

Spring Cloud App Broker

Version 1.1.0.RC2

Table of Contents

1. Introduction	2
2. Getting Started	3
2.1. Maven Dependencies	3
2.2. Gradle Dependencies	3
2.3. Configuring the Service Broker	3
3. Advertising Services	5
4. Service Instances	6
4.1. Configuring App Deployment	6
4.1.1. Static Customization	6
Properties Configuration	6
Environment Configuration	8
Service Configuration	8
4.1.2. Dynamic Customization	9
Backing Application Target	9
Service Instance Parameters	11
Credentials Generation	12
4.2. Creating a Service Instance	14
4.3. Updating a Service Instance	14
4.4. Deleting a Service Instance	15
4.5. Persisting Service Instance State	15
4.5.1. Example Implementation	15
5. Service Bindings	20
5.1. Creating a Service Binding	20
5.2. Deleting a Service Binding	20
5.3. Persisting Service Instance Binding State	20
5.3.1. Example Implementation	20
6. Deployment Platforms	26

Spring Cloud App Broker is a framework for building [Spring Boot](#) applications that implement the [Open Service Broker API](#) and deploy applications as brokered services.

Chapter 1. Introduction

Spring Cloud App Broker builds on [Spring Cloud Open Service Broker](#). It can be used to create a service broker that complies with the Open Service Broker API and deploys applications and backing services to a platform, such as Cloud Foundry or Kubernetes.

A service broker using Spring Cloud App Broker is a Spring Boot application. The broker can deploy applications written in any language supported by the targeted platform.

Chapter 2. Getting Started

To get started, create a Spring Boot application and include the Spring Cloud App Broker dependency in the application's build file.

2.1. Maven Dependencies

If you use Maven, include the following in your application's `pom.xml` file:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-app-broker-cloudfoundry</artifactId>
    <version>1.1.0.RC2</version>
  </dependency>
</dependencies>
```

2.2. Gradle Dependencies

If you use Gradle, include the following in your application's `build.gradle` file:

```
dependencies {
    api 'org.springframework.cloud:spring-cloud-starter-app-broker-
        cloudfoundry:1.1.0.RC2'
}
```

2.3. Configuring the Service Broker

The service broker is configured with Spring Boot externalized configuration, supplied by a YAML or Java Properties file (for example, you can provide configuration in the `application.yml` file). Because Spring Cloud App Broker builds on Spring Cloud Open Service Broker, you must provide Spring Cloud Open Service Broker configuration to use Spring Cloud App Broker.

To do so, include Spring Cloud Open Service Broker configuration using properties under `spring.cloud.openservicebroker` as follows:

```

spring:
  cloud:
    openservicebroker:
      catalog:
        services:
          - name: example
            id: ebca66fd-461d-415b-bba3-5e379d671c88
            description: A useful service
            bindable: true
            tags:
              - example
        plans:
          - name: standard
            id: e19e6bc3-37c1-4478-b70f-c7157ebbb28c
            description: A standard plan
            free: true

```

Then include Spring Cloud App Broker configuration using properties under `spring.cloud.appbroker`, as follows:

```

spring:
  cloud:
    appbroker:
      services:
        - service-name: example
          plan-name: standard
        apps:
          - name: example-service-app1
            path: classpath:app1.jar
          - name: example-service-app2
            path: classpath:app2.jar
      deployer:
        cloudfoundry:
          api-host: api.sys.example.local
          api-port: 443
          username: admin
          password: adminpass
          default-org: test
          default-space: development

```

Chapter 3. Advertising Services

The service broker catalog, through which the broker advertises service offerings, is provided by Spring Cloud Open Service Broker. In App Broker configuration (using properties under `spring.cloud.appbroker.services`), you can list services and their plans. This listing must correspond to the service and service plan listing given to Spring Cloud Open Service Broker (using properties under `spring.cloud.openservicebroker.catalog.services`).

For more information about configuring the broker catalog, see the [Spring Cloud Open Service Broker documentation](#).

Chapter 4. Service Instances

You can configure the details of services, including applications to deploy, application deployment details, and backing services to create, in App Broker configuration properties. These properties are generally under `spring.cloud.appbroker.services`.

4.1. Configuring App Deployment

Deployment details for a backing application can be configured statically in the service broker's application configuration and dynamically by using service instance parameters and customization implementations.

4.1.1. Static Customization

You can statically configure backing application deployment details in the application configuration for the service broker by using properties under `spring.cloud.appbroker`.

Properties Configuration

You can specify application deployment properties in configuration. These properties can have default values and service-specific values.

For Cloud Foundry, you can set default values for all services under `spring.cloud.appbroker.deployer.cloudfoundry.*`, as follows:

