



Spring Cloud AWS

Agim Emruli · Alain Sahli

◀ Back to index

1. Using Amazon Web Services
2. Basic setup
 - 2.1. Spring Cloud AWS maven dependency management
 - 2.2. Amazon SDK configuration
3. Cloud environment
 - 3.1. Retrieving instance metadata
 - 3.2. Integrating your Spring Cloud application with the AWS Parameter Store
 - 3.3. Integrating your Spring Cloud application with the AWS Secrets Manager
4. Managing cloud environments
 - 4.1. Automatic CloudFormation configuration
 - 4.2. Manual CloudFormation configuration
 - 4.3. CloudFormation configuration with Java config classes
 - 4.4. CloudFormation configuration in Spring Boot
 - 4.5. Manual name resolution
 - 4.6. Stack Tags
 - 4.7. Using custom CloudFormation client
5. Messaging
 - 5.1. Configuring messaging
 - 5.2. SQS support
 - 5.3. SNS support
 - 5.4. Using CloudFormation
6. Caching
 - 6.1. Configuring dependencies for Redis caches
 - 6.2. Configuring caching with XML
 - 6.3. Configuring caching using Java configuration

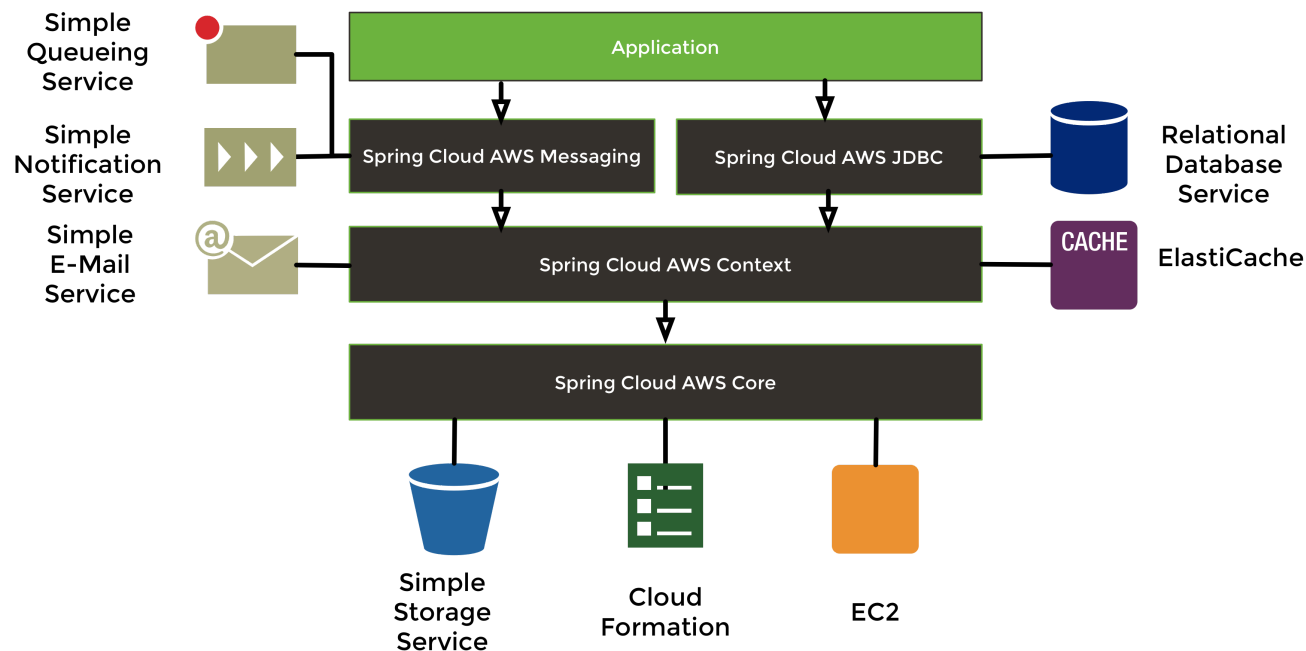
- 6.4. Configuring caching in Spring Boot
- 6.5. Using caching
- 6.6. Memcached client implementation
- 6.7. Using CloudFormation
- 7. Data Access with JDBC
 - 7.1. Configuring data source
 - 7.2. Configuring data source with Java config
 - 7.3. Configuring data source in Spring Boot
 - 7.4. Read-replica configuration
 - 7.5. Failover support
 - 7.6. CloudFormation support
 - 7.7. Database tags
- 8. Sending mails
 - 8.1. Configuring the mail sender
 - 8.2. Sending simple mails
 - 8.3. Sending attachments
 - 8.4. Configuring regions
 - 8.5. Authenticating e-mails
- 9. Resource handling
 - 9.1. Configuring the resource loader
 - 9.2. Downloading files
 - 9.3. Uploading files
 - 9.4. Searching resources
 - 9.5. Using CloudFormation

Spring Cloud for Amazon Web Services, part of the Spring Cloud umbrella project, eases the integration with hosted Amazon Web Services. It offers a convenient way to interact with AWS provided services using well-known Spring idioms and APIs, such as the messaging or caching API. Developers can build their application around the hosted services without having to care about infrastructure or maintenance.

Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at [github](#).

1. Using Amazon Web Services

Amazon provides a [Java SDK](#) to issue requests for the all services provided by the [Amazon Web Service](#) platform. Using the SDK, application developers still have to integrate the SDK into their application with a considerable amount of infrastructure related code. Spring Cloud AWS provides application developers already integrated Spring-based modules to consume services and avoid infrastructure related code as much as possible. The Spring Cloud AWS module provides a module set so that application developers can arrange the dependencies based on their needs for the particular services. The graphic below provides a general overview of all Spring Cloud AWS modules along with the service support for the respective Spring Cloud AWS services.



- **Spring Cloud AWS Core** is the core module of Spring Cloud AWS providing basic services for security and configuration setup. Developers will not use this module directly but rather through other modules. The core module provides support for cloud based environment configurations providing direct access to the instance based [EC2](#) metadata and the overall application stack specific [CloudFormation](#) metadata.
- **Spring Cloud AWS Context** delivers access to the [Simple Storage Service](#) via the Spring resource loader abstraction. Moreover developers can send e-mails using the [Simple E-Mail Service](#) and the Spring mail abstraction. Further the developers can introduce declarative caching using the Spring caching support and the [ElastiCache](#) caching service.
- **Spring Cloud AWS JDBC** provides automatic datasource lookup and configuration for the [Relational Database Service](#) which can be used with JDBC or any other support data access technology by Spring.
- **Spring Cloud AWS Messaging** enables developers to receive and send messages with the [Simple Queueing Service](#) for point-to-point communication. Publish-subscribe messaging is supported with the integration of the [Simple Notification Service](#).

- **Spring Cloud AWS Parameter Store Configuration** enables Spring Cloud applications to use the [AWS Parameter Store](#) as a Bootstrap Property Source, comparable to the support provided for the Spring Cloud Config Server or Consul's key-value store.
- **Spring Cloud AWS Secrets Manager Configuration** enables Spring Cloud applications to use the [AWS Secrets Manager](#) as a Bootstrap Property Source, comparable to the support provided for the Spring Cloud Config Server or Consul's key-value store.

2. Basic setup

Before using the Spring Cloud AWS module developers have to pick the dependencies and configure the Spring Cloud AWS module. The next chapters describe the dependency management and also the basic configuration for the Spring AWS Cloud project.

2.1. Spring Cloud AWS maven dependency management

Spring Cloud AWS module dependencies can be used directly in [Maven](#) with a direct configuration of the particular module. The Spring Cloud AWS module includes all transitive dependencies for the Spring modules and also the Amazon SDK that are needed to operate the modules. The general dependency configuration will look like this:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-aws-context</artifactId>
    <version>{spring-cloud-version}</version>
  </dependency>
</dependencies>
```

Different modules can be included by replacing the module name with the respective one (e.g. `spring-cloud-aws-messaging` instead of `spring-cloud-aws-context`)

The example above works with the Maven Central repository. To use the Spring Maven repository (e.g. for milestones or developer snapshots), you need to specify the repository location in your Maven configuration. For full releases:

```
<repositories>
  <repository>
    <id>io.spring.repo.maven.release</id>
    <url>https://repo.spring.io/release/</url>
    <snapshots><enabled>false</enabled></snapshots>
  </repository>
</repositories>
```

For milestones:

```
<repositories>
  <repository>
    <id>io.spring.repo.maven.milestone</id>
    <url>https://repo.spring.io/milestone/</url>
    <snapshots><enabled>false</enabled></snapshots>
  </repository>
</repositories>
```

```
</repository>
</repositories>
```

2.2. Amazon SDK configuration

The Spring Cloud AWS configuration is currently done using custom elements provided by Spring Cloud AWS namespaces. JavaConfig will be supported soon. The configuration setup is done directly in Spring XML configuration files so that the elements can be directly used. Each module of Spring Cloud AWS provides custom namespaces to allow the modular use of the modules. A typical XML configuration to use Spring Cloud AWS is outlined below:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aws-context="http://www.springframework.org/schema/cloud/aws/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/cloud/aws/context
    http://www.springframework.org/schema/cloud/aws/context/spring-cloud-aws-context.xsd">

    <aws-context:context-region region="..." />

</beans>
```

2.2.1. SDK credentials configuration

In order to make calls to the Amazon Web Service the credentials must be configured for the the Amazon SDK. Spring Cloud AWS provides support to configure an application context specific credentials that are used for *each* service call for requests done by Spring Cloud AWS components, with the exception of the Parameter Store and Secrets Manager Configuration. Therefore there must be **exactly one** configuration of the credentials for an entire application context.

The `com.amazonaws.auth.DefaultAWSCredentialsProviderChain` is used by all the clients if there is no dedicated credentials provider defined. This will essentially use the following authentication information

- use the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`
- use the system properties `aws.accessKeyId` and `aws.secretKey`
- use the user specific profile credentials file
- use ECS credentials if the `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` environment variable is set
- use the instance profile credentials (see below)

Based on the overall credentials policy there are different options to configure the credentials. The possible ones are described in the following sub-chapters.

Simple credentials configuration

Credentials for the Amazon SDK consist of an access key (which might be shared) and a secret key (which must **not** be shared). Both security attributes can be configured using the XML namespaces for each Amazon SDK service created by the Spring Cloud AWS module. The overall configuration looks like this

```
<beans ...>
  <aws-context:context-credentials>
    <aws-context:simple-credentials access-key="AKIAIO" secret-key="wJalrXUtnFEMI/K7M" />
  </aws-context:context-credentials>
</beans>
```

The access-key and secret-key should be externalized into property files (e.g. Spring Boot application configuration) and not be checked in into the source management system.

