Scope: this document collects requirements for a backup/restore/clone service, as a spin off of initial conversation in https://github.com/krujos/data-lifecycle-service-broker/issues/3
See opensource implementation available into
https://github.com/Orange-OpenSource/service-db-dumper

Motivation

Beyond the HA/durability properties of some of the db services, CloudFoundry users want to be able to backup/restore their db content to cope with application or operator-generated mistakes applied on the data. Also, there is a need to clone a data set in order to reproduce problems observed in production in a dev environment, or to share test data sets among team members. Last, there is the use-case of transferring a data set among different providers of a compatible service (e.g. various mysql providers avaiilable in a public Paas).

Currently, it is a common case for data services to be exposed on private addressing schemes in CF-managed services[1], preventing data services endpoints to be generally accessible from outside CF, typically from CF users desktops. Additionally the CF v1 “tunnelling to services” feature has not yet been restored with CF v2, nor diego. To fulfill the use-cases above, Cf users have therefore to bind their app to a web-based db client (such as phpmyadmin) and try to perform db dumps from there. This is however a tedious and unreliable process.

[1] See http://docs.cloudfoundry.org/concepts/security.html#system-boundaries

Goal

As a first step, this document is describing a basic full data dump into an S3 bucket, which may be applicable to stateful data services providing the ability to dump their content as one or multiple files, and to re-import them. This may not be applicable to large data sets for which performance impacts of such dump would rather require incremental dumps, or leverage snapshot capabilities of the underlying storage.

Backup/Restore use-cases

One use-case for the backup/restore plan is:

- before deploying a new version of an application, app-ops want to perform a backup of the db (i.e. the db service instance)
- the new version of the application is deployed which alters the db schema and db content.
- if the deployment shows regressions, they can restore the backup to the service instance, and rollback to initial app version.

Let’s assume the following deployment scenario:

- a maintenance window is planned with users told the app will be unavailable during maintenance
- public route is unbound from app “N” and bound to an app “M” displaying maintenance page.
- app “N” is potentially stopped if this is required for data consistency of the dump.
- a dump of the bound db service instance is requested
- the version N+1 is pushed as a second app “N+1”, and bound to the same db service instance.
- the app “N+1” is tested with a private route.
- If tests are OK, public route is bound to app N+1. N+1 version is live
- If tests are KO, and rollback is decided:
    - a reimport of the dump is requested into the db service instance
    - app N is potentially started (if stop was required for data consistency)
    - public route is bound to app N. N version is live.

Clone / Create from dump use-case

Another use-case is the need to clone a data set, to reproduce problems observed in production in a dev environment, or to reuse reference test data sets among successive integration test steps.

Requirements

Requirements for a db dump capability

A requester requires a full dump of a source db represented by a source service instance.

Before performing the dump

A requester is checked for control/ownership over the source db, (e.g. has space manager role within the space where the source service instance is created)

A requester may potentially require to anonymize the dump content, if usage of the dump is known in advance and secrets need to be prevented from being included in the dump.

During the dump:

The potential impact on the source db should be documented as to leave responsibility to the requester to choose the appropriate maintenance window to request this dump.

Some dumps logs of the dump process are made available to the requester to understand failures, slow responses of the dump. However, sensitive credentials are not present in these logs.

The requester has the ability to interrupt and cancel a dump process which is taking too long and exceeds the planned maintenance window.

The dump request completes with a success or failure status. In case of a failure, transient resources are cleaned up (in order to avoid leaks)

After the dump completion

The requester retains ownership and control over this dump (for easing housekeeping and managing confidentiality): the dump is exposed as a service instance, and therefore can be listed, inspected, deleted, and potentially charged/billed for the life duration of the dump. Also the usual CF features (e.g. sharing a service instance among spaces, service instance audit events) would normally apply to the dump.

The requester can access the dump in read-only mode using machine-readable credentials, or a dashboard url for humans (e.g. to check the anonymisation process was effective).

Requirements for an import dump capability into a new service instance

A requester requires to import a source dump into a target service instance.