```
spring:
  cloud:
    appbroker:
      deployer:
        cloudfoundry:
          <strong>properties:</strong>
          <strong>memory: 1G</strong>
          <strong>health-check: http</strong>
          <strong>health-check-http-endpoint: /health</strong>
          <strong>health-check-timeout: 180</strong>
          <strong>api-polling-timeout: 300</strong>
```

You can set overriding values for a specific service in the service's configuration under `spring.cloud.appbroker.services.*`, as follows:


```

spring:
  cloud:
    appbroker:
      services:
        - service-name: example
          plan-name: standard
          apps:
            - name: example-service-app1
              path: classpath:app1.jar
              <strong>properties:</strong>
                <strong>memory: 2G</strong>
                <strong>count: 2</strong>
                <strong>no-route: true</strong>

```

The following table lists properties that can be set for all or for specific application deployments:

Property	Description	Default
count		
memory		
disk		
host		
target		
buildpack	The buildpack to use for deploying the application.	
domain	The domain to use when mapping routes for the deployed application. domain and host are mutually exclusive with routes .	
routes	The routes to which to bind the deployed application.	
health-check	The type of health check to perform on the deployed application.	PORT
health-check-http-endpoint	The path used by the HTTP health check.	/health
health-check-timeout	The timeout value used by the health check, in seconds.	120
api-timeout	The timeout value used for blocking API calls, in seconds.	360
api-polling-timeout	The timeout value used for polling asynchronous API endpoints (for example, CF create/update/delete service instance), in seconds.	300

Property	Description	Default
status-timeout		
staging-timeout		
startup-timeout		
delete-routes	Whether to delete routes when un-deploying an application.	true
java-opts		

Environment Configuration

You can provide environment variables to be set on a deployed application. Environment variables are set by using properties under `environment` for the deployed application, as follows:

```
spring:
  cloud:
    appbroker:
      services:
        - service-name: example
          plan-name: standard
          apps:
            - name: example-service-app1
              path: classpath:app1.jar
              <strong>environment:</strong>
              <strong>logging.level.spring.security: DEBUG</strong>
              <strong>spring.profiles.active: cloud</strong>
```

Service Configuration

You can configure services that should be bound to a deployed application. Services are configured by using properties under `services` for the deployed application, as follows:

```

spring:
  cloud:
    appbroker:
      services:
        - service-name: example
          plan-name: standard
          apps:
            - name: example-service-app1
              path: classpath:app1.jar
              <strong>services:</strong>
              <strong>- service-instance-name: example-db</strong>
            <strong>services:</strong>
            <strong>- service-instance-name: example-db</strong>
              <strong>name: mysql</strong>
              <strong>plan: small</strong>
              <strong>parameters:</strong>
              <strong>param-key: param-value</strong>

```

4.1.2. Dynamic Customization

To customize the backing application deployment by using information that is only available when performing a service broker operation or that must be generated per service instance, you can use the service broker application configuration to provide the names of customization implementations.

Backing Application Target

You can configure the target location for backing applications (in Cloud Foundry, an org and space) using a **target** specification, as in the following example:

```

spring:
  cloud:
    appbroker:
      services:
        - service-name: example
          plan-name: standard
          <strong>target:</strong>
          * name: SpacePerServiceInstance*
      apps:
        apps:
          - name: example-service-app1
            path: classpath:app1.jar

```

By default (if you do not provide a **target** specification), all backing applications are deployed to the default target specified under `spring.cloud.appbroker.deployer`. For Cloud Foundry, this is the org

named by `spring.cloud.appbroker.deployer.cloudfoundry.default-org` and the space named by `spring.cloud.appbroker.deployer.cloudfoundry.default-space`.

The `SpacePerServiceInstance` Target

If you use the `SpacePerServiceInstance` target, App Broker deploys backing applications to a unique target location that is named by using the service instance GUID provided by the platform at service instance create time. For Cloud Foundry, this target location is the org named by `spring.cloud.appbroker.deployer.cloudfoundry.default-org`, and a new space is created by using the service instance GUID as the space name.

The `ServiceInstanceGuidSuffix` Target

If you use the `ServiceInstanceGuidSuffix` target, App Broker deploys backing applications by using a unique name and hostname that incorporates the service instance GUID provided by the platform at service instance create time. For Cloud Foundry, the target location is the org named by `spring.cloud.appbroker.deployer.cloudfoundry.default-org`, the space named by `spring.cloud.appbroker.deployer.cloudfoundry.default-space`, and an application name as `[APP-NAME]-[SI-GUID]`, where `[APP-NAME]` is the `name` listed for the application under `spring.cloud.appbroker.services.apps` and `[SI-GUID]` is the service instance GUID. The application also uses a hostname that incorporates the service instance GUID as a suffix, as `[APP-NAME]-[SI-GUID]`.

Creating a Custom Target

If you want to create a custom Target, App Broker provides a flexible way to add new targets by creating a new `Bean` that extends from `TargetFactory` and implementing the `create` method, as follows:

```

public class CustomSpaceTarget extends TargetFactory<CustomSpaceTarget.Config> {

    public CustomSpaceTarget() {
        super(Config.class);
    }

    @Override
    public Target create(Config config) {
        return this::apply;
    }

    private ArtifactDetails apply(Map<String, String> properties, String name,
String serviceInstanceId) {
        String space = "my-custom-space";
        properties.put(DeploymentProperties.TARGET_PROPERTY_KEY, space);

        return ArtifactDetails.builder()
            .name(name)
            .properties(properties)
            .build();
    }

    public static class Config {
    }
}

```

Once configured, we can specify in our service the new custom Target, as follows:

```

spring:
  cloud:
    appbroker:
      services:
        - service-name: example
          plan-name: standard
          target:
            name: CustomSpaceTarget

```

Service Instance Parameters

When a user provides parameters while creating or updating a service instance, App Broker can transform these parameters into the details of the backing app deployment by using parameters transformers. You can configure parameters transformers by using properties under `parameters-transformers`, as follows:

```

spring:
  cloud:
    appbroker:
      services:
        - service-name: example
          plan-name: standard
          apps:
            - name: example-service-app1
              path: classpath:app1.jar
              <strong>parameters-transformers:</strong>
                <strong>- name: EnvironmentMapping</strong>
                  <strong>args:</strong>
                    <strong>- include: parameter1,parameter2</strong>
                <strong>- name: PropertyMapping</strong>
                  <strong>args:</strong>
                    <strong>- include: count,memory</strong>

```

The named `parameters-transformers` refer to Java objects that have been contributed to the Spring application context. A parameters transformer can accept one or more arguments that configure its behavior and can modify any aspect of the backing application deployment (properties, environment variables, services, and so on).

The `EnvironmentMapping` Parameters Transformer

The `EnvironmentMapping` parameters transformer populates environment variables on the backing application from parameters provided when a service instance is created or updated. It supports a single argument, `include`, which specifies the names of parameters that are mapped to environment variables.

The `PropertyMapping` Parameters Transformer

The `PropertyMapping` parameters transformer sets deployment properties of the backing application from parameters provided when a service instance is created or updated. It supports a single argument, `include`, which specifies the names of deployment properties that should be recognized.

Credentials Generation

App Broker can generate and assign unique credentials for each backing app deployment. You can configure credential providers by using properties under `credential-providers`, as follows:

```

spring:
  cloud:
    appbroker:
      services:
        - service-name: example
          plan-name: standard
          apps:
            - name: example-service-app1
              path: classpath:app1.jar
              <strong>credential-providers:</strong>
                <strong>- name: SpringSecurityBasicAuth</strong>
                <strong>- name: SpringSecurityOAuth2</strong>

```

In this example, the named `credential-providers` refer to Java objects that have been contributed to the Spring application context. A credential provider can accept one or more arguments that configure its behavior. A credential provider typically generates credentials and sets environment variables on the backing application.

The `SpringSecurityBasicAuth` Credential Provider

The `SpringSecurityBasicAuth` credential provider generates a username and password and sets Spring Boot security properties to the generated values. Username and password generation can be configured with arguments, as follows:

```

spring:
  cloud:
    appbroker:
      services:
        - service-name: example
          plan-name: standard
          apps:
            - name: example-service-app1
              path: classpath:app1.jar
              <strong>credential-providers:</strong>
                <strong>- name: SpringSecurityBasicAuth</strong>
                  <strong>args:</strong>
                    <strong>length: 14</strong>
                    <strong>include-uppercase-alpha: true</strong>
                    <strong>include-lowercase-alpha: true</strong>
                    <strong>include-numeric: true</strong>
                    <strong>include-special: true</strong>

```

The `SpringSecurityOAuth2` Credential Provider

The `SpringSecurityOAuth2` credential provider creates an OAuth2 client in a token server (for

example, UAA for Cloud Foundry) by using details provided as arguments and a generated client secret. It also sets Spring Boot security properties to the generated values. Client secret generation can also be configured with arguments, as follows:

```
spring:
  cloud:
    appbroker:
      services:
        - service-name: example
          plan-name: standard
          apps:
            - name: example-service-app1
              path: classpath:app1.jar
              <strong>credential-providers:</strong>
                <strong>- name: SpringSecurityOAuth2</strong>
                  <strong>args:</strong>
                    <strong>registration: my-client-1</strong>
                    <strong>client-id: example-client</strong>
                    <strong>client-name: example-client</strong>
                    <strong>scopes: ["uaa.resource"]</strong>
                    <strong>authorities: ["uaa.resource"]</strong>
                    <strong>grant-types: ["client_credentials"]</strong>
                    <strong>identity-zone-subdomain:</strong>
                    <strong>identity-zone-id:</strong>
                    <strong>length: 14</strong>
                    <strong>include-uppercase-alpha: true</strong>
                    <strong>include-lowercase-alpha: true</strong>
                    <strong>include-numeric: true</strong>
                    <strong>include-special: true</strong>
```

4.2. Creating a Service Instance

Spring Cloud App Broker provides the `AppDeploymentCreateServiceInstanceWorkflow` workflow, which handles deploying the configured backing applications and services, as illustrated in the previous sections. The service broker application can implement the `CreateServiceInstanceWorkflow` interface to further modify the deployment. Multiple workflows can be annotated with `@Order` so as to process the workflows in a specific order. Alternatively, the service broker application can implement the `ServiceInstanceService` interface provided by Spring Cloud Open Service Broker. See [Service Instances](#) in the [Spring Cloud Open Service Broker documentation](#).

4.3. Updating a Service Instance

Spring Cloud App Broker provides the `AppDeploymentUpdateServiceInstanceWorkflow` workflow, which handles updating the configured backing applications and services, as illustrated in the previous sections. If the list of backing services is updated, the default behavior is to create and bind the new backing service instances and to unbind and delete the existing backing service

instances that are no longer listed in the configuration.

The service broker application can implement the `UpdateServiceInstanceWorkflow` interface to further modify the deployment. Multiple workflows can be annotated with `@Order` so as to process the workflows in a specific order. Alternatively, the service broker application can implement the `ServiceInstanceService` interface provided by Spring Cloud Open Service Broker. See [Service Instances](#) in the [Spring Cloud Open Service Broker documentation](#).



Modifying certain properties, such as disk and memory, when updating an application, may result in downtime.

4.4. Deleting a Service Instance

Spring Cloud App Broker provides the `AppDeploymentDeleteServiceInstanceWorkflow` workflow, which handles deleting the configured backing applications and services, as illustrated in the previous sections. The service broker application can implement the `DeleteServiceInstanceWorkflow` interface to further modify the deployment. Multiple workflows can be annotated with `@Order` so as to process the workflows in a specific order. Alternatively, the service broker application can implement the `ServiceInstanceService` interface provided by Spring Cloud Open Service Broker. See [Service Instances](#) in the [Spring Cloud Open Service Broker documentation](#).

4.5. Persisting Service Instance State

Spring Cloud App Broker provides the `ServiceInstanceStateRepository` interface for persisting service instance state. The default implementation is `InMemoryServiceInstanceStateRepository`, which uses an in memory `Map` to save state and offers an easy getting-started experience. To use a proper database for persisting state, you can implement `ServiceInstanceStateRepository` in your application.



The `InMemoryServiceInstanceStateRepository` is provided for demonstration and testing purposes only. It is not suitable for production applications!

4.5.1. Example Implementation

The following example shows a service instance state repository implementation:

```

package com.example.appbroker;

import reactor.core.publisher.Mono;

import org.springframework.cloud.appbroker.state.ServiceInstanceState;
import org.springframework.cloud.appbroker.state.ServiceInstanceStateRepository;
import org.springframework.cloud.servicebroker.model.instance.OperationState;

class ExampleServiceInstanceStateRepository implements
ServiceInstanceStateRepository {

    private final ServiceInstanceStateCrudRepository
serviceInstanceStateCrudRepository;

    ExampleServiceInstanceStateRepository(ServiceInstanceStateCrudRepository
serviceInstanceStateCrudRepository) {
        this.serviceInstanceStateCrudRepository =
serviceInstanceStateCrudRepository;
    }

    @Override
    public Mono<ServiceInstanceState> saveState(String serviceInstanceId,
OperationState state, String description) {
        return serviceInstanceStateCrudRepository.findByServiceInstanceId
(serviceInstanceId)
            .switchIfEmpty(Mono.just(new ServiceInstance()))
            .flatMap(serviceInstance -> {
                serviceInstance.setServiceInstanceId(serviceInstanceId);
                serviceInstance.setOperationState(state);
                serviceInstance.setDescription(description);
                return Mono.just(serviceInstance);
            })
            .flatMap(serviceInstanceStateCrudRepository::save)
            .map(ExampleServiceInstanceStateRepository::
toServiceInstanceState);
    }

    @Override
    public Mono<ServiceInstanceState> getState(String serviceInstanceId) {
        return serviceInstanceStateCrudRepository.findByServiceInstanceId
(serviceInstanceId)
            .switchIfEmpty(Mono.error(new IllegalArgumentException("Unknown
service instance ID " + serviceInstanceId)))
            .map(ExampleServiceInstanceStateRepository::
toServiceInstanceState);
    }

    @Override
    public Mono<ServiceInstanceState> removeState(String serviceInstanceId) {

```

```

        return getState(serviceInstanceId)
            .doOnNext(serviceInstanceState ->
serviceInstanceStateCrudRepository.deleteByServiceInstanceId(serviceInstanceId));
    }

    private static ServiceInstanceState toServiceInstanceState(ServiceInstance
serviceInstance) {
        return new ServiceInstanceState(serviceInstance.getOperationState(),
serviceInstance.getDescription(), null);
    }
}

```

One option for persisting service instance state is to use a Spring Data `CrudRepository`. The following example shows a `ReactiveCrudRepository` implementation:

```

package com.example.appbroker;

import reactor.core.publisher.Mono;

import org.springframework.data.r2dbc.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.data.repository.reactive.ReactiveCrudRepository;

interface ServiceInstanceStateCrudRepository extends ReactiveCrudRepository
<ServiceInstance, Long> {

    @Query("select * from service_instance where service_instance_id =
:service_instance_id")
    Mono<ServiceInstance> findByServiceInstanceId(@Param("service_instance_id")
String serviceInstanceId);

    @Query("delete from service_instance where service_instance_id =
:service_instance_id")
    Mono<Void> deleteByServiceInstanceId(@Param("service_instance_id") String
serviceInstanceId);
}

```

A model object is necessary for persisting data with a `CrudRepository`. The following example shows a `ServiceInstance` model:

```

package com.example.appbroker;

import org.springframework.cloud.servicebroker.model.instance.OperationState;
import org.springframework.data.annotation.Id;

class ServiceInstance {

    @Id
    private Long id;

    private String serviceInstanceId;

    private String description;

    private OperationState operationState;

    public ServiceInstance() {

    }

    public ServiceInstance(String serviceInstanceId, String description,
OperationState operationState) {
        this.serviceInstanceId = serviceInstanceId;
        this.description = description;
        this.operationState = operationState;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getServiceInstanceId() {
        return serviceInstanceId;
    }

    public void setServiceInstanceId(String serviceInstanceId) {
        this.serviceInstanceId = serviceInstanceId;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}

```

```
}

public OperationState getOperationState() {
    return operationState;
}

public void setOperationState(OperationState operationState) {
    this.operationState = operationState;
}

}
```

Chapter 5. Service Bindings

By default, Spring Cloud App Broker does not include functionality for managing bindings to its service instances. App Broker provides interfaces that service broker authors can implement to control service bindings.

5.1. Creating a Service Binding

The service broker application can implement the `CreateServiceInstanceAppBindingWorkflow` interface. Alternatively, the service broker application can implement the `ServiceInstanceBindingService` interface provided by Spring Cloud Open Service Broker. See [Service Bindings](#) in the [Spring Cloud Open Service Broker documentation](#).

5.2. Deleting a Service Binding

The service broker application can implement the `DeleteServiceInstanceBindingWorkflow` interface. Alternatively, the service broker application can implement the `ServiceInstanceBindingService` interface provided by Spring Cloud Open Service Broker. See [Service Bindings](#) in the [Spring Cloud Open Service Broker documentation](#).

5.3. Persisting Service Instance Binding State

Spring Cloud App Broker provides the `ServiceInstanceBindingStateRepository` interface for persisting service instance binding state. The default implementation is `InMemoryServiceInstanceBindingStateRepository`, which uses an in memory `Map` to save state and offers an easy getting started experience. In order to use a proper database for persisting state, implement `ServiceInstanceBindingStateRepository` in your application.



The `InMemoryServiceInstanceBindingStateRepository` is provided for demonstration and testing purposes only. It is not suitable for production applications!

5.3.1. Example Implementation

The following example shows a service instance binding state repository implementation:

```

package com.example.appbroker;

import reactor.core.publisher.Mono;

import org.springframework.cloud.appbroker.state.ServiceInstanceBindingStateRepository;
import org.springframework.cloud.appbroker.state.ServiceInstanceState;
import org.springframework.cloud.servicebroker.model.instance.OperationState;

class ExampleServiceInstanceBindingStateRepository implements
ServiceInstanceBindingStateRepository {

    private final ServiceInstanceBindingStateCrudRepository
serviceInstanceBindingStateCrudRepository;

    ExampleServiceInstanceBindingStateRepository(
        ServiceInstanceBindingStateCrudRepository
serviceInstanceBindingStateCrudRepository) {
        this.serviceInstanceBindingStateCrudRepository =
serviceInstanceBindingStateCrudRepository;
    }

    @Override
    public Mono<ServiceInstanceState> saveState(String serviceInstanceId, String
bindingId, OperationState state,
        String description) {
        return serviceInstanceBindingStateCrudRepository
            .findByServiceInstanceIdAndBindingId(serviceInstanceId, bindingId)
            .switchIfEmpty(Mono.just(new ServiceInstanceBinding()))
            .flatMap(binding -> {
                binding.setServiceInstanceId(serviceInstanceId);
                binding.setBindingId(bindingId);
                binding.setOperationState(state);
                binding.setDescription(description);
                return Mono.just(binding);
            })
            .flatMap(serviceInstanceBindingStateCrudRepository::save)
            .map(ExampleServiceInstanceBindingStateRepository:
:toServiceInstanceState);
    }

    @Override
    public Mono<ServiceInstanceState> getState(String serviceInstanceId, String
bindingId) {
        return serviceInstanceBindingStateCrudRepository
            .findByServiceInstanceIdAndBindingId(serviceInstanceId, bindingId)
            .switchIfEmpty(Mono.error(new IllegalArgumentException(
                "Unknown binding: serviceInstanceId=" + serviceInstanceId
+ ", bindingId=" + bindingId)))

```