Instance profile configuration

An [instance profile configuration](#) allows to assign a profile that is authorized by a role while starting an EC2 instance. All calls made from the EC2 instance are then authenticated with the instance profile specific user role. Therefore there is no dedicated access-key and secret-key needed in the configuration. The configuration for the instance profile in Spring Cloud AWS looks like this:

```
<beans ...>
  <aws-context:context-credentials>
    <aws-context:instance-profile-credentials/>
  </aws-context:context-credentials>
</beans>
```

Mixing both security configurations

In some cases it is useful to combine both authentication strategies to allow the application to use the instance profile with a fallback for an explicit access-key and secret-key configuration. This is useful if the application is tested inside EC2 (e.g. on a test server) and locally for testing. The next snippet shows a combination of both security configurations.

```
<beans ...>
  <aws-context:context-credentials>
    <aws-context:instance-profile-credentials/>
    <aws-context:simple-credentials access-key="${accessKey:}" secret-key="${secretKey:}"/>
  </aws-context:context-credentials>
</beans>
```

The access-key and secret-key are defined using a placeholder expressions along with a default value to avoid bootstrap errors

if the properties are not configured at all.

Parameter Store and Secrets Manager Configuration credentials and region configuration

The Parameter Store and Secrets Manager Configuration support uses a bootstrap context to configure a default `AWS.SimpleSystemsManagement` client, which uses a `com.amazonaws.auth.DefaultAWSCredentialsProviderChain` and `com.amazonaws.regions.DefaultAwsRegionProviderChain`. If you want to override this, then you need to [define your own Spring Cloud bootstrap configuration class](#) with a bean of type `AWS.SimpleSystemsManagement` that's configured to use your chosen credentials and/or region provider. Because this context is created when your Spring Cloud Bootstrap context is created, you can't simply override the bean in a regular `@Configuration` class.

2.2.2. Region configuration

Amazon Web services are available in different [regions](#). Based on the custom requirements, the user can host the application on different Amazon regions. The `spring-cloud-aws-context` module provides a way to define the region for the entire application context.

Explicit region configuration

The region can be explicitly configured using an XML element. This is particularly useful if the region can not be automatically derived because the application is not hosted on a EC2 instance (e.g. local testing) or the region must be manually overridden.

```
<beans ...>
  <aws-context:context-region region="eu-west-1"/>
</beans>
```

It is also allowed to use expressions or placeholders to externalize the configuration and ensure that the region can be reconfigured with property files or system properties.

Automatic region configuration

If the application context is started inside an EC2 instance, then the region can automatically be fetched from the [instance metadata](#) and therefore must not be configured statically. The configuration will look like this:

```
<beans ...>
  <aws-context:context-region auto-detect="true" />
</beans>
```

Service specific region configuration

A region can also be overridden for particular services if one application context consumes services from different regions. The configuration can be done globally like described above and configured for each service with a region attribute. The configuration might look like this for a database service (described later)

```
<beans ...>
  <aws-context:context-region region="eu-central-1" />
  <jdbc:data-source ... region="eu-west-1" />
</beans>
```

While it is theoretically possible to use multiple regions per application, we strongly recommend to write applications that are hosted only inside one region and split the application if it is hosted in different regions at the same time.

2.2.3. Spring Boot auto-configuration

Following the Spring Cloud umbrella project, Spring Cloud AWS also provides dedicated Spring Boot support. Spring Cloud AWS can be configured using Spring Boot properties and will also automatically guess any sensible configuration based on the general setup.

Maven dependencies

Spring Cloud AWS provides a dedicated module to enable the Spring Boot support. That module must be added to the general maven dependency inside the application. The typical configuration will look like this

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-aws-autoconfigure</artifactId>
    <version>{spring-cloud-version}</version>
  </dependency>
</dependencies>
```

Additional dependencies to enable particular features like messaging and JDBC have to be added. Spring Cloud AWS will only configure classes that are available in the Spring Boot application's classpath.

Configuring credentials

Spring Boot provides a standard way to define properties with property file or YAML configuration files. Spring Cloud AWS provides support to configure the credential information with the Spring Boot application configuration files. Spring Cloud AWS provides the following properties to configure the credentials setup for the whole application.

property	example	description
cloud.aws.credentials.accessKey	AKIAIOSFODNN7EXAMPLE	The access key to be used with a static provider
cloud.aws.credentials.secretKey	wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY	The secret key to be used with a static provider
cloud.aws.credentials.instanceProfile	true	Configures an instance profile credentials provider with no further configuration
cloud.aws.credentials.useDefaultAwsCredentialsChain	true	Use the DefaultAWSCredentialsChain instead of configuring a custom credentials chain

Configuring region

Like for the credentials, the Spring Cloud AWS module also supports the configuration of the region inside the Spring Boot configuration files. The region can be automatically detected or explicitly configured (e.g. in case of local tests against the AWS cloud).

The properties to configure the region are shown below

property	example	description
cloud.aws.region.auto	true	Enables automatic region detection based on the EC2 meta data service
cloud.aws.region.static	eu-west-1	Configures a static region for the application. Possible regions are (currently) us-east-1, us-west-1, us-west-2, eu-west-1, eu-central-1, ap-southeast-1, ap-southeast-1, ap-northeast-1, sa-east-1, cn-north-1 and any custom region configured with own region meta data

3. Cloud environment

Applications often need environment specific configuration information, especially in changing environments like in the Amazon cloud environment. Spring Cloud AWS provides a support to retrieve and use environment specific data inside the application context using common Spring mechanisms like property placeholder or the Spring expression language.

3.1. Retrieving instance metadata

[Instance metadata](#) are available inside an EC2 environment. The metadata can be queried using a special HTTP address that provides the instance metadata. Spring Cloud AWS enables application to access this metadata directly in expression or property placeholder without the need to call an external HTTP service.

3.1.1. Enabling instance metadata support with XML

The instance metadata retrieval support is enabled through an XML element like the standard property placeholder in Spring. The following code sample demonstrates the activation of the instance metadata support inside an application context.

```
<beans ...>
  <aws-context:context-instance-data />
</beans>
```

XML

Instance metadata can be retrieved without an authorized service call, therefore the configuration above does not require any region or security specific configuration.

3.1.2. Enabling instance metadata support with Java

The instance metadata can also be configured within a Java configuration class without the need for an XML configuration. The next example shows a typical Spring `@Configuration` class that enables the instance metadata with the `org.springframework.cloud.aws.context.config.annotation.EnableInstanceData`

```
@Configuration
@EnableContextInstanceData
public static class ApplicationConfiguration {
}
```

JAVA

3.1.3. Enabling instance metadata support in Spring Boot

The instance metadata is automatically available in a Spring Boot application as a property source if the application is running on an EC2 instance.

3.1.4. Using instance metadata

Instance metadata can be used in XML, Java placeholders and expressions. The example below demonstrates the usage of instance metadata inside an XML file using placeholders and also the expression referring to the special variable `environment`

```
<beans ...>
  <bean class="org.springframework.cloud.aws....SimpleConfigurationBean">
    <property name="value1" value="#{environment.ami-id}" />
    <property name="value2" value="#{environment.hostname}" />
    <property name="value3" value="${instance-type}" />
    <property name="value4" value="${instance-id}" />
  </bean>
</beans>
```

Instance metadata can also be injected with the Spring `org.springframework.beans.factory.annotation.Value` annotation directly into Java fields. The next example demonstrates the use of instance metadata inside a Spring bean.

```
@Component
public class ApplicationInfoBean {

    @Value("${ami-id:N/A}")
    private String amiId;

    @Value("${hostname:N/A}")
    private String hostname;

    @Value("${instance-type:N/A}")
    private String instanceType;

    @Value("${services/domain:N/A}")
    private String serviceDomain;
}
```

Every instance metadata can be accessed by the key available in the [instance metadata service](#). Nested properties can be accessed by separating the properties with a slash ('/').

3.1.5. Using instance user data

Besides the default instance metadata it is also possible to configure user data on each instance. This user data is retrieved and parsed by Spring Cloud AWS. The user data can be defined while starting an EC2 instance with the application. Spring Cloud AWS expects the format `<key>:<value>;<key>:<value>` inside the user data so that it can parse the string and extract the key value pairs.


The user data can be configured using either the management console shown below or a [CloudFormation template](#).

View/Change User Data ✕

Instance ID:

User Data:

data1:value1;data2:value2

 To edit your instance's user data you first need to stop your instance.

Cancel Save

A CloudFormation template snippet for the configuration of the user data is outlined below:

```
...
"Resources": {
  "ApplicationServerInstance": {
    "Type": "AWS::EC2::Instance",
    "Properties": {
      "ImageId": "ami-6a56b81d",
      "UserData": {
        "Fn::Base64": "data1:value1;data2:value2"
      },
      "InstanceType": "t1.micro",
    }
  }
}
...
```

JSON

The user data can be accessed directly in the application context like the instance metadata through placeholders or expressions.

```
@Component
public class SecondConfigurationBean {

    @Value("${data1}")
    private String firstDataOption;
```

JAVA

```
@Value("${data2}")  
private String secondDataOption;  
}
```

3.1.6. Using instance tags

User configured properties can also be configured with tags instead of user data. Tags are a global concept in the context of Amazon Web services and used in different services. Spring Cloud AWS supports instance tags also across different services. Compared to user data, user tags can be updated during runtime, there is no need to stop and restart the instance.

User data can also be used to execute scripts on instance startup. Therefore it is useful to leverage instance tags for user configuration and user data to execute scripts on instance startup.








Instance specific tags can be configured on the instance level through the management console outlined below and like user data also with a CloudFormation template shown afterwards.

Add/Edit Tags



Apply tags to your resources to help organize and identify them.

A tag consists of a case-sensitive key-value pair. For example, you could define a tag with key = Name and value = Webserver. [Learn more](#) about tagging your Amazon EC2 resources.

Key	Value	
<input type="text" value="tag2"/>	<input type="text" value="tagv2"/>	 Show Column
<input type="text" value="tag1"/>	<input type="text" value="tagv1"/>	 Show Column
<input type="text" value="tag4"/>	<input type="text" value="tagv4"/>	 Show Column
<input type="text" value="aws:cloudformation:stack-name"/>	<input type="text" value="IntegrationTestStack"/>	 Show Column
<input type="text" value="tag3"/>	<input type="text" value="tagv3"/>	 Show Column
<input type="text" value="aws:cloudformation:stack-id"/>	<input type="text" value="arn:aws:cloudformation:eu-west-1:7"/>	 Show Column
<input type="text" value="aws:cloudformation:logical-id"/>	<input type="text" value="UserTagAndUserDataInstance"/>	 Show Column

A CloudFormation template snippet for the configuration of the instance tags is outlined below:

```
...
"Resources": {
  "UserTagAndUserDataInstance": {
    "Type": "AWS::EC2::Instance",
    "Properties": {
      "ImageId": "ami-6a56b81d",
      "InstanceType": "t1.micro",
```

JSON

```

    "Tags": [
      {
        "Key": "tag1",
        "Value": "tagv1"
      },
      {
        "Key": "tag3",
        "Value": "tagv3"
      },
      {
        "Key": "tag2",
        "Value": "tagv2"
      },
      {
        "Key": "tag4",
        "Value": "tagv4"
      }
    ]
  }
}
...

```

To retrieve the instance tags, Spring Cloud AWS has to make authenticated requests and therefore it will need the region and security configuration before actually resolving the placeholders. Also because the instance tags are not available while starting the application context, they can only be referenced as expressions and not with placeholders. The `context-instance-data` element defines an attribute `user-tags-map` that will create a map in the application context for the name. This map can then be queried using expression for other bean definitions.