Before the import:

The requester is checked for control/ownership over the source dump

During the import:

Some dumps logs of the new service instance creation and import process are made available to the requester to understand failures, slow responses of the import. However, sensitive credentials are not present in these logs.

The requester has the ability to interrupt and cancel an import process which is taking too long.

The import request completes with a success or failure status. In case of a failure, transient resources are cleaned up (in order to avoid leaks).

Proposed design

Export capability through service-broker [de]provision operation(s)

A “data-service-dump” service is exposed in the marketplace, with a plan for each support db dialect/protocol

```
$ cf m
```

service	plans	description
p-mysql	100mb, 1gb	MySQL databases on demand
data-service-dump	mysql, pg, mongo	Db dumps on demand (currently supporting mysql, pg, mongodb)

```
$ cf s
```

name	service	plan	bound apps	last operation
sourcedb	p-mysql	100 mb	spring-travel	create succeeded

A requester requires a full dump of a source db represented by a source service instance name “sourcedb”, by invoking the dump service, specifying the source service instance name into arbitrary params,and potentially an anonymisation spec (similar to data-anonymization-dsl)

```
$ cf cs data-service-dump mysql  dump-before-upgrade -c '{"source":”sourcedb”}'
Creating service instance dump-before-upgrade in org orange / space elpaaso as gberche
… in progress
connecting to db instance… OK
starting dump… OK
1000 statement dumps...OK
1000 statement dumps...OK
Dump complete.
OK
```

TODO: precise the mechanism by which the request is granting the service broker permissions to act on his behalf, rather than use an CC admin account.[2]

The service creation is async, so the dump process can be stopped with a delete request.[3]

```
$ cf s
```

name	service	plan	bound apps	last operation
dump-before-upgrade	data-service-dump	mysql		create succeeded
sourcedb	p-mysql	100 mb	spring-travel	create succeeded

[2] See discussion in autosleep service proposal for a potential UX and future enhancements to this flow
[3] Currently, the service broker does not allow for a delete/deprovision request while a provision request has not completed. See http://docs.cloudfoundry.org/services/asynchronous-operations.html#blocking

The service instance exposes an HTTP endpoint to potentially request other operations from the dump. The HTTP endpoint is communicated through the dashboard URL.

```
$ cf s dump-before-upgrade

Service instance: dump-before-upgrade
Service: data-service-dump
Plan: mysql
Description: MySQL, Pgsql, Mongo db dumps on demand
Documentation url:
Dashboard: https://….

Last Operation
Status: create succeeded
Message:
```

When the dump service instance is bound to an application, the service broker bind endpoint returns read-only credentials to the S3 bucket, allowing an app to programmatically access the dump.

Limitation with this approach:

- it is not possible to interrupt interactively a dump which is taking too long to complete through the service broker API.

Import capability through service-broker update operation

The same “data-service-dump” service used to trigger a dump, also exposes an “update” operation supporting arbitrary params which is used to request the import of the dump into an existing target service instance.

```
$ cf m
[...]
```

service	plans	description
mongodb26	free	MongoDB 2.6 service for application development and testing
p-mysql	100mb, 1gb	MySQL databases on demand
cleardb	spark, boost*, amp*, shock*	Highly available MySQL for your Apps.
data-service-dump	mysql, pg, mongo	Db dumps on demand (currently supporting mysql, pg, mongodb)

A requester requires to import a source dump instance into an existing target service instance by identifying the existing target db to import into through arbitrary service params. The target db is identified by the name of the corresponding service instance in the same space[4].

[4] Further refinements will extend this to support target service instances in different spaces or org, provided that they are accessible to the requester

```
$ cf s
Getting services in org orange / space elpaaso as gberche...
OK
```

name	service	plan	bound apps	last operation
sourcedb	p-mysql	100 mb	spring-travel	create succeeded
dump-before-upgrade	data-service-dump	default		create succeeded
dump-after-failed-upgrade	data-service-dump	default		create succeeded
clone-before-upgrade	p-mysql	100 mb	spring-travel	create succeeded

```
$ cf update-service dump-before-upgrade -c { target-db=clone-before-upgrade, ...}
Updating service instance clone-before-upgrade in org orange / space elpaaso as gberche…
… Importing dump into clone-before-upgrade mysql db … SUCCESS
OK
```

A refinement to this approach is to provide a CLI plugin to interactively prompt compatible target db service instances (e.g. in all accessible spaces by the current user).

