(/)

Integrando Groovy en aplicaciones Java

por Gian Mario Contessa (https://www.baeldung.com/author/gianmario-contessa/)
(https://www.baeldung.com/author/gianmario-contessa/)

Programación (https://www.baeldung.com/category/programming/)

Maravilloso (https://www.baeldung.com/tag/groovy/)

1. Introducción

En este tutorial, exploraremos las últimas técnicas para integrar Groovy en una aplicación Java.

2. Algunas palabras sobre Groovy

El lenguaje de programación Groovy es un lenguaje potente, de tipo **opcional y dinámico** . Está respaldado por Apache Software Foundation y la comunidad Groovy, con contribuciones de más de 200 desarrolladores.

Se puede utilizar para crear una aplicación completa, para crear un módulo o una biblioteca adicional que interactúa con nuestro código Java, o para ejecutar scripts evaluados y compilados sobre la marcha.

Para obtener más información, lea Introducción a Groovy Language (https://www.baeldung.com/groovy-language) o vaya a la documentación oficial (http://groovy-lang.org/) .

3. Dependencias de Maven

En el momento de escribir este artículo, la última versión estable es 2.5.7, mientras que Groovy 2.6 y 3.0 (ambos comenzaron en otoño '17) aún están en etapa alfa.

Similar a Spring Boot, **solo necesitamos incluir el *maravilloso (https://search.maven.org/search?q=g:org.codehaus.groovy%20a:groovy-all)* pom para agregar todas las dependencias** que podamos necesitar, sin preocuparnos por sus versiones:

```
    <version>${groovy.version}</version>
    <type>pom</type>
</dependency>
```

4. Compilación conjunta

Antes de entrar en detalles sobre cómo configurar Maven, necesitamos entender con qué estamos tratando.

Nuestro código contendrá archivos Java y Groovy . Groovy no tendrá ningún problema para encontrar las clases de Java, pero ¿qué pasa si queremos que Java encuentre las clases y métodos de Groovy?

¡Aquí viene la compilación conjunta al rescate!

La compilación conjunta es un proceso diseñado para compilar archivos **Java y Groovy** en el mismo proyecto, en un solo comando Maven.

Con la compilación conjunta, el compilador Groovy:

- analizar los archivos de origen
- dependiendo de la implementación, cree apéndices que sean compatibles con el compilador de Java
- invoque el compilador de Java para compilar los apéndices junto con las fuentes de Java, de esta forma las clases de Java pueden encontrar dependencias de Groovy
- compilar las fuentes Groovy: ahora nuestras fuentes Groovy pueden encontrar sus dependencias Java

Dependiendo del complemento que lo implemente, es posible que se nos solicite separar los archivos en carpetas específicas o decirle al compilador dónde encontrarlos.

Sin una compilación conjunta, los archivos fuente de Java se compilarían como si fueran fuentes Groovy. A veces esto podría funcionar ya que la mayoría de la sintaxis de Java 1.7 es compatible con Groovy, pero la semántica sería diferente.

5. Complementos del compilador Maven

Hay algunos complementos de compilador disponibles que admiten la compilación conjunta , cada uno con sus fortalezas y debilidades.

Los dos más utilizados con Maven son Groovy-Eclipse Maven y GMaven +.

5.1. El complemento Groven-Eclipse Maven

El complemento Groovy-Eclipse Maven (https://github.com/groovy/groovy-eclipse/wiki/Groovy-Eclipse-Maven-plugin#why-another-groovy-compiler-for-maven-what-about-gmaven) **simplifica la compilación conjunta al evitar la generación de** apéndices, que sigue siendo un paso obligatorio para otros compiladores como GMaven + , pero presenta algunas peculiaridades de configuración.