```

<beans ...>
  <aws-context:context-instance-data user-tags-map="instanceData" />
</beans>

```

A java bean might resolve expressions with the `@Value` annotation.

```

public class SimpleConfigurationBean {

  @Value("#{instanceData.tag1}")
  private String value1;

  @Value("#{instanceData.tag2}")
  private String value2;

  @Value("#{instanceData.tag3}")
  private String value3;

  @Value("#{instanceData.tag4}")
  private String value4;

}

```

3.1.7. Configuring custom EC2 client

In some circumstances it is necessary to have a custom EC2 client to retrieve the instance information. The `context-instance-data` element supports a custom EC2 client with the `amazon-ec2` attribute. The next example shows the use of a custom EC2 client that

might have a special configuration in place.

```
<beans ...>

<aws-context:context-credentials>...</aws-context:context-credentials>
<aws-context:context-region ... />
<aws-context:context-instance-data amazon-ec2="myCustomClient"/>

<bean id="myCustomClient" class="com.amazonaws.services.ec2.AmazonEC2Client">
  ...
</bean>
</beans>
```

3.1.8. Injecting the default EC2 client

If there are user tags configured for the instance data (see above) Spring Cloud AWS configures an EC2 client with the specified region and security credentials. Application developers can inject the EC2 client directly into their code using the `@Autowired` annotation.

```
public class ApplicationService {

    private final AmazonEC2 amazonEc2;

    @Autowired
    public ApplicationService(AmazonEC2 amazonEc2) {
        this.amazonEc2 = amazonEc2;
    }

}
```

3.2. Integrating your Spring Cloud application with the AWS Parameter Store

Spring Cloud provides support for centralized configuration, which can be read and made available as a regular Spring `PropertySource` when the application is started. The Parameter Store Configuration allows you to use this mechanism with the [AWS Parameter Store](#).

Simply add a dependency on the `spring-cloud-starter-aws-parameter-store-config` starter module to activate the support. The support is similar to the support provided for the Spring Cloud Config Server or Consul's key-value store: configuration parameters can be defined to be shared across all services or for a specific service and can be profile-specific. Encrypted values will be decrypted when retrieved.

All configuration parameters are retrieved from a common path prefix, which defaults to `/config`. From there shared parameters are retrieved from a path that defaults to `application` and service-specific parameters use a path that defaults to the configured `spring.application.name`. You can use both dots and forward slashes to specify the names of configuration keys. Names of activated profiles will be appended to the path using a separator that defaults to an underscore.

That means that for a service called `my-service` the module by default would find and use these parameters:

parameter key	Spring property	description
/config/application/cloud.aws.stack.name	cloud.aws.stack.name	Shared by all services that have the Configuration support enabled. Can be overridden with a service- or profile-specific property.
/config/application_production/cloud.aws.stack.name	cloud.aws.stack.name	Shared by all services that have the Configuration support enabled and have a <code>production</code> Spring profile activated. Can be overridden with a service-specific property.
/config/my-service/cloud/aws/stack/auto	cloud.aws.stack.auto	Specific to the <code>my-service</code> service. Note that slashes in the key path are replaced with dots.
/config/my-service_production/cloud/aws/stack/auto	cloud.aws.stack.auto	Specific to the <code>my-service</code> service when a <code>production</code> Spring profile is activated.

Note that this module does not support full configuration files to be used as parameter values like e.g. Spring Cloud Consul does: AWS parameter values are limited to 4096 characters, so we support individual Spring properties to be configured only.

You can configure the following settings in a Spring Cloud `bootstrap.properties` or `bootstrap.yml` file (note that relaxed property binding is applied, so you don't have to use this exact syntax):

property	default	explanation
aws.paramstore.prefix	/config	Prefix indicating first level for every property loaded from the Parameter Store. Value must start with a forward slash followed by one or more valid path segments or be empty.
aws.paramstore.defaultContext	application	Name of the context that defines properties shared across all services
aws.paramstore.profileSeparator	-	String that separates an appended profile from the context name. Note that an AWS parameter key can only contain dots, dashes and underscores next to alphanumeric characters.

property	default	explanation
<code>aws.paramstore.failFast</code>	true	Indicates if an error while retrieving the parameters should fail starting the application.
<code>aws.paramstore.name</code>	the configured value for <code>spring.application.name</code>	Name to use when constructing the path for the properties to look up for this specific service.
<code>aws.paramstore.enabled</code>	true	Can be used to disable the Parameter Store Configuration support even though the auto-configuration is on the classpath.

3.3. Integrating your Spring Cloud application with the AWS Secrets Manager

Spring Cloud provides support for centralized configuration, which can be read and made available as a regular Spring `PropertySource` when the application is started. The Secrets Manager Configuration allows you to use this mechanism with the [AWS Secrets Manager](#).

Simply add a dependency on the `spring-cloud-starter-aws-secrets-manager-config` starter module to activate the support. The support is similar to the support provided for the Spring Cloud Config Server or Consul's key-value store: configuration parameters can be defined to be shared across all services or for a specific service and can be profile-specific.

All configuration parameters are retrieved from a common path prefix, which defaults to `/secret`. From there shared parameters are retrieved from a path that defaults to `application` and service-specific parameters use a path that defaults to the configured `spring.application.name`. You can use both dots and forward slashes to specify the names of configuration keys. Names of activated profiles will be appended to the path using a separator that defaults to an underscore.

That means that for a service called `my-service` the module by default would find and use these parameters:

parameter key	description
<code>/secret/application</code>	Shared by all services that have the Configuration support enabled. Can be overridden with a service- or profile-specific property.
<code>/secret/application_production</code>	Shared by all services that have the Configuration support enabled and have a <code>production</code> Spring profile activated. Can be overridden with a service-specific property.
<code>/secret/my-service</code>	Specific to the <code>my-service</code> service..

parameter key	description
<code>/secret/my-service_production</code>	Specific to the <code>my-service</code> service when a <code>production</code> Spring profile is activated.

You can configure the following settings in a Spring Cloud `bootstrap.properties` or `bootstrap.yml` file (note that relaxed property binding is applied, so you don't have to use this exact syntax):

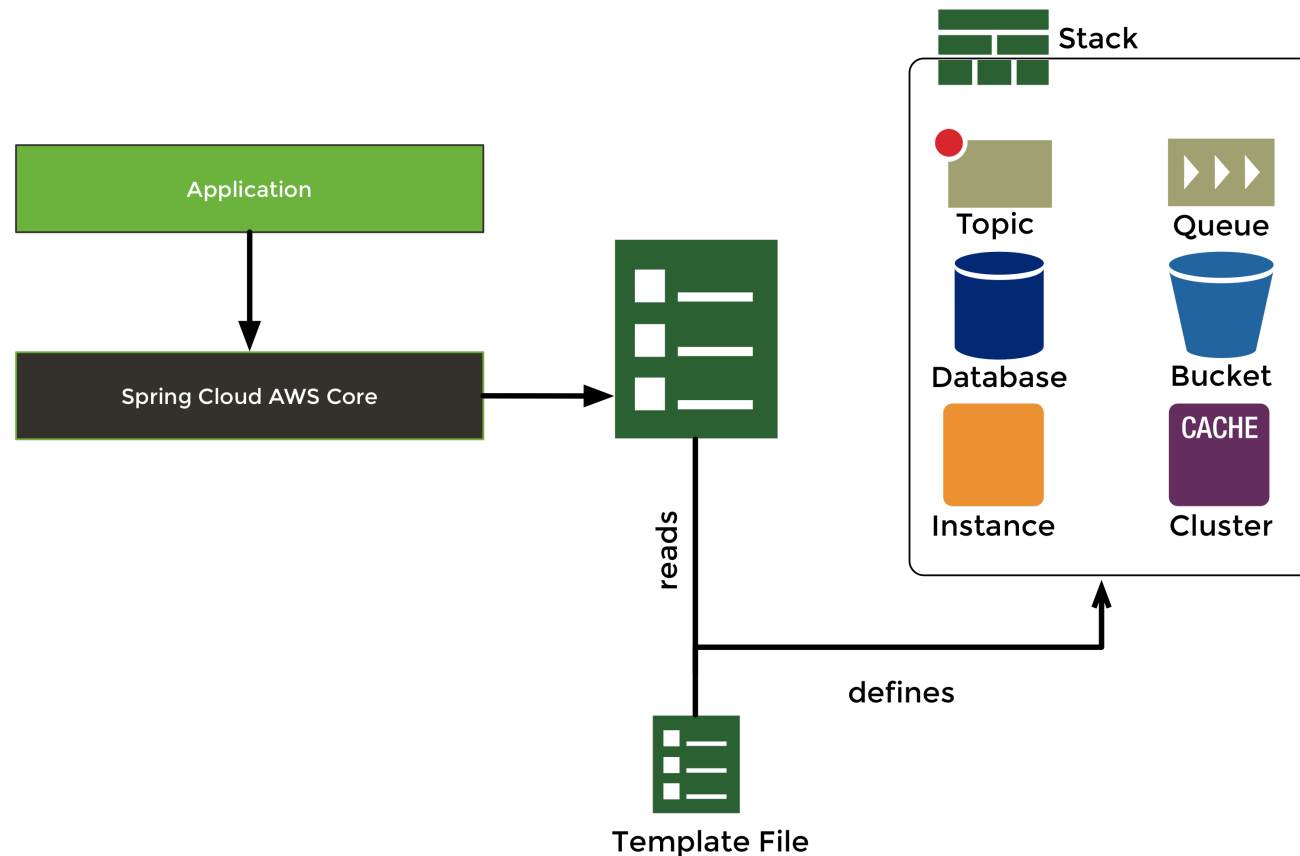
property	default	explanation
<code>aws.secretsmanager.prefix</code>	<code>/secret</code>	Prefix indicating first level for every property loaded from the Secrets Manager. Value must start with a forward slash followed by one or more valid path segments or be empty.
<code>aws.secretsmanager.defaultContext</code>	<code>application</code>	Name of the context that defines properties shared across all services
<code>aws.secretsmanager.profileSeparator</code>	<code>_</code>	String that separates an appended profile from the context name.
<code>aws.secretsmanager.failFast</code>	<code>true</code>	Indicates if an error while retrieving the secrets should fail starting the application.
<code>aws.secretsmanager.name</code>	the configured value for <code>spring.application.name</code>	Name to use when constructing the path for the properties to look up for this specific service.
<code>aws.secretsmanager.enabled</code>	<code>true</code>	Can be used to disable the Secrets Manager Configuration support even though the auto-configuration is on the classpath.

4. Managing cloud environments

Managing environments manually with the management console does not scale and can become error-prone with the increasing complexity of the infrastructure. Amazon Web services offers a [CloudFormation](#) service that allows to define stack configuration templates and bootstrap the whole infrastructure with the services. In order to allow multiple stacks in parallel, each resource in the

stack receives a unique physical name that contains some arbitrary generated name. In order to interact with the stack resources in a unified way Spring Cloud AWS allows developers to work with logical names instead of the random physical ones.

The next graphics shows a typical stack configuration.



The **Template File** describes all stack resources with their *logical name*. The **CloudFormation** service parses the stack template file and creates all resources with their *physical name*. The application can use all the stack configured resources with the *logical name* defined in the template. Spring Cloud AWS resolves all *logical names* into the respective *physical name* for the application developer.

4.1. Automatic CloudFormation configuration

If the application runs inside a stack (because the underlying EC2 instance has been bootstrapped within the stack), then Spring Cloud AWS will automatically detect the stack and resolve all resources from the stack. Application developers can use all the logical names from the stack template to interact with the services. In the example below, the database resource is configured using a CloudFormation template, defining a logical name for the database instance.

