN, and Native Backends

Bir GIT veya SVN deposu veya native backend için aşağıdaki örneği göz önünde bulundurun:

```
application.yml
nginx.conf
```

nginx.conf aşağıdaki listeye benzeyebilir:

```
server {
  listen          80;
  server_name     ${nginx.server.name};
}
```

application.yml aşağıdaki listeye benzeyebilir:

```
nginx:
 server:
  name: example.com
---
spring:
 profiles: development
nginx:
 server:
  name: develop.com
```

/sample/default/master/nginx.conf kaynağı aşağıdaki gibi olabilir:

```
server {
  listen          80;
  server_name     example.com;
}
```

/sample/development/master/nginx.conf aşağıdaki gibi olabilir:

```
server {
  listen          80;
  server_name     develop.com;
}
```

AWS S3

AWS s3 için düz metin sunmayı etkinleştirmek için Config Server uygulamasının Spring Cloud AWS'ye bir bağımlılık içermesi gerekir. Bu bağımlılığın nasıl kurulacağına ilişkin ayrıntılar için Spring Cloud AWS Reference Guide'ına bakın. Ardından Spring Cloud AWS Referans Kılavuzunda açıklandığı gibi Spring Cloud AWS'yi yapılandırmanız gerekir.

Decrypting Plain Text (Düz Metnin Şifresini Çözme)

Varsayılan olarak, düz metin dosyalarındaki şifrelenmiş değerlerin şifresi çözülmez. Düz metin dosyaları için şifre çözmeyi etkinleştirmek için, **bootstrap. [yml|properties]** 'de **spring.cloud.config.server.encrypt.enabled=true** ve **spring.cloud.config.server.encrypt.plainTextEncrypt=true** ayarlayın.

Düz metin dosyalarının şifresinin çözülmesi yalnızca YAML, JSON ve özellikler dosya uzantıları için desteklenir.

Bu özellik etkinleştirilirse ve desteklenmeyen bir dosya uzantısı istenirse, dosyadaki şifrelenmiş değerlerin şifresi çözülmez.

Embedding the Config Server (Yapılandırma Sunucusunu Gömme)

Config Server en iyi bağımsız bir uygulama olarak çalışır. Ancak gerekirse başka bir uygulamaya gömebilirsiniz. Bunu yapmak için **@EnableConfigServer** anatasyonunu kullanın. Bu durumda, **spring.cloud.config.server.bootstrap** adlı isteğe bağlı bir özellik yararlı olabilir. Serverin kendisini kendi remote repository'den yapılandırması gerekip gerekmediğini gösteren bir bayraktır. Varsayılan olarak, başlatmayı geciktirebileceğinden bayrak kapalıdır. Ancak, başka bir uygulamaya gömülü olduğunda, başka herhangi bir uygulamayla aynı şekilde başlatmak mantıklıdır. **spring.cloud.config.server.bootstrap**'i **true** olarak ayarlarken, bir bileşik ortam havuzu(a composite environment repository) yapılandırması da kullanmanız gerekir. Örneğin

```
spring:
  application:
    name: configserver
  profiles:
    active: composite
  cloud:
    config:
      server:
        composite:
          - type: native
            search-locations: ${HOME}/Desktop/config
        bootstrap: true
```

Boostrap bayrağını kullanırsanız, config serverin adının ve repository URI'sinin **bootstrap.yml** içinde yapılandırılmış olması gerekir.

Sunucu uç noktalarının(endpoints'lerinin) konumunu değiştirmek için, kaynakları bir önek altında sunmak için (isteğe bağlı olarak) **spring.cloud.config.server.prefix**'i (örneğin, **/config**) ayarlayabilirsiniz. Önek bir **/** ile başlamalı ancak bitmemelidir. Config Serverdeki **@RequestMappings**'e uygulanır (yani, Spring Boot **server.servletPath** ve **server.contextPath** öneklerinin altında).
Bir uygulamanın yapılandırmasını doğrudan backend repository'den (yapılandırma sunucusu yerine) okumak istiyorsanız, temel olarak uç noktaları olmayan yerleşik bir config server istersiniz. **@EnableConfigServer** anatasyonunu kullanmayarak uç noktaları(endpoints) tamamen kapatabilirsiniz (**spring.cloud.config.server.bootstrap=true** ayarlayın).