Para permitir la recuperación de los artefactos más nuevos del compilador, tenemos que agregar el repositorio

```
        <id>bintray</id>
        <name>Groovy Bintray</name>
        <url>https://dl.bintray.com/groovy/maven (https://dl.bintray.com/groovy/maven)</url>
        <releases>
            <!-- avoid automatic updates -->
            <updatePolicy>never</updatePolicy>
        </releases>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </pluginRepository>
</pluginRepositories>
```

Luego, en la sección de complementos, **le decimos al compilador Maven qué versión del compilador Groovy tiene que usar.**

De hecho, el plugin usaremos - Maven plugin del compilador (https://www.baeldung.com/maven-compiler-plugin) - en realidad no compilar, pero en lugar delegados el trabajo a la *groovy-Eclipse-lotes* artefacto (https://search.maven.org/search?q=g:org.codehaus.groovy%20a:groovy-eclipse-batch) :

```
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.0</version>
    <configuration>
        <compilerId>groovy-eclipse-compiler</compilerId>
        <source>${java.version}</source>
        <target>${java.version}</target>
    </configuration>
    <dependencies>
        <dependency>
            <groupId>org.codehaus.groovy</groupId>
            <artifactId>groovy-eclipse-compiler</artifactId>
            <version>3.3.0-01</version>
        </dependency>
        <dependency>
            <groupId>org.codehaus.groovy</groupId>
            <artifactId>groovy-eclipse-batch</artifactId>
            <version>${groovy.version}-01</version>
        </dependency>
    </dependencies>
</plugin>
```

La versión de dependencia *groovy-all* debe coincidir con la versión del compilador.

Finalmente, necesitamos configurar nuestro descubrimiento automático de origen: de manera predeterminada, el compilador buscaría en carpetas como *src / main / java* y *src / main / groovy,* pero **si nuestra carpeta java está vacía, el compilador no buscará nuestro groovy fuentes** .

El mismo mecanismo es válido para nuestras pruebas.

Para forzar el descubrimiento de archivos, podemos añadir cualquier archivo en *src / / java principal* y *src / test / java* , o simplemente añadir el *maravilloso-Eclipse-compilador de* plug-in (https://search.maven.org/search?q=g:org.codehaus.groovy%20a:groovy-eclipse-compiler) :

```
<plugin>
    <groupId>org.codehaus.groovy</groupId>
    <artifactId>groovy-eclipse-compiler</artifactId>
    <version>3.3.0-01</version>
    <extensions>true</extensions>
</plugin>
```

La sección *<extensión>* es obligatoria para permitir que el complemento agregue la fase de compilación adicional y los objetivos, que contienen las dos carpetas de origen Groovy.

5.2. El complemento GMavenPlus

El complemento GMavenPlus (https://github.com/groovy/GMavenPlus/wiki/Choosing-Your-Build-Tool) puede tener un nombre similar al antiguo complemento GMaven, pero en lugar de crear un simple parche, el autor hizo un esfuerzo para **simplificar y desacoplar el compilador de una versión específica de Groovy** .

Para hacerlo, el complemento se separa de las pautas estándar para los complementos del compilador.

El compilador GMavenPlus **agrega soporte para características que todavía no estaban presentes en otros compiladores en ese momento** , como invocador dinámico (http://groovy-lang.org/indy.html) , la consola de shell interactiva y Android.

Por otro lado, presenta algunas complicaciones:

- se **modifica directorios de origen de Maven** para contener tanto el las fuentes Groovy y Java, pero no los talones de Java
- que **nos obliga a gestionar los talones** si no les borramos con los objetivos adecuados

Para configurar nuestro proyecto, necesitamos agregar el complemento gmavenplus (https://search.maven.org/search?q=g:org.codehaus.gmavenplus%20a:gmavenplus-plugin) :

```
<plugin>
    <groupId>org.codehaus.gmavenplus</groupId>
    <artifactId>gmavenplus-plugin</artifactId>
    <version>1.7.0</version>
    <executions>
        <execution>
            <goals>
                <goal>compile</goal>
                <goal>generateTestStubs</goal>
                <goal>compileTests</goal>
                <goal>removeStubs</goal>
                <goal>removeTestStubs</goal>
            </goals>
        </execution>
    </executions>
    <dependencies>
        <dependency>
            <groupId>org.codehaus.groovy</groupId>
            <artifactId>groovy-all</artifactId>
            <!-- any version of Groovy \>= 1.5.0 should work here -->
            <version>2.5.6</version>
            <scope>runtime</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</plugin>
```

Para permitir la prueba de este complemento, creamos un segundo archivo pom llamado *gmavenplus-pom.xml* en la muestra.