```

"applicationDatabase": {
  "Type": "AWS::RDS::DBInstance",
  "Properties": {
    "AllocatedStorage": "5",
    "DBInstanceClass": "db.t1.micro",
    "DBName": "test"
    ...
  }
}

```

The datasource is then created and will receive a physical name (e.g. `ir142c39k6o5irj`) as the database service name. Application developers can still use the logical name (in this case `applicationDatabase`) to interact with the database. The example below shows the stack configuration which is defined by the element `aws-context:stack-configuration` and resolves automatically the particular stack. The `data-source` element uses the logical name for the `db-instance-identifier` attribute to work with the database.

```

<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aws-context="http://www.springframework.org/schema/cloud/aws/context"
  xmlns="http://www.springframework.org/schema/beans"
  xsi:schemaLocation="http://www.springframework.org/schema/cloud/aws/context
    http://www.springframework.org/schema/cloud/aws/context/spring-cloud-aws-context.xsd">

  <aws-context:context-credentials>
    ...
  </aws-context:context-credentials>

  <aws-context:context-region .. />

  <aws-context:stack-configuration/>

  <jdbc:data-source db-instance-identifier="applicationDatabase" ... />
</beans>

```

Further detailed information on the Amazon RDS configuration and setup can be found in the respective chapter in this documentation.

4.2. Manual CloudFormation configuration

If the application is not running inside a stack configured EC2 instance, then the stack configuration must be configured manually. The configuration consists of an additional element attribute `stack-name` that will be used to resolve all the respective stack configuration information at runtime.

```

<beans ....>
  ...
  <aws-context:stack-configuration stack-name="myStackName" />
  ...
</beans>

```

4.3. CloudFormation configuration with Java config classes

Spring Cloud AWS also supports the configuration of the CloudFormation support within Java classes avoiding the use of XML inside the application configuration. Spring Cloud AWS provides the annotation

`org.springframework.cloud.aws.context.config.annotation.EnableStackConfiguration` that allows the automatic and manual stack configuration. The next example shows a configuration class that configures the CloudFormation support with an explicit stack name (here `manualStackName`).

```
@Configuration
@EnableStackConfiguration(stackName = "manualStackName")
class ApplicationConfiguration {
}
```

Do not define the `stackName` attribute if an automatic stack name should be enabled.

4.4. CloudFormation configuration in Spring Boot

Spring Cloud AWS also supports the configuration of the CloudFormation support within the Spring Boot configuration. The manual and automatic stack configuration can be defined with properties that are described in the table below.

property	example	description
cloud.aws.stack.name	myStackName	The name of the manually configured stack name that will be used to retrieve the resources.
cloud.aws.stack.auto	true	Enables the automatic stack name detection for the application.

4.5. Manual name resolution

Spring Cloud AWS uses the CloudFormation stack to resolve all resources internally using the logical names. In some circumstances it might be needed to resolve the physical name inside the application code. Spring Cloud AWS provides a pre-configured service to resolve the physical stack name based on the logical name. The sample shows a manual stack resource resolution.

```
@Service
public class ApplicationService {

    private final ResourceIdResolver resourceIdResolver;
```

```
@Autowired
public ApplicationService(ResourceIdResolver resourceIdResolver) {
    this.resourceIdResolver = resourceIdResolver;
}

public void handleApplicationLogic() {
    String physicalBucketName =
        this.resourceIdResolver.resolveToPhysicalResourceId("someLogicalName");
}
}
```

4.6. Stack Tags

Like for the Amazon EC2 instances, CloudFormation also provides stack specific tags that can be used to configure stack specific configuration information and receive them inside the application. This can for example be a stage specific configuration property (like DEV, INT, PRD).

```
<beans ....>
...
<aws-context:stack-configuration user-tags-map="stackTags"/>
...
</beans>
```

The application can then access the stack tags with an expression like `#{stackTags.key1}`.

4.7. Using custom CloudFormation client

Like for the EC2 configuration setup, the `aws-context:stack-configuration` element supports a custom CloudFormation client with a special setup. The client itself can be configured using the `amazon-cloud-formation` attribute as shown in the example:

```
<beans>
<aws-context:stack-configuration amazon-cloud-formation=""/>

<bean class="com.amazonaws.services.cloudformation.AmazonCloudFormationClient">
</bean>
</beans>
```

5. Messaging

Spring Cloud AWS provides [Amazon SQS](#) and [Amazon SNS](#) integration that simplifies the publication and consumption of messages over SQS or SNS. While SQS fully relies on the messaging API introduced with Spring 4.0, SNS only partially implements it as the receiving part must be handled differently for push notifications.

5.1. Configuring messaging

Before using and configuring the messaging support, the application has to include the respective module dependency into the Maven configuration. Spring Cloud AWS Messaging support comes as a separate module to allow the modularized use of the modules.

5.1.1. Maven dependency configuration

The Spring Cloud AWS messaging module comes as a standalone module and can be imported with the following dependency declaration:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-aws-messaging</artifactId>
  <version>{spring-cloud-version}</version>
</dependency>
```

5.2. SQS support

Amazon SQS is a hosted messaging service on the Amazon Web Service platform that provides point-to-point communication with queues. Compared to JMS or other message services Amazon SQS has several features and limitations that should be taken into consideration.

- Amazon SQS allows only `String` payloads, so any `Object` must be transformed into a String representation. Spring Cloud AWS has dedicated support to transfer Java objects with Amazon SQS messages by converting them to JSON.
- Amazon SQS has no transaction support, so messages might therefore be retrieved twice. Application have to be written in an idempotent way so that they can receive a message twice.
- Amazon SQS has a maximum message size of 256kb per message, so bigger messages will fail to be sent.

5.2.1. Sending a message

The `QueueMessagingTemplate` contains many convenience methods to send a message. There are send methods that specify the destination using a `QueueMessageChannel` object and those that specify the destination using a string which is going to be resolved against the SQS API. The send method that takes no destination argument uses the default destination.

```
import com.amazonaws.services.sqs.AmazonSQS;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.aws.messaging.core.QueueMessagingTemplate;
import org.springframework.messaging.support.MessageBuilder;

public class SqsQueueSender {

    private final QueueMessagingTemplate queueMessagingTemplate;

    @Autowired
    public SqsQueueSender(AmazonSQS amazonSqs) {
        this.queueMessagingTemplate = new QueueMessagingTemplate(amazonSqs);
    }
}
```



```
public void send(String message) {
    this.queueMessagingTemplate.send("physicalQueueName", MessageBuilder.withPayload(message).build());
}
}
```

This example uses the `MessageBuilder` class to create a message with a string payload. The `QueueMessagingTemplate` is constructed by passing a reference to the `AmazonSQS` client. The destination in the send method is a string value that must match the queue name defined on AWS. This value will be resolved at runtime by the Amazon SQS client. Optionally a `ResourceIdResolver` implementation can be passed to the `QueueMessagingTemplate` constructor to resolve resources by logical name when running inside a CloudFormation stack (see [Managing cloud environments](#) for more information about resource name resolution).

With the messaging namespace a `QueueMessagingTemplate` can be defined in an XML configuration file.

```
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aws-context="http://www.springframework.org/schema/cloud/aws/context"
  xmlns:aws-messaging="http://www.springframework.org/schema/cloud/aws/messaging"
  xmlns="http://www.springframework.org/schema/beans"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/cloud/aws/context
    http://www.springframework.org/schema/cloud/aws/context/spring-cloud-aws-context.xsd
    http://www.springframework.org/schema/cloud/aws/messaging
    http://www.springframework.org/schema/cloud/aws/messaging/spring-cloud-aws-messaging">

  <aws-context:context-credentials>
    <aws-context:instance-profile-credentials />
  </aws-context:context-credentials>

  <aws-messaging:queue-messaging-template id="queueMessagingTemplate" />

</beans>
```

In this example the messaging namespace handler constructs a new `QueueMessagingTemplate`. The `AmazonSQS` client is automatically created and passed to the template's constructor based on the provided credentials. If the application runs inside a configured CloudFormation stack a `ResourceIdResolver` is passed to the constructor (see [Managing cloud environments](#) for more information about resource name resolution).

Using message converters

In order to facilitate the sending of domain model objects, the `QueueMessagingTemplate` has various send methods that take a Java object as an argument for a message's data content. The overloaded methods `convertAndSend()` and `receiveAndConvert()` in `QueueMessagingTemplate` delegate the conversion process to an instance of the `MessageConverter` interface. This interface defines a simple contract to convert between Java objects and SQS messages. The default implementation `SimpleMessageConverter` simply unwraps the message payload as long as it matches the target type. By using the converter, you and your application code can focus on the business object that is being sent or received via SQS and not be concerned with the details of how it is represented as an SQS message.

As SQS is only able to send `String` payloads the default converter `SimpleMessageConverter` should only be used to send `String`

payloads. For more complex objects a custom converter should be used like the one created by the messaging namespace handler.

It is recommended to use the XML messaging namespace to create `QueueMessagingTemplate` as it will set a more sophisticated `MessageConverter` that converts objects into JSON when Jackson is on the classpath.

```
<aws-messaging:queue-messaging-template id="queueMessagingTemplate" />
```

XML

```
this.queueMessagingTemplate.convertAndSend("queueName", new Person("John", "Doe"));
```

JAVA

In this example a `QueueMessagingTemplate` is created using the messaging namespace. The `convertAndSend` method converts the payload `Person` using the configured `MessageConverter` and sends the message.

5.2.2. Receiving a message

There are two ways for receiving SQS messages, either use the `receive` methods of the `QueueMessagingTemplate` or with annotation-driven listener endpoints. The latter is by far the more convenient way to receive messages.

```
Person person = this.queueMessagingTemplate.receiveAndConvert("queueName", Person.class);
```

JAVA

In this example the `QueueMessagingTemplate` will get one message from the SQS queue and convert it to the target class passed as argument.