Push Notifications and Spring Cloud Bus

Birçok kaynak kodu repository sağlayıcısı (Github, Gitlab, Gitea, Gitee, Gogs veya Bitbucket gibi), bir webhook(web kancası) aracılığıyla bir repository'deki değişiklikleri size bildirir. Webhook(Web kancasını) sağlayıcının kullanıcı arabirimi aracılığıyla bir URL ve ilgilendiğiniz bir dizi olay olarak yapılandırabilirsiniz. Örneğin, Github, bir taahhüt listesi içeren bir JSON gövdesi ve **push** olarak ayarlanmış bir başlık (**X-Github-Event**) içeren webhook(web kancasına) bir POST kullanır. **Spring-cloud-config-monitor** kitaplığına bir bağımlılık ekler ve Config Server'inizde Spring Cloud Bus'ı etkinleştirirseniz, daha sonra bir **/monitor** uç noktası etkinleştirilir.

Webhook(Web kancası) etkinleştirildiğinde, Config Server değişmiş olabileceğini düşündüğü uygulamaları hedefleyen bir **RefreshRemoteApplicationEvent** gönderir. Değişiklik tespiti stratejik hale getirilebilir. Ancak, varsayılan olarak, uygulama adıyla eşleşen dosyalardaki değişiklikleri arar (örneğin, **foo.properties**, **foo** uygulamasında hedeflenirken **application.properties** tüm uygulamalarda hedeflenir). Davranışı geçersiz kılmak istediğinizde kullanılacak strateji, istek başlıklarını ve gövdesini parametre olarak kabul eden ve değişen dosya yollarının bir listesini döndüren **PropertyPathNotificationExtractor**'dır.

Varsayılan yapılandırma, Github, Gitlab, Gitea, Gitee, Gogs veya Bitbucket ile kutudan çıkar çıkmaz çalışır. Github, Gitlab, Gitee veya Bitbucket'ten gelen JSON bildirimlerine ek olarak, **path={application}** modelinde form kodlu gövde parametreleriyle **/monitor**'a POST göndererek bir değişiklik bildirimini tetikleyebilirsiniz. Bunu yapmak, **{application}** modeliyle (joker karakterler içerebilir) eşleşen uygulamalara yayın yapar.

RefreshRemoteApplicationEvent, yalnızca **spring-cloud-bus** hem Config Server'de hem de client uygulamasında etkinleştirilirse iletilir.

Varsayılan yapılandırma, yerel git repository'lerindeki dosya sistemi değişikliklerini de algılar. Bu durumda, webhook(web kancası) kullanılmaz. Ancak, bir yapılandırma dosyasını düzenlediğiniz anda bir yenileme yayınlanır.

Spring Cloud Config Client

Bir Spring Boot uygulaması, Spring Config Server'dan (veya uygulama geliştiricisi tarafından sağlanan diğer harici özellik kaynaklarından) anında yararlanabilir. Ayrıca, **Environment** değişikliği olaylarıyla ilgili bazı ek kullanışlı özellikler alır.

Spring Boot Config Data Import

Spring Boot 2.4, **spring.config.import** özelliği aracılığıyla yapılandırma verilerini içe aktarmanın yeni bir yolunu tanıttı. Bu artık Config Server'a bağlanmanın varsayılan yoludur.
İsteğe bağlı olarak config server'e bağlanmak için application.properties'de aşağıdakileri ayarlayın:

application.properties

```
spring.config.import=optional:configserver:
```

Bu, "http://localhost:8888" varsayılan konumunda Config Serve'e bağlanacaktır. **optional:** önekinin kaldırılması, Config Server'a bağlanamazsa Config Client'ın başarısız olmasına neden olur. Config Server'ın konumunu değiştirmek için ya **spring.cloud.config.uri**'yi ayarlayın ya da url'yi **spring.config.import=optionel:configserver:http://myhost:8888** gibi **spring.config.import** ifadesine ekleyin. Import özelliğindeki konum, uri özelliğine göre önceliğe sahiptir.