Under the hood, the data-service-dump service brokers fetches the access credentials of the specified target db, and imports the dump content into it, overriding previous data.

To restore an dump into a new db instance, the user has to provision a new fresh target db service instance.

Under the hood

From the paas-ops point of view, this means deploying a spring-boot app with the following env vars, and registering it as the “data-service-dump” service broker:

- cc account[5] with necessary credentials to be able to look up dumps service instances in spaces
    - cc login
    - cc password
- S3 bucket credentials to read the dumps (e.g a bound S3 service instance)

When the update-service is called to restore the dump into a target db, the data-service-dump service broker requires the creation of service keys to the target db specified in arbitrary params. These service keys are transiently created in the space associated with the target db service instance, and deleted upon completion of the dump import process.

Limitations in this approach:

-

[5] See discussion in autosleep service proposal for a potential UX to have the broker act on behalf of the requester user, and therefore avoid needing to have cc.admin scopes to access anything else than the shadow space(s).

Other considered design alternatives

Import capability through dataservice service-broker [de]provision operation(s)

A “data-service-importable” service is exposed in the marketplace for each stateful service, and acts as a facade (i.e exposing the same catalog, and returning same credentials and dashboard url as the original data service). This face service is supporting additional arbitrary params in the create and update requests, to respectively create a new data service from a dump, or update an existing service instance from a dump.

The following marketplace example shows ~~striked through~~ original data services to illustrate the fact that a CF operator might choose to only expose the enhanced facade services as to avoid user confusion with similar offers.

```
$ cf m
[...]
```

service	plans	description
mongodb26	free	MongoDB 2.6 service for application development and testing
~~p-mysql~~	~~100mb, 1gb~~	~~MySQL databases on demand~~
~~cleardb~~	~~spark, boost*, amp*, shock*~~	~~Highly available MySQL for your Apps.~~
p-mysql-importable	100mb, 1gb	MySQL databases on demand, optionally created/updated from dumps
cleardb--importable	spark, boost*,	Highly available MySQL for your Apps, optionally created/updated from dumps

	amp*, shock*	

A requester requires to import a source dump instance into a new target service instance by identifying the dump to import through arbitrary service params. This results into a new service instance, whose content is initialized with the source dump. This assumes the source dump and new instances are in the same space.

```
$ cf s
Getting services in org orange / space elpaaso as gberche...
OK
```

name	service	plan	bound apps	last operation
sourcedb	p-mysql-importable	100 mb	spring-travel	create succeeded
dump-before-upgrade	data-service-dump	default		create succeeded
dump-after-failed-upgrade	data-service-dump	default		create succeeded

```
$ cf cs p-mysql-importable 100mb clone-before-upgrade -c {
source-dump=dump-before-upgrade, ...}
Creating service instance clone-before-upgrade in org orange / space elpaaso as gberche...
OK
```

The service p-mysql-import broker receives the provisioning request with the org, space params. Under the hood, in a system space, with a specific CC account, it asks the creation of a “p-mysql” instance with plan “100mb”, and returns the status to CC.

The service creation and dump import is async, so it can be interrupted with a service instance delete request.

```
$ cf s
Getting services in org orange / space elpaaso as gberche...
```

```
OK
```

name	service	plan	bound apps	last operation
sourcedb	p-mysql-importable	100 mb	spring-travel	create succeeded
dump-before-upgrade	data-service-dump	default		create succeeded
dump-after-failed-upgrade	data-service-dump	default		create succeeded
clone-before-upgrade	p-mysql-importable	100 mb		create succeeded

Alternatively, to re-import a source dump into an existing target service instance, the update-service can be used. This translates into the service broker to receive an update service instance request

```
$ cf update-service source-db -c { source-dump=dump-before-upgrade, ...}
Updating service instance source-db in org orange / space elpaaso as gberche...
OK
```

Under the hood

From the paas-ops point of view, this means deploying a spring-boot app with the following env vars, and registering it as the “data-service-importable” service broker:

- shadow-space: shadow space into which original service-instance will be created
- cc account[6] with necessary credentials to be able to look up dumps service instances in spaces and instantiate service instances in the shadow space).
  - cc login
  - cc password
- target-service-type (e.g “p-mysql”)
- S3 bucket credentials to read the dumps (e.g a bound S3 service instance)
- A persistent store (shared with the

[6] See discussion in autosleep service proposal for a potential UX to have the broker act on behalf of the requester user, and therefore avoid needing to have cc.admin scopes to access anything else than the shadow space(s).