5.3. Compilar con el complemento Eclipse-Maven

Ahora que todo está configurado, finalmente podemos construir nuestras clases.

En el ejemplo que proporcionamos, creamos una aplicación Java simple en la carpeta fuente *src / main / java* y algunos scripts Groovy en *src / main / groovy* , donde podemos crear clases y scripts Groovy.

Construyamos todo con el complemento Eclipse-Maven:

```
$ mvn clean compile
...
[INFO] --- maven-compiler-plugin:3.8.0:compile (default-compile) @ core-groovy-2 ---
[INFO] Changes detected - recompiling the module!
[INFO] Using Groovy-Eclipse compiler to compile both Java and Groovy files
...
```

Aquí vemos que **Groovy está compilando todo** .

5.4. Compilando con GMavenPlus

GMavenPlus muestra algunas diferencias:

```
$ mvn -f gmavenplus-pom.xml clean compile
...
[INFO] --- gmavenplus-plugin:1.7.0:generateStubs (default) @ core-groovy-2 ---
[INFO] Using Groovy 2.5.7 to perform generateStubs.
```

```
[INFO] Compiling 3 source files to XXX\Baeldung\TutorialsRepo\core-groovy-2\target\classes
[INFO]
...
[INFO] --- gmavenplus-plugin:1.7.0:compile (default) @ core-groovy-2 ---
[INFO] Using Groovy 2.5.7 to perform compile.
[INFO] Compiled 2 files.
[INFO]
...
[INFO] --- gmavenplus-plugin:1.7.0:removeStubs (default) @ core-groovy-2 ---
[INFO]
...
```

Notamos de inmediato que GMavenPlus realiza los pasos adicionales de:

1. Generando trozos, uno para cada archivo maravilloso
2. Compilando los archivos Java - stubs y código Java por igual
3. Compilando los archivos Groovy

Al generar stubs, GMavenPlus hereda una debilidad que causó muchos dolores de cabeza a los desarrolladores en los últimos años, al trabajar con la compilación conjunta.

En el escenario ideal, todo funcionaría bien, pero al introducir más pasos también tenemos más puntos de falla: por ejemplo, **la compilación puede fallar antes de poder limpiar los stubs.**

Si esto sucede, los trozos viejos que quedan pueden confundir nuestro IDE, que luego mostraría errores de compilación donde sabemos que todo debe ser correcto.

Solo una construcción limpia evitaría una dolorosa y larga caza de brujas.

5.5. Dependencias de empaquetado en el archivo Jar

Para **ejecutar el programa como un jar (https://www.baeldung.com/executable-jar-with-maven) desde la línea de comandos** , agregamos el *complemento maven-assembly-plugin* (https://search.maven.org/search?q=g:org.apache.maven.plugins%20a:maven-assembly-plugin) , que incluirá todas las dependencias de Groovy en un "jar gordo" nombrado con el postfix definido en el *descriptor de* propiedadRef *:*

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>3.1.0</version>
    <configuration>
        <!-- get all project dependencies -->
        <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
        <archive>
            <manifest>
                <mainClass>com.baeldung.MyJointCompilationApp</mainClass>
            </manifest>
        </archive>
    </configuration>
    <executions>
        <execution>
            <id>make-assembly</id>
            <!-- bind to the packaging phase -->
            <phase>package</phase>
            <goals>
                <goal>single</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Una vez que se completa la compilación, podemos ejecutar nuestro código con este comando:

```
$ java -jar target/core-groovy-2-1.0-SNAPSHOT-jar-with-dependencies.jar com.baeldung.MyJointCompilationAp
```

6. Cargando código maravilloso sobre la marcha

La compilación de Maven nos permitió incluir archivos Groovy en nuestro proyecto y hacer referencia a sus clases y métodos de Java.

Sin **embargo** , esto no es suficiente si queremos cambiar la lógica en tiempo de ejecución: la compilación se ejecuta fuera de la etapa de tiempo de ejecución, por lo **que aún tenemos que reiniciar nuestra aplicación para ver nuestros cambios** .