5.2.3. Annotation-driven listener endpoints

Annotation-driven listener endpoints are the easiest way for listening on SQS messages. Simply annotate methods with `MessageMapping` and the `QueueMessageHandler` will route the messages to the annotated methods.

```
<aws-messaging:annotation-driven-queue-listener />
```

XML

```
@SqsListener("queueName")
public void queueListener(Person person) {
    // ...
}
```

JAVA

In this example a queue listener container is started that polls the SQS `queueName` passed to the `MessageMapping` annotation. The incoming messages are converted to the target type and then the annotated method `queueListener` is invoked.

In addition to the payload, headers can be injected in the listener methods with the `@Header` or `@Headers` annotations. `@Header` is used to inject a specific header value while `@Headers` injects a `Map<String, String>` containing all headers.

Only the [standard message attributes](#) sent with an SQS message are supported. Custom attributes are currently not supported.

In addition to the provided argument resolvers, custom ones can be registered on the `aws-messaging:annotation-driven-queue-listener` element using the `aws-messaging:argument-resolvers` attribute (see example below).

```
<aws-messaging:annotation-driven-queue-listener>
  <aws-messaging:argument-resolvers>
    <bean class="org.custom.CustomArgumentResolver" />
  </aws-messaging:argument-resolvers>
</aws-messaging:annotation-driven-queue-listener>
```

XML

By default the `SimpleMessageListenerContainer` creates a `ThreadPoolTaskExecutor` with computed values for the core and max pool sizes. The core pool size is set to twice the number of queues and the max pool size is obtained by multiplying the number of queues by the value of the `maxNumberOfMessages` field. If these default values do not meet the need of the application, a custom task executor can be set with the `task-executor` attribute (see example below).

```
<aws-messaging:annotation-driven-queue-listener task-executor="simpleTaskExecutor" />
```

XML

Message reply

Message listener methods can be annotated with `@SendTo` to send their return value to another channel. The `SendToHandlerMethodReturnValueHandler` uses the defined messaging template set on the `aws-messaging:annotation-driven-queue-listener` element to send the return value. The messaging template must implement the `DestinationResolvingMessageSendingOperations` interface.

```
<aws-messaging:annotation-driven-queue-listener send-to-message-template="queueMessagingTemplate"/>
```

XML

```
@SqsListener("treeQueue")
@SendTo("leafsQueue")
public List<Leaf> extractLeafs(Tree tree) {
    // ...
}
```

JAVA

In this example the `extractLeafs` method will receive messages coming from the `treeQueue` and then return a `List` of `Leaf`'s which is going to be sent to the `leafsQueue`. Note that on the `aws-messaging:annotation-driven-queue-listener` XML element there is an attribute `send-to-message-template` that specifies `QueueMessagingTemplate` as the messaging template to be used to send the return value of the message listener method.

5.2.4. The SimpleMessageListenerContainerFactory

The `SimpleMessageListenerContainer` can also be configured with Java by creating a bean of type `SimpleMessageListenerContainerFactory`.

```

@Bean
public SimpleMessageListenerContainerFactory simpleMessageListenerContainerFactory(AmazonSQSAsync amazonSqs) {
    SimpleMessageListenerContainerFactory factory = new SimpleMessageListenerContainerFactory();
    factory.setAmazonSqs(amazonSqs);
    factory.setAutoStartup(false);
    factory.setMaxNumberOfMessages(5);
    // ...

    return factory;
}

```

5.2.5. Consuming AWS Event messages with Amazon SQS

It is also possible to receive AWS generated event messages with the SQS message listeners. Because AWS messages does not contain the mime-type header, the Jackson message converter has to be configured with the `strictContentTypeMatch` property false to also parse message without the proper mime type.

The next code shows the configuration of the message converter using the `QueueMessageHandlerFactory` and re-configuring the `MappingJackson2MessageConverter`

```

@Bean
public QueueMessageHandlerFactory queueMessageHandlerFactory() {
    QueueMessageHandlerFactory factory = new QueueMessageHandlerFactory();
    MappingJackson2MessageConverter messageConverter = new MappingJackson2MessageConverter();

    //set strict content type match to false
    messageConverter.setStrictContentTypeMatch(false);
    factory.setArgumentResolvers(Collections.<HandlerMethodArgumentResolver>singletonList(new PayloadArgumentResolver(messageConverter)));
    return factory;
}

```

With the configuration above, it is possible to receive event notification for S3 buckets (and also other event notifications like elastic transcoder messages) inside `@SqsListener` annotated methods s shown below.

```

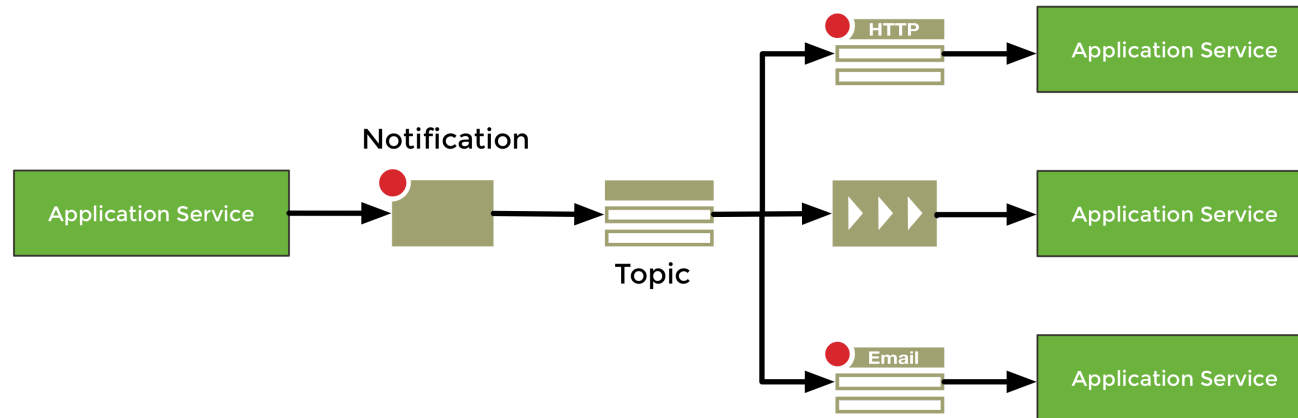
@SqsListener("testQueue")
public void receive(S3EventNotification s3EventNotificationRecord) {
    S3EventNotification.S3Entity s3Entity = s3EventNotificationRecord.getRecords().get(0).getS3();
}

```

5.3. SNS support

Amazon SNS is a publish-subscribe messaging system that allows clients to publish notification to a particular topic. Other interested clients may subscribe using different protocols like HTTP/HTTPS, e-mail or an Amazon SQS queue to receive the messages.

The next graphic shows a typical example of an Amazon SNS architecture.



Spring Cloud AWS supports Amazon SNS by providing support to send notifications with a `NotificationMessagingTemplate` and to receive notifications with the HTTP/HTTPS endpoint using the Spring Web MVC `@Controller` based programming model. Amazon SQS based subscriptions can be used with the annotation-driven message support that is provided by the Spring Cloud AWS messaging module.

5.3.1. Sending a message

The `NotificationMessagingTemplate` contains two convenience methods to send a notification. The first one specifies the destination using a `String` which is going to be resolved against the SNS API. The second one takes no destination argument and uses the default destination. All the usual send methods that are available on the `MessageSendingOperations` are implemented but are less convenient to send notifications because the subject must be passed as header.

Currently only `String` payloads can be sent using the `NotificationMessagingTemplate` as this is the expected type by the SNS API.

```
import com.amazonaws.services.sns.AmazonSNS;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.aws.messaging.core.NotificationMessagingTemplate;

public class SnsNotificationSender {

    private final NotificationMessagingTemplate notificationMessagingTemplate;

    @Autowired
    public SnsNotificationSender(AmazonSNS amazonSns) {
        this.notificationMessagingTemplate = new NotificationMessagingTemplate(amazonSns);
    }
}
```

```

public void send(String subject, String message) {
    this.notificationMessagingTemplate.sendNotification("physicalTopicName", message, subject);
}
}

```

This example constructs a new `NotificationMessagingTemplate` by passing an `AmazonSNS` client as argument. In the `send` method the convenience `sendNotification` method is used to send a `message` with `subject` to an SNS topic. The destination in the `sendNotification` method is a string value that must match the topic name defined on AWS. This value is resolved at runtime by the Amazon SNS client. Optionally a `ResourceIdResolver` implementation can be passed to the `NotificationMessagingTemplate` constructor to resolve resources by logical name when running inside a CloudFormation stack. (See [Managing cloud environments](#) for more information about resource name resolution.)

It is recommended to use the XML messaging namespace to create `NotificationMessagingTemplate` as it will automatically configure the SNS client to setup the default converter.

```

<aws-messaging:notification-messaging-template id="notificationMessagingTemplate" />

```

XML

5.3.2. Annotation-driven HTTP notification endpoint

SNS supports multiple endpoint types (SQS, Email, HTTP, HTTPS), Spring Cloud AWS provides support for HTTP(S) endpoints. SNS sends three type of requests to an HTTP topic listener endpoint, for each of them annotations are provided:

- Subscription request → `@NotificationSubscriptionMapping`
- Notification request → `@NotificationMessageMapping`
- Unsubscription request → `@NotificationUnsubscribeMapping`

HTTP endpoints are based on Spring MVC controllers. Spring Cloud AWS added some custom argument resolvers to extract the message and subject out of the notification requests.

```

@Controller
@RequestMapping("/topicName")
public class NotificationTestController {

    @NotificationSubscriptionMapping
    public void handleSubscriptionMessage(NotificationStatus status) throws IOException {
        //We subscribe to start receive the message
        status.confirmSubscription();
    }

    @NotificationMessageMapping
    public void handleNotificationMessage(@NotificationSubject String subject, @NotificationMessage String message) {
        // ...
    }

    @NotificationUnsubscribeConfirmationMapping
    public void handleUnsubscribeMessage(NotificationStatus status) {
        //e.g. the client has been unsubscribed and we want to "re-subscribe"
        status.confirmSubscription();
    }
}

```

JAVA

```
}
}
```

Currently it is not possible to define the mapping URL on the method level therefore the `RequestMapping` must be done at type level and must contain the full path of the endpoint.

This example creates a new Spring MVC controller with three methods to handle the three requests listed above. In order to resolve the arguments of the `handleNotificationMessage` methods a custom argument resolver must be registered. The XML configuration is listed below.

```
<mvc:annotation-driven>
  <mvc:argument-resolvers>
    <ref bean="notificationResolver" />
  </mvc:argument-resolvers>
</mvc:annotation-driven>

<aws-messaging:notification-argument-resolver id="notificationResolver" />
```

The `aws-messaging:notification-argument-resolver` element registers three argument resolvers: `NotificationStatusHandlerMethodArgumentResolver`, `NotificationMessageHandlerMethodArgumentResolver`, and `NotificationSubjectHandlerMethodArgumentResolver`.