Spring.config.import yoluyla içe aktarmanın Spring Boot Config Data metodu için bir **bootstrap** dosyası (properties veya yaml) gerekli değildir.

Config First Bootstrap

Config Server'a bağlanmanın legacy bootstrap yöntemini kullanmak için, bootstrap'ın bir özellik veya **spring-cloud-starter-bootstrap** başlatıcısı aracılığıyla etkinleştirilmesi gerekir. Özellik, **spring.cloud.bootstrap.enabled=true**'dur. Bir Sistem Özelliği veya environment değişkeni olarak ayarlanmalıdır. Bootstrap etkinleştirildikten sonra, sınıf yolundaki Spring Cloud Config Client ile herhangi bir uygulama Config Server'a aşağıdaki şekilde bağlanacaktır:

Bir config client'ı başladığında, Config Server'a bağlanır (**spring.cloud.config.uri** bootstrap konfigürasyon özelliği aracılığıyla) ve Spring **Environment**'ı remote(uzak) özellik kaynaklarıyla başlatır.

Bu davranışın net sonucu, Config Server kullanmak isteyen tüm client(alıcı) uygulamalarının, server adresi **spring.cloud.config.uri**'de ayarlanmış olan bir **bootstrap.yml**'ye ihtiyaç duymasıdır.(varsayılan olarak "http:/ /localhost:8888").

Discovery First Lookup

config first bootstrap'i kullanmıyorsanız, konfigürasyon özelliklerinizde **optional:** bir önek ile bir **spring.config.import** özelliğine sahip olmanız gerekir. Örneğin, **spring.config.import=optional:configserver:**.

Eğer Spring Cloud Netflix ve Eureka Service Discovery veya Spring Cloud Consul gibi bir **DiscoveryClient** implementasyonu kullanıyorsanız, Config Server'ın Discovery Service'e kaydolmasını sağlayabilirsiniz.
Config Server'ı bulmak için **DiscoveryClient** kullanmayı tercih ederseniz, bunu **spring.cloud.config.discovery.enabled=true** (varsayılan **false**'dır) ayarlayarak yapabilirsiniz. Örneğin, Spring Cloud Netflix ile Eureka server adresini tanımlamanız gerekir (örneğin, **eureka.client.serviceUrl.defaultZone** içinde). Bu seçeneği kullanmanın bedeli, hizmet kaydını bulmak için başlangıçta ek bir ağ gidiş-dönüş yolculuğudur. Avantajı, Discovery Service sabit bir nokta olduğu sürece Config Server'ın koordinatlarını değiştirebilmesidir. Varsayılan hizmet kimliği **configserver**'dır, ancak bunu client'te **spring.cloud.config.discovery.serviceId** ayarını yaparak (ve serverde, bir hizmet için olağan şekilde, örneğin **spring.application.name** ayarlayarak) değiştirebilirsiniz. .
Keşif client'i uygulamalarının tümü bir tür meta veri haritasını destekler (örneğin, Eureka için **eureka.instance.metadataMap**'imiz var). Client'lerin doğru şekilde bağlanabilmesi için Config Server'ın bazı ek özelliklerinin hizmet kaydı meta verilerinde yapılandırılması gerekebilir. Config Server, HTTP Temel ile güvence altına alınmışsa, kimlik bilgilerini **user** ve **password** olarak yapılandırabilirsiniz. Ayrıca, Config Server'in bir bağlam yolu varsa, **configPath**'i ayarlayabilirsiniz. Örneğin, aşağıdaki YAML dosyası, Eureka istemcisi olan bir Config Server içindir:

```
eureka:
 instance:
  ...
  metadataMap:
   user: osufhalskjrtl
   password: lviuhlszvaorhvlo5847
   configPath: /config
```

Discovery First Bootstrap Using Eureka And WebClient

Spring Cloud Netflix'ten Eureka **DiscoveryClient** kullanıyorsanız ve Jersey veya **RestTemplate** yerine **WebClient** kullanmak istiyorsanız, **WebClient**'i sınıf yolunuza eklemeniz ve **eureka.client.webclient.enabled=true** ayarlamanız gerekir.