Para aprovechar la potencia dinámica (y los riesgos) de Groovy, necesitamos explorar las técnicas disponibles para cargar nuestros archivos cuando nuestra aplicación ya se está ejecutando.

6.1. *GroovyClassLoader*

Para lograr esto, necesitamos el *GroovyClassLoader,* que puede analizar el código fuente en formato de texto o archivo y generar los objetos de clase resultantes.

Cuando la fuente es un archivo, el resultado de la compilación también se almacena en caché , para evitar sobrecarga cuando le pedimos al cargador múltiples instancias de la misma clase.

El script que proviene directamente de un **objeto *String* , en cambio, no se almacenará en caché** , por lo tanto, llamar al mismo script varias veces aún podría causar pérdidas de memoria.

GroovyClassLoader es la base sobre la que se *basan* otros sistemas de integración.

La implementación es relativamente simple:

```
    Class calcClass = loader.parseClass(
      new File("src/main/groovy/com/baeldung/", "CalcMath.groovy"));
    GroovyObject calc = (GroovyObject) calcClass.newInstance();
    return (Double) calc.invokeMethod("calcSum", new Object[] { x, y });
}

public MyJointCompilationApp() {
    loader = new GroovyClassLoader(this.getClass().getClassLoader());
    // ...
}
```

6.2. *GroovyShell*

El método *parse ()* Shell Script Loader acepta fuentes en formato de texto o archivo y **genera una instancia de la clase *Script* .**

Esta instancia hereda el método *run ()* de *Script* , que ejecuta el archivo completo de arriba a abajo y devuelve el resultado dado por la última línea ejecutada.

Si queremos, también podemos extender la *secuencia* de *comandos* en nuestro código y anular la implementación predeterminada para llamar directamente a nuestra lógica interna.

La implementación para llamar a *Script.run () se* ve así:

```
private Double addWithGroovyShellRun(int x, int y) throws IOException {
    Script script = shell.parse(new File("src/main/groovy/com/baeldung/", "CalcScript.groovy"));
    return (Double) script.run();
}

public MyJointCompilationApp() {
    // ...
    shell = new GroovyShell(loader, new Binding());
    // ...
}
```

Tenga en cuenta que *run ()* no acepta parámetros, por lo que deberíamos agregar a nuestro archivo algunas variables globales para inicializarlas a través del objeto *Binding* .

A medida que este objeto se pasa en la inicialización de *GroovyShell* , las variables se comparten con todas las instancias de *Script* .

Si preferimos un control más granular, podemos usar *invokeMethod ()* , que puede acceder a nuestros propios métodos a través de la reflexión y pasar argumentos directamente.

Veamos esta implementación:

```
private final GroovyShell shell;

private Double addWithGroovyShell(int x, int y) throws IOException {
    Script script = shell.parse(new File("src/main/groovy/com/baeldung/", "CalcScript.groovy"));
    return (Double) script.invokeMethod("calcSum", new Object[] { x, y });
}
```

```
}
```

Debajo de las cubiertas, *GroovyShell se* basa en *GroovyClassLoader* para compilar y almacenar en caché las clases resultantes, por lo que las mismas reglas explicadas anteriormente se aplican de la misma manera.

6.3. *GroovyScriptMotor*

La clase *GroovyScriptEngine* es particularmente para aquellas aplicaciones que **dependen de la recarga de un script y sus dependencias** .

Aunque tenemos estas características adicionales, la implementación tiene solo algunas pequeñas diferencias:

```
private final GroovyScriptEngine engine;

private void addWithGroovyScriptEngine(int x, int y) throws IllegalAccessException,
  InstantiationException, ResourceException, ScriptException {
    Class<GroovyObject> calcClass = engine.loadScriptByName("CalcMath.groovy");
    GroovyObject calc = calcClass.newInstance();
    Object result = calc.invokeMethod("calcSum", new Object[] { x, y });
    LOG.info("Result of CalcMath.calcSum() method is {}", result);
}

public MyJointCompilationApp() {
    ...
    URL url = null;
    try {
        url = new File("src/main/groovy/com/baeldung/").toURI().toURL();
    } catch (MalformedURLException e) {
        LOG.error("Exception while creating url", e);
    }
    engine = new GroovyScriptEngine(new URL[] {url}, this.getClass().getClassLoader());
    engineFromFactory = new GroovyScriptEngineFactory().getScriptEngine();
}
```

Esta vez tenemos que configurar las raíces de origen, y nos referimos al script con solo su nombre, que es un poco más limpio.