5.4. Using CloudFormation

Amazon SQS queues and SNS topics can be configured within a stack and then be used by applications. Spring Cloud AWS also supports the lookup of stack-configured queues and topics by their logical name with the resolution to the physical name. The example below shows an SNS topic and SQS queue configuration inside a CloudFormation template.

```
"LogicalQueueName": {
  "Type": "AWS::SQS::Queue",
  "Properties": {
  }
},
"LogicalTopicName": {
  "Type": "AWS::SNS::Topic",
  "Properties": {
  }
}
```

The logical names `LogicalQueueName` and `LogicalTopicName` can then be used in the configuration and in the application as shown below:

```
<aws-messaging:queue-messaging-template default-destination="LogicalQueueName" />
```

```
<aws-messaging:notification-messaging-template default-destination="LogicalTopicName" />
```

```
@SqsListener("LogicalQueueName")
public void receiveQueueMessages(Person person) {
    // Logical names can also be used with messaging templates
    this.notificationMessagingTemplate.sendNotification("anotherLogicalTopicName", "Message", "Subject");
}
```

JAVA

When using the logical names like in the example above, the stack can be created on different environments without any configuration or code changes inside the application.

6. Caching

Caching in a cloud environment is useful for applications to reduce the latency and to save database round trips. Reducing database round trips can significantly reduce the requirements for the database instance. The Spring Framework provides, since version 3.1, a unified Cache abstraction to allow declarative caching in applications analogous to the declarative transactions.

Spring Cloud AWS integrates the [Amazon ElastiCache](#) service into the Spring unified caching abstraction providing a cache manager based on the memcached and Redis protocols. The caching support for Spring Cloud AWS provides its own memcached implementation for ElastiCache and uses [Spring Data Redis](#) for Redis caches.

6.1. Configuring dependencies for Redis caches

Spring Cloud AWS delivers its own implementation of a memcached cache, therefore no other dependencies are needed. For Redis Spring Cloud AWS relies on Spring Data Redis to support caching and also to allow multiple Redis drivers to be used. Spring Cloud AWS supports all Redis drivers that Spring Data Redis supports (currently Jedis, JRedis, SRP and Lettuce) with Jedis being used internally for testing against ElastiCache. A dependency definition for Redis with Jedis is shown in the example

```
<dependencies>
  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-redis</artifactId>
    <version>${spring-data-redis.version}</version>
  </dependency>
  <dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.6.1</version>
  </dependency>
</dependencies>
```

XML

Spring Cloud AWS will automatically detect the Redis driver and will use one of them automatically.

6.2. Configuring caching with XML

The cache support for Spring Cloud AWS resides in the context module and can therefore be used if the context module is already imported in the project. The cache integration provides its own namespace to configure cache clusters that are hosted in the Amazon ElastiCache service. The next example contains a configuration for the cache cluster and the Spring configuration to enable declarative, annotation-based caching.

```
<beans xmlns:aws-cache="http://www.springframework.org/schema/cloud/aws/cache"
  xmlns:cache="http://www.springframework.org/schema/cache"
  xmlns="http://www.springframework.org/schema/beans"
  xsi:schemaLocation="http://www.springframework.org/schema/cloud/aws/cache
    http://www.springframework.org/schema/cloud/aws/cache/spring-cloud-aws-cache.xsd
    http://www.springframework.org/schema/cache
    https://www.springframework.org/schema/cache/spring-cache.xsd">

  <aws-context:context-credentials>
    ...
  </aws-context:context-credentials>

  <aws-cache:cache-manager>
    <aws-cache:cache-cluster name="CacheCluster" />
  </aws-cache:cache-manager>

  <cache:annotation-driven />
</beans>
```

The configuration above configures a `cache-manager` with one cache with the name `CacheCluster` that represents an [ElasticCache cluster](#).

6.2.1. Mixing caches

Applications may have the need for multiple caches that are maintained by one central cache cluster. The Spring Cloud AWS caching support allows to define multiple caches inside one cache manager and also to use externally defined caches inside the cache manager.

The example below demonstrates a configuration example that contains a pre-configured cache with a `cache-ref` element (which might be a local cache) and a `cache-cluster` configuration for ElastiCache cache clusters.

```
<beans ...>
  <aws-cache:cache-manager id="cacheManager">
    <aws-cache:cache-ref ref="memcached" />
    <aws-cache:cache-cluster name="SimpleCache" />
  </aws-cache:cache-manager>
</beans>
```

6.2.2. Defining expiration

The Spring cache demarcation does not support expiry time configuration and leaves it up to the cache implementation to support an expiry time. The Spring Cloud AWS cache configuration supports the expiry time setting per cache. The expiry time will be passed to the memcached service.

The `cache-cluster` element accepts an expiration attribute that defines the expiration time in seconds. No configured values implies that there is an infinite expiration time.

```
<beans>
  <aws-cache:cache-manager>
    <aws-cache:cache-cluster expiration="10000" name="CacheCluster" />
  </aws-cache:cache-manager>
</beans>
```

6.3. Configuring caching using Java configuration

Spring Cloud AWS also support the cache configuration with Java configuration classes. On any `Configuration` class, the caching can be configured using the `org.springframework.cloud.aws.cache.config.annotation.EnableElasticCache` annotation provided by Spring Cloud AWS. The next example shows a configuration of two cache clusters.

```
@EnableElasticCache({@CacheClusterConfig(name = "firstCache"), @CacheClusterConfig(name = "secondCache")})
public class ApplicationConfiguration {
}
```

If you leave the `value` attribute empty, then all the caches inside your CloudFormation stack (if available) will be configured automatically.

6.3.1. Configuring expiry time for caches

The Java configuration also allows to configure the expiry time for the caches. This can be done for all caches using the `defaultExpiration` attribute as shown in the example below.

```
@EnableElasticCache(defaultExpiration = 23)
public class ApplicationConfiguration {
}
```

The expiration can be defined on a cache level using the `@CacheClusterConfig` annotations expiration attribute as shown below (using seconds as the value).

```
@EnableElasticCache({@CacheClusterConfig(name = "firstCache", expiration = 23), @CacheClusterConfig(name = "secondCache", expiration = 23)})
public class ApplicationConfiguration {
}
```

6.4. Configuring caching in Spring Boot

The caches will automatically be configured in Spring Boot without any explicit configuration property.

6.5. Using caching

Based on the configuration of the cache, developers can annotate their methods to use the caching for method return values. The next example contains a caching declaration for a service for which the return values should be cached

```
@Service
public class ExpensiveService {

    @Cacheable("CacheCluster")
    public String calculateExpensiveValue(String key) {
        ...
    }
}
```

JAVA

6.6. Memcached client implementation

There are different memcached client implementations available for Java, the most prominent ones are [Spymemcached](#) and [XMemcached](#). Amazon AWS supports a dynamic configuration and delivers an enhanced memcached client based on Spymemcached to support the [auto-discovery](#) of new nodes based on a central configuration endpoint.

Spring Cloud AWS relies on the Amazon ElastiCache Client implementation and therefore has a dependency on that.

6.7. Using CloudFormation

Amazon ElastiCache clusters can also be configured within a stack and then be used by applications. Spring Cloud AWS also supports the lookup of stack-configured cache clusters by their logical name with the resolution to the physical name. The example below shows a cache cluster configuration inside a CloudFormation template.

```
"CacheCluster": {
  "Type": "AWS::ElastiCache::CacheCluster",
  "Properties": {
    "AutoMinorVersionUpgrade": "true",
    "Engine": "memcached",
    "CacheNodeType": "cache.t2.micro",
    "CacheSubnetGroupName": "sample",
    "NumCacheNodes": "1",
    "VpcSecurityGroupIds": ["sample1"]
  }
}
```

JSON

The cache cluster can then be used with the name `CacheCluster` inside the application configuration as shown below:

```
<beans...>
  <aws-cache:cache-manager>
    <aws-cache:cache-cluster name="CacheCluster" expiration="15"/>
  </aws-cache:cache-manager>
</beans>
```

With the configuration above the application can be deployed with multiple stacks on different environments without any configuration change inside the application.

7. Data Access with JDBC

Spring has a broad support of data access technologies built on top of JDBC like `JdbcTemplate` and dedicated ORM (JPA, Hibernate support). Spring Cloud AWS enables application developers to re-use their JDBC technology of choice and access the [Amazon Relational Database Service](#) with a declarative configuration. The main support provided by Spring Cloud AWS for JDBC data access are:

- Automatic data source configuration and setup based on the Amazon RDS database instance.
- Automatic read-replica detection and configuration for Amazon RDS database instances.
- Retry-support to handle exception during Multi-AZ failover inside the data center.

7.1. Configuring data source

Before using and configuring the database support, the application has to include the respective module dependency into its Maven configuration. Spring Cloud AWS JDBC support comes as a separate module to allow the modularized use of the modules.

7.1.1. Maven dependency configuration

The Spring Cloud AWS JDBC module comes as a standalone module and can be imported with the following dependency declaration.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-aws-jdbc</artifactId>
  <version>{spring-cloud-version}</version>
</dependency>
```

7.1.2. Basic data source configuration

The data source configuration requires the security and region configuration as a minimum allowing Spring Cloud AWS to retrieve the database metadata information with the Amazon RDS service. Spring Cloud AWS provides an additional `jdbc` specific namespace to configure the data source with the minimum attributes as shown in the example:

```

<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jdbc="http://www.springframework.org/schema/cloud/aws/jdbc"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/cloud/aws/jdbc
       http://www.springframework.org/schema/cloud/aws/jdbc/spring-cloud-aws-jdbc.xsd">

    <aws-context:context-credentials>
        ...
    </aws-context:context-credentials>

    <aws-context:context-region region="..." />

    <jdbc:data-source
        db-instance-identifier="myRdsDatabase"
        password="{rdsPassword}">
    </jdbc:data-source>
</beans>

```

The minimum configuration parameters are a unique `id` for the data source, a valid `db-instance-identifier` attribute that points to a valid Amazon RDS database instance. The master user password for the master user. If there is another user to be used (which is recommended) then the `username` attribute can be set.

With this configuration Spring Cloud AWS fetches all the necessary metadata and creates a [Tomcat JDBC pool](#) with the default properties. The data source can be later injected into any Spring Bean as shown below:

```

@Service
public class SimpleDatabaseService implements DatabaseService {

    private final JdbcTemplate jdbcTemplate;

    @Autowired
    public SimpleDatabaseService(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }
}

```

It is possible to qualify the data source injection point with an `@Qualifier` annotation to allow multiple data source configurations inside one application context and still use auto-wiring.

7.1.3. Data source pool configuration

Spring Cloud AWS creates a new Tomcat JDBC pool with the default properties. Often these default properties do not meet the requirements of the application with regards to pool size and other settings. The data source configuration supports the configuration of all valid pool properties with a nested XML element. The following example demonstrates the re-configuration of the data source with custom pool properties.