Config Client Fail Fast

Bazı durumlarda, Config Server'a bağlanamıyorsa bir hizmetin başlatılmasında başarısız olmak isteyebilirsiniz. İstenen davranış buysa, istemcinin bir İstisna ile durmasını sağlamak için önyükleme yapılandırma özelliğini **spring.cloud.config.fail-fast=true** ayarlayın.

Spring.config.import kullanarak benzer işlevsellik elde etmek için **optional:** önekini çıkarmanız yeterlidir.

Config Client Retry

Uygulamanız başladığında config serverin ara sıra kullanılamayacağını düşünüyorsanız, bir hatadan sonra denemeye devam etmesini sağlayabilirsiniz. İlk olarak, **spring.cloud.config.fail-fast=true** değerini ayarlamanız gerekir. Ardından, sınıf yolunuza **spring-retry** ve **spring-boot-starter-aop** eklemeniz gerekir. Varsayılan davranış, 1000 ms'lik bir başlangıç geri çekilme aralığı ve sonraki geri çekilmeler için 1,1'lik bir üstel çarpan ile altı kez yeniden denemektir. **spring.cloud.config.retry.*** yapılandırma özelliklerini ayarlayarak bu özellikleri (ve diğerlerini) yapılandırabilirsiniz.

Yeniden deneme davranışının tam denetimini almak ve legacy bootstrap'ı kullanmak için, **configServerRetryInterceptor** kimliğine sahip **RetryOperationsInterceptor** türünde bir **@Bean** ekleyin. Spring Retry, bir tane oluşturmayı destekleyen bir **RetryInterceptorBuilder**'a sahiptir.

Config Client Retry with spring.config.import

Retry, Spring Boot **spring.config.import** deyimiyle çalışır ve normal özellikler çalışır. Ancak, import ifadesi **application-prod.properties** gibi bir profildeyse, retry'i yapılandırmak için farklı bir yola ihtiyacınız vardır. Yapılandırma, import ifadesine url parametreleri olarak yerleştirilmelidir.

application-prod.properties

```
spring.config.import=configserver:http://configserver.example.com?fail-fast=true&max-attempts=10&max-interval=1500&multiplier=1.2&initial-interval=1100"
```

Bu, **spring.cloud.config.fail-fast=true** (yukarıdaki eksik ön eke dikkat edin) ve mevcut tüm **spring.cloud.config.retry.*** yapılandırma özelliklerini ayarlar.

Locating Remote Configuration Resources

Config Server hizmeti,client uygulamasındaki varsayılan bağlamaların aşağıdaki gibi olduğu **/{application}/{profile}/{label}** öğesinden özellik kaynakları sunar:

- «application" = **${spring.application.name}**
- "profile" = **${spring.profiles.active}** (aslında **Environment.getActiveProfiles()**)
- «label" = " master"

${spring.application.name} özelliğini ayarlarken, doğru özellik kaynağını çözen sorunları önlemek için uygulama adınızın önüne ayrılmış **application-** kelimesi eklemeyin.

spring.cloud.config.* ayarını yaparak hepsini geçersiz kılabilirsiniz (burada ***; **name**, **profile** veya **label**'dir). **Label**, konfigürasyonun önceki sürümlerine geri dönmek için kullanışlıdır. Varsayılan Config Server uygulamasıyla bir git label'i, şube adı veya kesinleştirme kimliği olabilir. Label ayrıca virgülle ayrılmış bir liste olarak da sağlanabilir. Bu durumda listedeki maddeler tek tek başarılı olana kadar denenir. Bu davranış, bir özellik dalında çalışırken faydalı olabilir. Örneğin, yapılandırma label'i dalınızla hizalamak, ancak optional(isteğe bağlı) yapmak isteyebilirsiniz
(bu durumda, **spring.cloud.config.label=myfeature,develop** kullanın).