Mirando dentro del método *loadScriptByName* , podemos ver de inmediato la verificación *isSourceNewer* donde el motor verifica si la fuente actualmente en caché aún es válida.

Cada vez que nuestro archivo cambia, *GroovyScriptEngine* volverá a cargar automáticamente ese archivo en particular y todas las clases dependiendo de él.

Aunque esta es una característica útil y poderosa, podría causar un efecto secundario muy peligroso: **recargar muchas veces una gran cantidad de archivos resultará en una sobrecarga de la CPU sin previo aviso.**

Si eso sucede, es posible que necesitemos implementar nuestro propio mecanismo de almacenamiento en caché para tratar este problema.

6.4. *GroovyScriptEngineFactory* (JSR-223)

JSR-223 (https://jcp.org/aboutJava/communityprocess/final/jsr223/index.html) proporciona una **API estándar para invocar marcos de scripting** desde Java 6.

La implementación se ve similar, aunque volvemos a cargar a través de rutas de archivos completas.

```
private final ScriptEngine engineFromFactory;

private void addWithEngineFactory(int x, int y) throws IllegalAccessException,
  InstantiationException, javax.script.ScriptException, FileNotFoundException {
    Class calcClas = (Class) engineFromFactory.eval(
      new FileReader(new File("src/main/groovy/com/baeldung/", "CalcMath.groovy")));
    GroovyObject calc = (GroovyObject) calcClas.newInstance();
```

```
public MyJointCompilationApp() {
    // ...
    engineFromFactory = new GroovyScriptEngineFactory().getScriptEngine();
}
```

Es genial si estamos integrando nuestra aplicación con varios lenguajes de script, pero **su conjunto de características es más restringido.** Por ejemplo, **no admite la recarga de clases** . Como tal, si solo nos estamos integrando con Groovy, entonces puede ser mejor seguir con enfoques anteriores.

7. Errores de la compilación dinámica

Usando cualquiera de los métodos anteriores, podríamos crear una aplicación que **lea scripts o clases desde una carpeta específica fuera de nuestro archivo jar** .

Esto nos daría la **flexibilidad de agregar nuevas funciones mientras el sistema se está ejecutando** (a menos que necesitemos un nuevo código en la parte de Java), logrando así algún tipo de desarrollo de entrega continua.

Pero tenga cuidado con esta espada de doble filo: ahora debemos protegernos con mucho cuidado de las **fallas que podrían ocurrir tanto en el momento de la compilación como en el tiempo de ejecución** , de hecho, asegurando que nuestro código falle con seguridad.

8. Errores de ejecutar Groovy en un proyecto Java

8.1. Actuación

Todos sabemos que cuando un sistema necesita ser muy eficiente, hay que seguir algunas reglas de oro.

Dos que pueden pesar más en nuestro proyecto son:

- evitar la reflexión
- minimizar el número de instrucciones de bytecode

La reflexión, en particular, es una operación costosa debido al proceso de verificación de la clase, los campos, los métodos, los parámetros del método, etc.

Si analizamos las llamadas de método de Java a Groovy, por ejemplo, al ejecutar el ejemplo *addWithCompiledClasses* , la pila de operaciones entre *.calcSum* y la primera línea del método Groovy real se ve así:

```
calcSum:4, CalcScript (com.baeldung)
addWithCompiledClasses:43, MyJointCompilationApp (com.baeldung)
addWithStaticCompiledClasses:95, MyJointCompilationApp (com.baeldung)
main:117, App (com.baeldung)
```

Lo cual es consistente con Java. Lo mismo sucede cuando lanzamos el objeto devuelto por el cargador y llamamos a su método.