```

<beans ..>

    <aws-context:context-credentials>
        ...
    </aws-context:context-credentials>

```

```
<aws-context:context-region region="..." />

<jdbc:data-source
  db-instance-identifier="myRdsDatabase"
  password="${rdsPassword}">
  <jdbc:pool-attributes initialSize="1" " maxActive="200" minIdle="10"
    testOnBorrow="true" validationQuery="SELECT 1" />
</jdbc:data-source>

</beans>
```

A full list of all configuration attributes with their value is available [here](#).

7.2. Configuring data source with Java config

Spring Cloud AWS also supports the configuration of the data source within an `@Configuration` class. The `org.springframework.cloud.aws.jdbc.config.annotation.EnableRdsInstance` annotation can be used to configure one data source. Multiple ones can be used to configure more than one data source. Each annotation will generate exactly one data source bean.

The class below shows a data source configuration inside a configuration class

```
@Configuration
@EnableRdsInstance(dbInstanceIdentifier = "test", password = "secret", readReplicaSupport = true)
public class ApplicationConfiguration {
}
```

The configuration attributes are the same in the XML element. The required attributes are also the same for the XML configuration (the `dbInstanceIdentifier` and `password` attribute)

7.2.1. Java based data source pool configuration

It is also possible to override the pool configuration with custom values. Spring Cloud AWS provides a `org.springframework.cloud.aws.jdbc.config.annotation.RdsInstanceConfigurer` that creates a `org.springframework.cloud.aws.jdbc.datasource.DataSourceFactory` which might contain custom pool attributes. The next examples shows the implementation of one configurer that overrides the validation query and the initial size.

```
@Configuration
@EnableRdsInstance(dbInstanceIdentifier = "test", password = "secret")
public class ApplicationConfiguration {

    @Bean
    public RdsInstanceConfigurer instanceConfigurer() {
        return new RdsInstanceConfigurer() {
            @Override
            public DataSourceFactory getDataSourceFactory() {
                TomcatJdbcDataSourceFactory dataSourceFactory = new TomcatJdbcDataSourceFactory();
```

```
        dataSourceFactory.setInitialSize(10);
        dataSourceFactory.setValidationQuery("SELECT 1 FROM DUAL");
        return dataSourceFactory;
    }
};
}
```

This class returns an anonymous class of type `org.springframework.cloud.aws.jdbc.config.annotation.RdsInstanceConfigurer`, which might also of course be a standalone class.

7.3. Configuring data source in Spring Boot

The data sources can also be configured using the Spring Boot configuration files. Because of the dynamic number of data sources inside one application, the Spring Boot properties must be configured for each data source.

A data source configuration consists of the general property name `cloud.aws.rds.<instanceName>` for the data source identifier following the sub properties for the particular data source where `instanceName` is the name of the concrete instance. The table below outlines all properties for a data source using `test` as the instance identifier.

property	example	description
cloud.aws.rds.test		The configuration property that configures a data source with the name test
cloud.aws.rds.test.password	verySecret	The password for the db instance test
cloud.aws.rds.test.username	admin	The username for the db instance test (optional)
cloud.aws.rds.test.readReplicaSupport	true	If read-replicas should be used for the data source (see below)
cloud.aws.rds.test.databaseName	fooDb	Custom database name if the default one from rds should not be used

7.4. Read-replica configuration

Amazon RDS allows to use [MySQL read-replica](#) instances to increase the overall throughput of the database by offloading read data access to one or more read-replica slaves while maintaining the data in one master database.

Spring Cloud AWS supports the use of read-replicas in combination with Spring read-only transactions. If the read-replica support is enabled, any read-only transaction will be routed to a read-replica instance while using the master database for write operations.

Using read-replica instances does not guarantee strict [ACID](#) semantics for the database access and should be used with care. This is due to the fact that the read-replica might be behind and a write might not be immediately visible to the read transaction. Therefore it is recommended to use read-replica instances only for transactions that read data which is not changed very often and where outdated data can be handled by the application.

The read-replica support can be enabled with the `read-replica` attribute in the datasource configuration.

```
<beans ...>
<jdbc:data-source db-instance-identifier="RdsSingleMicroInstance"
  password="${rdsPassword}" read-replica-support="true">

</jdbc:data-source>
</beans>
```

Spring Cloud AWS will search for any read-replica that is created for the master database and route the read-only transactions to one of the read-replicas that are available. A business service that uses read-replicas can be implemented like shown in the example.

```
@Service
public class SimpleDatabaseService {

    private final JdbcTemplate jdbcTemplate;

    @Autowired
    public SimpleDatabaseService(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    @Transactional(readOnly = true)
    public Person loadAll() {
        // read data on the read replica
    }

    @Transactional
    public void updatePerson(Person person) {
        // write data into database
    }
}
```

7.5. Failover support

Amazon RDS supports a [Multi-AZ](#) fail-over if one availability zone is not available due to an outage or failure of the primary instance. The replication is synchronous (compared to the read-replicas) and provides continuous service. Spring Cloud AWS supports a

Multi-AZ failover with a retry mechanism to recover transactions that fail during a Multi-AZ failover.

In most cases it is better to provide direct feedback to a user instead of trying potentially long and frequent retries within a user interaction. Therefore the fail-over support is primarily useful for batch application or applications where the responsiveness of a service call is not critical.

The Spring Cloud AWS JDBC module provides a retry interceptor that can be used to decorate services with an interceptor. The interceptor will retry the database operation again if there is a temporary error due to a Multi-AZ failover. A Multi-AZ failover typically lasts only a couple of seconds, therefore a retry of the business transaction will likely succeed.

The interceptor can be configured as a regular bean and then be used by a pointcut expression to decorate the respective method calls with the interceptor. The interceptor must have a configured database to retrieve the current status (if it is a temporary fail-over or a permanent error) from the Amazon RDS service.

The configuration for the interceptor can be done with a custom element from the Spring Cloud AWS jdbc namespace and will be configured like shown:

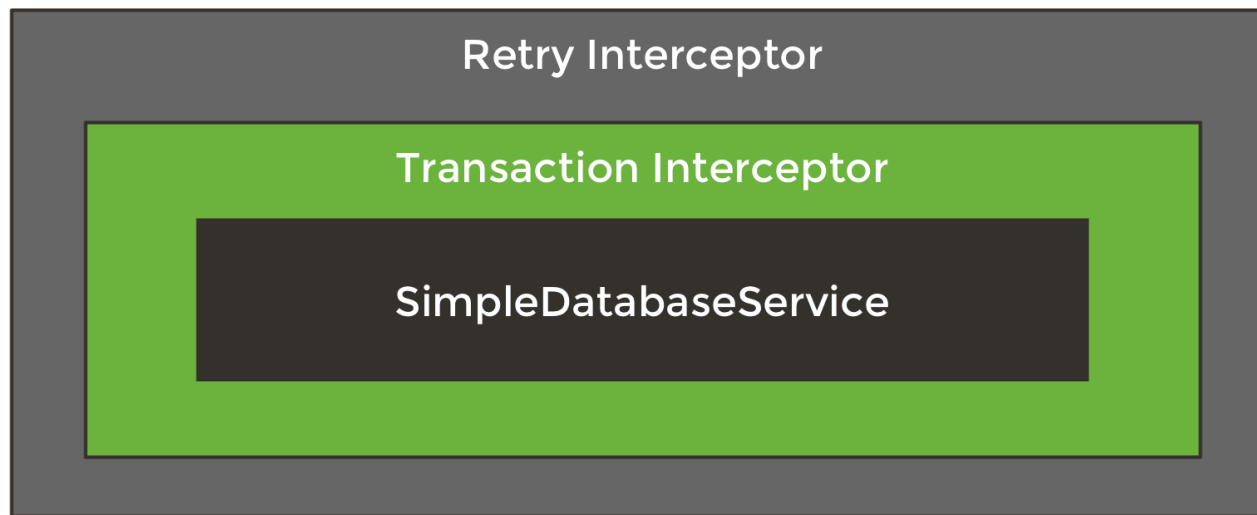
```
<beans ..>
  <jdbc:retry-interceptor id="myInterceptor"
    db-instance-identifier="myRdsDatabase"
    max-number-of-retries="10" />
</beans>
```

The interceptor itself can be used with any Spring advice configuration to wrap the respective service. A pointcut for the services shown in the chapter before can be defined as follows:

```
<beans ..>
  <aop:config>
    <aop:advisor advice-ref="myInterceptor" pointcut="bean(simpleDatabaseService)" order="1" />
  </aop:config>
</beans>
```

It is important that the interceptor is called outside the transaction interceptor to ensure that the whole transaction will be re-executed. Configuring the interceptor inside the transaction interceptor will lead to a permanent error because the broken connection will never be refreshed.

The configuration above in combination with a transaction configuration will produce the following proxy configuration for the service.



7.6. CloudFormation support

Spring Cloud AWS supports database instances that are configured with CloudFormation. Spring Cloud AWS can use the logical name inside the database configuration and lookup the concrete database with the generated physical resource name. A database configuration can be easily configured in CloudFormation with a template definition that might look like the following example.

```
"myRdsDatabase": {
  "Type": "AWS::RDS::DBInstance",
  "Properties": {
    "AllocatedStorage": "5",
    "DBInstanceClass": "db.t1.micro",
    "DBName": "test",
    "Engine": "mysql",
    "MasterUsername": "admin",
    "MasterUserPassword": {"Ref": "RdsPassword"},
    ...
  }
},
"readReplicaDatabase": {
  "Type": "AWS::RDS::DBInstance",
  "Properties": {
    "AllocatedStorage": "5",
    "SourceDBInstanceIdentifier": {
      "Ref": "myRdsDatabase"
    },
    "DBInstanceClass": "db.t1.micro"
  }
}
```

The database can then be configured using the name set in the template. Also, the read-replica can be enabled to use the configured read-replica database in the application. A configuration to use the configured database is outlined below:

```
<beans>
  <aws-context:stack-configuration/>

  <jdbc:data-source db-instance-identifier="myRdsDatabase" password="${rdsPassword}" read-replica-support="true"/>
</beans>
```

7.7. Database tags

Amazon RDS instances can also be configured using RDS database specific tags, allowing users to configure database specific configuration metadata with the database. Database instance specific tags can be configured using the `user-tags-map` attribute on the `data-source` element. Configure the tags support like in the example below:

```
<jdbc:data-source
  db-instance-identifier="myRdsDatabase"
  password="${rdsPassword}" user-tags-map="dbTags" />
```

That allows the developer to access the properties in the code using expressions like shown in the class below:

```
public class SampleService {

    @Value("#{dbTags['aws:cloudformation:aws:cloudformation:stack-name']}")
    private String stackName;
}
```

The database tag `aws:cloudformation:aws:cloudformation:stack-name` is a default tag that is created if the database is configured using CloudFormation.

8. Sending mails

Spring has a built-in support to send e-mails based on the [Java Mail API](#) to avoid any static method calls while using the Java Mail API and thus supporting the testability of an application. Spring Cloud AWS supports the [Amazon SES](#) as an implementation of the Spring Mail abstraction.

As a result Spring Cloud AWS users can decide to use the Spring Cloud AWS implementation of the Amazon SES service or use the standard Java Mail API based implementation that sends e-mails via SMTP to Amazon SES.

It is preferred to use the Spring Cloud AWS implementation instead of SMTP mainly for performance reasons. Spring Cloud AWS uses one API call to send a mail message, while the SMTP protocol makes multiple requests (EHLO, MAIL FROM, RCPT TO, DATA, QUIT) until it sends an e-mail.

8.1. Configuring the mail sender

Spring Cloud AWS provides an XML element to configure a Spring `org.springframework.mail.MailSender` implementation for the client to be used. The default mail sender works without a Java Mail dependency and is capable of sending messages without attachments as simple mail messages. A configuration with the necessary elements will look like this:

```
<beans xmlns:aws-mail="http://www.springframework.org/schema/cloud/aws/mail"
      xsi:schemaLocation="http://www.springframework.org/schema/cloud/aws/mail
      http://www.springframework.org/schema/cloud/aws/mail/spring-cloud-aws-mail.xsd">

  <aws-context:context-credentials>
    ..
  </aws-context:context-credentials>

  <aws-context:context-region region="eu-west-1" />

  <aws-mail:mail-sender id="testSender" />

</beans>
```

8.2. Sending simple mails

Application developers can inject the `MailSender` into their application code and directly send simple text based e-mail messages. The sample below demonstrates the creation of a simple mail message.