Specifying Multiple Urls for the Config Server

Dağıtılmış birden çok Config Server örneğiniz olduğunda ve bir veya daha fazla örneğin zaman zaman kullanılamaması durumunda yüksek kullanılabilirlik sağlamak için, birden çok URL belirtebilirsiniz (**spring.cloud.config.uri** özelliği altında virgülle ayrılmış bir liste olarak) veya tüm örneklerinizi Eureka gibi bir Hizmet Kaydı'na kaydettirin (Discovery-First Bootstrap modu kullanılıyorsa). Bunu yapmanın, yalnızca Config Server çalışmadığında (yani uygulamadan çıkıldığında) veya bir bağlantı zaman aşımı oluştuğunda yüksek kullanılabilirlik sağladığını unutmayın. Örneğin, Config Server bir 500 (Dahili Sunucu Hatası) yanıtı döndürürse veya ***Config Client'ı Config Server'den bir 401 alırsa*** (kötü kimlik bilgileri veya diğer nedenlerden dolayı), Config Client'ı diğer URL'lerden özellikleri getirmeye çalışmaz. Bu tür bir hata, kullanılabilirlik sorunu yerine kullanıcı sorununu gösterir.

Config Server'inizde HTTP temel güvenliği kullanıyorsanız, şu anda yalnızca kimlik bilgilerini **spring.cloud.config.uri** özelliği altında belirttiğiniz her bir URL'ye yerleştirirseniz Config Server başına kimlik doğrulama bilgilerini desteklemek mümkündür. Başka bir tür güvenlik mekanizması kullanıyorsanız, (şu anda) Config Server kimlik doğrulamasını ve yetkilendirmeyi destekleyemezsiniz.

Configuring Timeouts (Zaman Aşımlarını Yapılandırma)

Zaman aşımı eşiklerini yapılandırmak istiyorsanız:

- Okuma zaman aşımları, **spring.cloud.config.request-read-timeout** özelliği kullanılarak yapılandırılabilir.
- Bağlantı zaman aşımları, **spring.cloud.config.request-connect-timeout** özelliği kullanılarak yapılandırılabilir.

Security (Güvenlik)

Sunucuda HTTP Temel güvenliğini kullanıyorsanız, client'lerden(alıcılardan) parolayı (ve varsayılan değilse kullanıcı adını) bilmesi gerekir. Kullanıcı adı ve parolayı, aşağıdaki örnekte gösterildiği gibi, config server URI'si veya ayrı kullanıcı adı ve parola özellikleri aracılığıyla belirtebilirsiniz:

```
spring:
  cloud:
    config:
     uri: https://user:secret@myconfig.mycompany.com
```

Aşağıdaki örnek, aynı bilgiyi iletmenin alternatif bir yolunu gösterir:

```
spring:
  cloud:
    config:
     uri: https://myconfig.mycompany.com
     username: user
     password: secret
```

spring.cloud.config.password ve **spring.cloud.config.username** değerleri, URI'de sağlanan her şeyi geçersiz kılar.

Uygulamalarınızı Cloud Foundry'de dağıtırsanız, parola sağlamanın en iyi yolu hizmet kimlik bilgileridir (bir yapılandırma dosyasında olması gerekmediğinden URI'deki gibi). Aşağıdaki örnek, yerel olarak ve **configserver** adlı Cloud Foundry'de kullanıcı tarafından sağlanan bir hizmet için çalışır:

```
spring:
  cloud:
    config:
     uri: ${vcap.services.configserver.credentials.uri:http://user:password@localhost:8888}
```

Config Server, client(alıcı) tarafı TLS sertifikası gerektiriyorsa, aşağıdaki örnekte gösterildiği gibi, özellikler aracılığıyla client tarafı TLS sertifikasını ve güven deposunu yapılandırabilirsiniz:

```
spring:
  cloud:
    config:
      uri: https://myconfig.myconfig.com
      tls:
        enabled: true
        key-store: <path-of-key-store>
        key-store-type: PKCS12
        key-store-password: <key-store-password>
        key-password: <key-password>
        trust-store: <path-of-trust-store>
        trust-store-type: PKCS12
        trust-store-password: <trust-store-password>
```

spring.cloud.config.tls.enabled, config client tarafı TLS'yi etkinleştirmek için true olmalıdır. **spring.cloud.config.tls.trust-store** atlandığında, bir JVM varsayılan güven repositroy'si kullanılır. **spring.cloud.config.tls.key-store-type** ve **spring.cloud.config.tls.trust-store-type** için varsayılan değer PKCS12'dir. Parola özellikleri atlandığında, boş parola olduğu varsayılır.