When the “clone-before-upgrade” service instance is bound to an app, the p-mysql-importable service broker requires the creation of service keys to the shadow p-mysql service instance (create-service-key), and returns it as credentials of the “clone-before-upgrade” service instance.

Symmetrically, on unbinding the p-mysql-importable service broker translates this into a delete-service-key to p-mysql service.

Finally, a delete request on the “clone-before-upgrade” results into an underlying delete request to the p-mysql shadow service instance.

Limitations in this approach:

- lack of consistency with original source service type: imported dbs from dumps don’t have the original type (e.g. p-mysql), which may confuse a bit inventories, usage reports and billing. This may somewhat be overcome by entirely removing the original data service (p-mysql) from the marketplace of private CF instances (TBC public CF instance don’t yet seem allow to restriction of marketplace per orgs/spaces)
- wrapping (p-mysql into p-mysql-importeable) implies a tight coupling between the original services and the import/export service). How would the import/export service catalog (p-mysql-importeable) would be updated to match the original service (p-mysql) ?
    - What kind of changes:
        - catalog changes
            - service plans
        - service binding changes ?
            - dashboard url
    - It is a manual task left to the CF operator ?
    - Can this be automated ?
        - How to detect changes in the catalog of the wrapped service ?
            - [Upcoming CF Notifications on service broker changes]
            - Regular polling
        - How to update the wrapped service ?
            -
- quota enforcement: lack of a unique quota for p-mysql service whether directly instanciated or instanciated from a dump. A user could instanciate 10 p-mysql instance and 10 p-mysql-importeable instance. This may somewhat be overcome by entirely removing the original data service (p-mysql) from the marketplace.

Import capability through a dashboard/REST api on the dump service broker

The dump service instance exposes a HTTP endpoint to request the import of the dump into an existing compatible service instance. This may be exposed as a web ui (protected by an oauth authorization) and asking

```
$ cf service dump-before-upgrade

Service instance: dump-before-upgrade
Service: data-service-dump
Plan: Default
Description: MySQL, Pgsql, Mongo db dumps on demand
Documentation url:
Dashboard: https://dump-service/dump-guid/

Last Operation
Status: create succeeded
Message:
```

The REST API exposes the following verbs

Import request

Request an import of a source dump into the specified existing target service instance, or into a new service instance. The import, which is an async task, which can then be queried to check its status, get execution logs, and potentially cancelled if too long.

POST /dump-importer? [update-target-service-instance-guid=] | (create-target-service-instance-name= & create-target-space-guid)
where:

- update-target-service-instance-guid: [Optional[7]] the guid of an existing compatible target service instance (e.g. p-mysql) to import the source dump into. A temporary new service key wil be requested to connect this instance and import the dump, before being deleted. [8]

[7] if omitted, the create-target-* params should be provided instead
[8] Other alternatives could be to pass in a service binding instance (which contain credentials), however this requires that the target service be bound. Otherwise, the access credentials themselves could be passed in (e.g. mysql uri). This is simpler and restricts granted oauth access to the dump service broker (e.g. to not mess up with another service instance than the specified one). However, this requires the service key to pre-exist and discloses it to a 3rd party, and provides less isolation and traceability.