```
public class MailSendingService {

  private MailSender mailSender;

  @Autowired
  public MailSendingService(MailSender mailSender) {
    this.mailSender = mailSender;
  }

  public void sendMailMessage() {
    SimpleMailMessage simpleMailMessage = new SimpleMailMessage();
    simpleMailMessage.setFrom("foo@bar.com");
    simpleMailMessage.setTo("bar@baz.com");
    simpleMailMessage.setSubject("test subject");
    simpleMailMessage.setText("test content");
    this.mailSender.send(simpleMailMessage);
  }
}
```

8.3. Sending attachments

Sending attachments with e-mail requires MIME messages to be created and sent. In order to create MIME messages, the Java Mail dependency is required and has to be included in the classpath. Spring Cloud AWS will detect the dependency and create a `org.springframework.mail.javamail.JavaMailSender` implementation that allows to create and build MIME messages and send them. A dependency configuration for the Java Mail API is the only change in the configuration which is shown below.

```
<dependency>
  <groupId>javax.mail</groupId>
  <artifactId>mailapi</artifactId>
  <version>1.4.1</version>
  <exclusions>
    <!-- exclusion because we are running on Java 1.7 that includes the activation API by default-->
    <exclusion>
      <artifactId>activation</artifactId>
      <groupId>javax.activation</groupId>
    </exclusion>
  </exclusions>
</dependency>
```

Even though there is a dependency to the Java Mail API there is still the Amazon SES API used underneath to send mail messages. There is no [SMTP setup](#) required on the Amazon AWS side.

Sending the mail requires the application developer to use the `JavaMailSender` to send an e-mail as shown in the example below.

```
public class MailSendingService {

    private JavaMailSender mailSender;

    @Autowired
    public MailSendingService(JavaMailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void sendMailMessage() {
        this.mailSender.send(new MimeMessagePreparator() {

            @Override
            public void prepare(MimeMessage mimeMessage) throws Exception {
                MimeMessageHelper helper =
                    new MimeMessageHelper(mimeMessage, true, "UTF-8");
                helper.addTo("foo@bar.com");
                helper.setFrom("bar@baz.com");
                helper.addAttachment("test.txt", ...);
                helper.setSubject("test subject with attachment");
                helper.setText("mime body", false);
            }
        });
    }
}
```

8.4. Configuring regions

Amazon SES is not available in all [regions](#) of the Amazon Web Services cloud. Therefore an application hosted and operated in a region that does not support the mail service will produce an error while using the mail service. Therefore the region must be overridden for the mail sender configuration. The example below shows a typical combination of a region (EU-CENTRAL-1) that does not provide an SES service where the client is overridden to use a valid region (EU-WEST-1).

```
<beans ...>

  <aws-context:context-region region="eu-central-1" />
  <aws-mail:mail-sender id="testSender" region="eu-west-1"/>

</beans>
```

8.5. Authenticating e-mails

To avoid any spam attacks on the Amazon SES mail service, applications without production access must [verify](#) each e-mail receiver otherwise the mail sender will throw a `com.amazonaws.services.simpleemail.model.MessageRejectedException`.

[Production access](#) can be requested and will disable the need for mail address verification.

9. Resource handling

The Spring Framework provides a `org.springframework.core.io.ResourceLoader` abstraction to load files from the filesystem, servlet context and the classpath. Spring Cloud AWS adds support for the [Amazon S3](#) service to load and write resources with the resource loader and the `s3` protocol.

The resource loader is part of the context module, therefore no additional dependencies are necessary to use the resource handling support.

9.1. Configuring the resource loader

Spring Cloud AWS does not modify the default resource loader unless it encounters an explicit configuration with an XML namespace element. The configuration consists of one element for the whole application context that is shown below:

```
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aws-context="http://www.springframework.org/schema/cloud/aws/context"
  xsi:schemaLocation="http://www.springframework.org/schema/cloud/aws/context
  http://www.springframework.org/schema/cloud/aws/context/spring-cloud-aws-context.xsd">

  <aws-context:context-credentials>
    ...
  </aws-context:context-credentials>
```

```
<aws-context:context-resource-loader/>
</beans>
```

9.2. Downloading files

Downloading files can be done by using the `s3` protocol to reference Amazon S3 buckets and objects inside their bucket. The typical pattern is `s3://<bucket>/<object>` where bucket is the global and unique bucket name and object is a valid object name inside the bucket. The object name can be a file in the *root* folder of a bucket or a nested file within a directory inside a bucket.

The next example demonstrates the use of the resource loader to load different resources.

```
public class SimpleResourceLoadingBean {

    @Autowired
    private ResourceLoader resourceLoader;

    public void resourceLoadingMethod() throws IOException {
        Resource resource = this.resourceLoader.getResource("s3://myBucket/rootFile.log");
        Resource secondResource = this.resourceLoader.getResource("s3://myBucket/rootFolder/subFile");

        InputStream inputStream = resource.getInputStream();
        //read file
    }
}
```

9.3. Uploading files

Since Spring Framework 3.1 the resource loader can also be used to upload files with the

`org.springframework.core.io.WritableResource` interface which is a specialization of the `org.springframework.core.io.ResourceLoader` interface. Clients can upload files using the `WritableResource` interface. The next example demonstrates an upload of a resource using the resource loader.

```
public class SimpleResourceLoadingBean {

    @Autowired
    private ResourceLoader resourceLoader;

    public void writeResource() throws IOException {
        Resource resource = this.resourceLoader.getResource("s3://myBucket/rootFile.log");
        WritableResource writableResource = (WritableResource) resource;
        try (OutputStream outputStream = writableResource.getOutputStream()) {
            outputStream.write("test".getBytes());
        }
    }
}
```

9.3.1. Uploading multi-part files

Amazon S3 supports [multi-part uploads](#) to increase the general throughput while uploading. Spring Cloud AWS by default only uses one thread to upload the files and therefore does not provide parallel upload support. Users can configure a custom `org.springframework.core.task.TaskExecutor` for the resource loader. The resource loader will queue multiple threads at the same time to use parallel multi-part uploads.

The configuration for a resource loader that uploads with 10 Threads looks like the following

```
<beans ...>
  <aws-context:context-resource-loader task-executor="executor" />
  <task:executor id="executor" pool-size="10" queue-capacity="0" rejection-policy="CALLER_RUNS" />
</beans>
```

Spring Cloud AWS consumes up to 5 MB (at a minimum) of memory per thread. Therefore each parallel thread will incur a memory footprint of 5 MB in the heap, and a thread size of 10 will consume therefore up to 50 mb of heap space. Spring Cloud AWS releases the memory as soon as possible. Also, the example above shows that there is no `queue-capacity` configured, because queued requests would also consume memory.

9.3.2. Uploading with the TransferManager

The Amazon SDK also provides a high-level abstraction that is useful to upload files, also with multiple threads using the multi-part functionality. A `com.amazonaws.services.s3.transfer.TransferManager` can be easily created in the application code and injected with the pre-configured `com.amazonaws.services.s3.AmazonS3` client that is already created with the Spring Cloud AWS resource loader configuration.

This example shows the use of the `transferManager` within an application to upload files from the hard-drive.

```
public class SimpleResourceLoadingBean {
    @Autowired
    private AmazonS3 amazonS3;

    public void withTransferManager() {
        TransferManager transferManager = new TransferManager(this.amazonS3);
        transferManager.upload("myBucket", "filename", new File("someFile"));
    }
}
```

9.4. Searching resources

The Spring resource loader also supports collecting resources based on an Ant-style path specification. Spring Cloud AWS offers the same support to resolve resources within a bucket and even throughout buckets. The actual resource loader needs to be wrapped

with the Spring Cloud AWS one in order to search for s3 buckets, in case of non s3 bucket the resource loader will fall back to the original one. The next example shows the resource resolution by using different patterns.

```
public class SimpleResourceLoadingBean {

    private ResourcePatternResolver resourcePatternResolver;

    @Autowired
    public void setupResolver(ApplicationContext applicationContext, AmazonS3 amazonS3){
        this.resourcePatternResolver = new PathMatchingSimpleStorageResourcePatternResolver(amazonS3, applicationContext);
    }

    public void resolveAndLoad() throws IOException {
        Resource[] allTxtFilesInFolder = this.resourcePatternResolver.getResources("s3://bucket/name/*.txt");
        Resource[] allTxtFilesInBucket = this.resourcePatternResolver.getResources("s3://bucket/**/*.txt");
        Resource[] allTxtFilesGlobally = this.resourcePatternResolver.getResources("s3://**/*.txt");
    }
}
```

Resolving resources throughout all buckets can be very time consuming depending on the number of buckets a user owns.

9.5. Using CloudFormation

CloudFormation also allows to create buckets during stack creation. These buckets will typically have a generated name that must be used as the bucket name. In order to allow application developers to define *static* names inside their configuration, Spring Cloud AWS provides support to resolve the generated bucket names. Application developers can use the `org.springframework.cloud.aws.core.env.ResourceIdResolver` interface to resolve the physical names that are generated based on the logical names.

The next example shows a bucket definition inside a CloudFormation stack template. The bucket will be created with a name like *integrationteststack-sampleBucket-23qysfs62tc2*

```
{
  "Resources": {
    "sampleBucket": {
      "Type": "AWS::S3::Bucket"
    }
  }
}
```

Application developers can resolve that name and use it to load resources as shown in the next example below.

```
public class SimpleResourceLoadingBean {

    private final ResourceLoader loader;
    private final ResourceIdResolver idResolver;
```

```
@Autowired
public SimpleResourceLoadingBean(ResourceLoader loader, ResourceIdResolver idResolver) {
    this.loader = loader;
    this.idResolver = idResolver;
}

public void resolveAndLoad() {
    String sampleBucketName = this.idResolver.
        resolveToPhysicalResourceId("sampleBucket");
    Resource resource = this.loader.
        getResource("s3://" + sampleBucketName + "/test");
}
}
```