Sin embargo, esto es lo que hace la llamada *invokeMethod* :

```
calcSum:4, CalcScript (com.baeldung)
invoke0:-1, NativeMethodAccessorImpl (sun.reflect)
invoke:62, NativeMethodAccessorImpl (sun.reflect)
invoke:43, DelegatingMethodAccessorImpl (sun.reflect)
invoke:498, Method (java.lang.reflect)
invoke:101, CachedMethod (org.codehaus.groovy.reflection)
doMethodInvoke:323, MetaMethod (groovy.lang)
```

```
invokeMethod:77, Script (groovy.lang)
addWithGroovyShell:52, MyJointCompilationApp (com.baeldung)
addWithDynamicCompiledClasses:99, MyJointCompilationApp (com.baeldung)
main:118, MyJointCompilationApp (com.baeldung)
```

En este caso, podemos apreciar lo que realmente está detrás del poder de Groovy: el *MetaClass* .

Una ***MetaClass* define el comportamiento de cualquier clase de Groovy o Java, por lo que Groovy lo analiza cada vez que hay una operación dinámica que ejecutar** para encontrar el método o campo de destino. Una vez encontrado, el flujo de reflexión estándar lo ejecuta.

¡Dos reglas de oro rotas con un método de invocación!

Si necesitamos trabajar con cientos de archivos Groovy dinámicos, la **forma en que llamamos a nuestros métodos marcará una gran diferencia de rendimiento** en nuestro sistema.

8.2. Método o propiedad no encontrada

Como se mencionó anteriormente, si queremos **implementar nuevas versiones de archivos Groovy** en un ciclo de vida de CD, debemos **tratarlos como si fueran una API** separada de nuestro sistema central.

Esto significa implementar **múltiples comprobaciones a prueba de fallas y restricciones de diseño de código** para que nuestro desarrollador recién incorporado no explote el sistema de producción con un impulso incorrecto.

Ejemplos de cada uno son: tener una canalización de CI y usar la eliminación de métodos en lugar de la eliminación.

¿Qué pasa si no lo hacemos? Obtenemos terribles excepciones debido a métodos faltantes y recuentos y tipos de argumentos incorrectos.

Y si creemos que la compilación nos salvaría, veamos el método *calcSum2 ()* de nuestros scripts Groovy:

```
// this method will fail in runtime
def calcSum2(x, y) {
    // DANGER! The variable "log" may be undefined
    log.info "Executing $x + $y"
    // DANGER! This method doesn't exist!
    calcSum3()
    // DANGER! The logged variable "z" is undefined!
    log.info("Logging an undefined variable: $z")
}
```

Al examinar todo el archivo, vemos inmediatamente dos problemas: el método *calcSum3 ()* y la variable *z* no están definidos en ningún lado.

Aun así, el script se compila correctamente, sin una sola advertencia, tanto estáticamente en Maven como dinámicamente en GroovyClassLoader.

Solo fallará cuando intentemos invocarlo.

La compilación estática de Maven mostrará un error solo si nuestro código Java se refiere directamente a *calcSum3 ()* , después de *convertir* el *GroovyObject* como lo hacemos en el método *addWithCompiledClasses ()* , pero sigue siendo ineficaz si usamos la reflexión en su lugar.

9. Conclusión

En este artículo, exploramos cómo podemos integrar Groovy en nuestra aplicación Java, analizando diferentes métodos de integración y algunos de los problemas que podemos encontrar con lenguajes mixtos.

Como de costumbre, el código fuente utilizado en los ejemplos se puede encontrar en GitHub (https://github.com/eugenp/tutorials/tree/master/core-groovy-2) .

SERIE

TUTORIAL DE JAVA "VOLVER A LO BÁSICO" (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
RESTO CON SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)

ACERCA DE

SOBRE BAELDUNG (/ABOUT)
LOS CURSOS (HTTPS://COURSES.BAELDUNG.COM)
TRABAJO DE CONSULTORÍA (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
EL ARCHIVO COMPLETO (/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORES (/EDITORS)
NUESTROS COMPAÑEROS (/PARTNERS)
ANUNCIE EN BAELDUNG (/ADVERTISE)

TÉRMINOS DE SERVICIO (/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (/PRIVACY-POLICY)
INFORMACIÓN DE LA COMPAÑÍA (/BAELDUNG-COMPANY-INFO)
CONTACTO (/CONTACT)