```

        .map(ExampleServiceInstanceBindingStateRepository:
:toServiceInstanceState);
    }

    @Override
    public Mono<ServiceInstanceState> removeState(String serviceInstanceId, String
bindingId) {
        return getState(serviceInstanceId, bindingId)
            .doOnNext(serviceInstanceState ->
serviceInstanceBindingStateCrudRepository
                .deleteByServiceInstanceIdAndBindingId(serviceInstanceId,
bindingId));
    }

    private static ServiceInstanceState toServiceInstanceState
(ServiceInstanceBinding binding) {
        return new ServiceInstanceState(binding.getOperationState(), binding
.getDescription(), null);
    }
}

```

One option for persisting service instance binding state is to use a Spring Data **CrudRepository**. The following example shows a **ReactiveCrudRepository** implementation:


```

package com.example.appbroker;

import reactor.core.publisher.Mono;

import org.springframework.data.r2dbc.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.data.repository.reactive.ReactiveCrudRepository;

interface ServiceInstanceBindingStateCrudRepository extends
ReactiveCrudRepository<ServiceInstanceBinding, Long> {

    @Query("select * from service_instance_binding " +
           "where service_instance_id = :service_instance_id " +
           "and binding_id = :binding_id")
    Mono<ServiceInstanceBinding> findByServiceInstanceIdAndBindingId(
        @Param("service_instance_id") String serviceInstanceId,
        @Param("binding_id") String bindingId);

    @Query("delete from service_instance_binding " +
           "where service_instance_id = :service_instance_id " +
           "and binding_id = :binding_id")
    Mono<Void> deleteByServiceInstanceIdAndBindingId(
        @Param("service_instance_id") String serviceInstanceId,
        @Param("binding_id") String bindingId);

}

```

A model object is necessary for persisting data with a `CrudRepository`. The following example shows a `ServiceInstanceBinding` model:

```

package com.example.appbroker;

import org.springframework.cloud.servicebroker.model.instance.OperationState;
import org.springframework.data.annotation.Id;

class ServiceInstanceBinding {

    @Id
    private Long id;

    private String bindingId;

    private String serviceInstanceId;

    private String description;

    private OperationState operationState;

    public ServiceInstanceBinding() {

    }

    public ServiceInstanceBinding(String bindingId, String serviceInstanceId,
String description,
        OperationState operationState) {
        this.bindingId = bindingId;
        this.serviceInstanceId = serviceInstanceId;
        this.description = description;
        this.operationState = operationState;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getBindingId() {
        return bindingId;
    }

    public void setBindingId(String bindingId) {
        this.bindingId = bindingId;
    }

    public String getServiceInstanceId() {
        return serviceInstanceId;
    }

```

```
}

    public void setServiceInstanceId(String serviceInstanceId) {
        this.serviceInstanceId = serviceInstanceId;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public OperationState getOperationState() {
        return operationState;
    }

    public void setOperationState(OperationState operationState) {
        this.operationState = operationState;
    }
}
```

Chapter 6. Deployment Platforms

You can configure details of deployment platforms in App Broker configuration properties. These properties are under `spring.cloud.appbroker.deployer`. Currently, Spring Cloud App Broker supports only Cloud Foundry as a deployment platform.

To configure a Cloud Foundry deployment platform, use properties under `spring.cloud.appbroker.deployer.cloudfoundry`, as follows:

```
spring:
  cloud:
    appbroker:
      deployer:
        <strong>cloudfoundry:</strong>
        <strong>api-host: api.sys.example.local</strong>
        <strong>api-port: 443</strong>
        <strong>username: admin</strong>
        <strong>password: adminpass</strong>
        <strong>client-id: EXAMPLE_ID</strong>
        <strong>client-secret: EXAMPLE_SECRET</strong>
        <strong>default-org: test</strong>
        <strong>default-space: development</strong>
```



The two properties, `username` and `password`, and the two properties, `client-id` and `client-secret`, are mutually exclusive. The `client-id` and `client-secret` properties are for use with OAuth 2.0.