Başka bir güvenlik biçimi kullanıyorsanız, **ConfigServicePropertySourceLocator**'a bir **RestTemplate** sağlamanız gerekebilir (örneğin, onu bootstrap bağlamında yakalayıp enjekte ederek).

Health Indicator (Sağlık Göstergesi)

Config Client, Config Server'dan yapılandırmayı yüklemeye çalışan bir Spring Boot Health Göstergesi sağlar. Sağlık göstergesi, **health.config.enabled=false** ayarlanarak devre dışı bırakılabilir. Yanıt ayrıca performans nedenleriyle önbelleğe alınır. Yaşamak için varsayılan önbellek süresi 5 dakikadır. Bu değeri değiştirmek için, **health.config.time-to-live** özelliğini (milisaniye cinsinden) ayarlayın.

Providing A Custom RestTemplate (Özel bir RestTemplate Sağlama)

Bazı durumlarda, client'dan config server'e yapılan istekleri özelleştirmeniz gerekebilir. Tipik olarak, bunu yapmak, servere yapılan isteklerin kimliğini doğrulamak için özel **Authorization** başlıklarının iletilmesini içerir. Özel bir **RestTemplate** sağlamak için:

1.Aşağıdaki örnekte gösterildiği gibi, **PropertySourceLocator** uygulamasıyla yeni bir konfigürasyon çekirdeği oluşturun:

CustomConfigServiceBootstrapConfiguration.java

```
@Configuration
public class CustomConfigServiceBootstrapConfiguration {
   @Bean
   public ConfigServicePropertySourceLocator configServicePropertySourceLocator() {
      ConfigClientProperties clientProperties = configClientProperties();
     ConfigServicePropertySourceLocator configServicePropertySourceLocator = new
ConfigServicePropertySourceLocator(clientProperties);
      configServicePropertySourceLocator.setRestTemplate(customRestTemplate(clientProperties));
      return configServicePropertySourceLocator;
   }
}
```

Authorization üstbilgileri eklemeye yönelik basitleştirilmiş bir yaklaşım için bunun yerine **spring.cloud.config.headers.*** özelliği kullanılabilir.

2.**resources/META-INF**'de, **spring.factories** adlı bir dosya oluşturun ve aşağıdaki örnekte gösterildiği gibi özel yapılandırmanızı belirtin:

spring.factories

```
org.springframework.cloud.bootstrap.BootstrapConfiguration = com.my.config.client.CustomConfigServiceBootstrapConfiguration
```

Vault

Vault‘ı config serveriniz için backend olarak kullanırken, client’in, serverin vault‘undan değerleri alması için bir belirteç sağlaması gerekir. Bu belirteç, aşağıdaki örnekte gösterildiği gibi, **bootstrap.yml**'de **spring.cloud.config.token** ayarlanarak client içinde sağlanabilir:

```
spring:
  cloud:
    config:
      token: YourVaultToken
```

Nested Keys In Vault (Vault’ta iç içe anahtarlar)

Vault, aşağıdaki örnekte gösterildiği gibi, anahtarları(keys) Vault’ta depolanan bir değere yerleştirme özelliğini destekler:

echo -n '{"appA": {"secret": "appAsecret"}, "bar": "baz"}' | vault write secret/myapp –

Bu komut, Vault’unuza bir JSON nesnesi yazar. Spring’de bu değerlere erişmek için aşağıdaki örnekte gösterildiği gibi geleneksel dot(.) anatasyonunu kullanırsınız.

```
@Value("${appA.secret}")
String name = "World";
```

Önceki kod, **name** değişkeninin değerini **appAsecret** olarak ayarlar.