- create-target-service-instance-name: [Optional[9]] the name of a service instance to use in creating a service into which the source dump will be imported
- create-target-service-name: [Optional] the name of a service to use in creating a service into which the source dump will be imported
- create-target-service-plan: [Optional] the name of the service plan to use in creating a service into which the source dump will be imported
- create-target-service-params: [Optional] a JSON formatted string holding the arbitrary params to use in creating a service into which the source dump will be imported
- create-target-space-guid: [optional] the space guid into which a service instance will be created, and into which the source dump will be imported. When omitted, the service instance is created in the same space as the source service instance dump.
- HTTP Authorization: Bearer OAuth Token: OAuth token of the request to request a service key for the specified service instance to import, user will be asked to grant the cloud_controller.write scope

Returns:
Status code:
- 200,
- 403: insufficient authorization
- 404: no such update-target-db, target-service-name, target-service-plan, target-space-guid
- 409: conflict, incompatible target (e.g. pg) with source dump (mysql)

Body: { import-process-task-id=123, error-details='blabla" }

Get status on a requested import task

GET import-task/import-process-task-id?log_offset=bytes
where:
- log_offset=offset from start of logs to return. Allows paging of logs to not returned previously displayed logs

Returns: 200, 404
Body: { status=in-progress|completed, next-log-offset=1024, log='log importation started. connecting to service instance, batch1 (10023 statements, 1024 bytes) sent… done'}

To ease consumption of the REST endpoint, a CLI plugin is provided, resulting in the following UX

```
$ cf s
Getting services in org orange / space elpaaso as gberche...
OK
```

[9] if omitted, the update-target-* params should be provided instead

name	service	plan	bound apps	last operation
dump-before-upgrade	data-service-dump	default		create succeeded
sourcedb	p-mysql	100 mb	spring-travel	create succeeded
target-db	p-mysql	500mb		create succeeded

```
#update existing target instance
$ cf import-dump -d dump-before-upgrade -u targetdb
Importing dump ‘dump-before-upgrade’ into service ‘targetdb as gberche
6/2/2015 22:26:00: log importation started. creating service key for service instance 123456 as gberche… done connecting to service instance, batch1 (623 statements, 1024,000 bytes) sent… done
6/2/2015 22:26:10: batch2 (239statements, 1003,123 bytes) sent… done
6/2/2015 22:26:00: batch import done. deleting service key… done. dump import complete
OK
```

```
#restore dump into a new service instance to be created
$ cf import-dump -d dump-before-upgrade -cs targetdb p-mysql 100mb -c '{"instance_size":m1.medium}'
```

Limitations/drawbacks of this approach:

- Requires the installation of a CLI plugin to request importation of a dump, or cloning of a data service instance.

Extension of the Service Broker API

An alternative approach is to move these actions within the lifecycle of the service instance on which you want to perform the actions.

Make the ability to perform actions such as “backup” or “restore” or “upgrade” a concept that CC and Service Brokers understand.

`Actions` could be a new concept introduced to the API. Upon triggering an action from the CLI, this is merely passed through to the service broker which executes the action in an async manner and returns the result.

There could either be a list of pre-agreed actions we want to support, or we could increase the complexity and allow service brokers to register a list of actions they support.

That action could be performing a dump of the database on that instance, using details pre-configured either by the Operator at a service wide level or per app instance through the dashboard by the application developer, such as the destination.

This could give way to a user flow more like

```
$ cf cs p-mysql standard my-db
<fill it with some data>

$cf list-actions p-mysql my-db
<this returns a list of actions that the service broker supports, such as
    - take-snapshot = snapshot the existing data set now
    - import-dataset = pass in the location as an arbitrary param of the tgz file to be restored

$ cf perform-action take-snapshot p-mysql my-db
<async this actions happens and returns true>
```

Advantages:

- more intuitive, it's a first class citizen of the API and the CLI, supporting
  - explicitly supported actions and their discovery
  - cancelling async execution of long running actions (whereas cancelling is not yet supported on current CUD operations of service broker API)
- Can start to get fancy and chain services. For example the app instance could be bound to an anonymizer service which is passed the data after it's exported and anonymized, before being passed back to the service broker to store on S3.

Disadvantages:

- Requires changes to CC / SB APIs & CLI
- If the actions are directly implemented by the data service (say p-mysql) and not a distinct service (say data-service-dump) then:
  - The dumps (tgz) are not materialized as service instances, making their management by end-user less explicit (request to delete, request to access it, bind it to an app, billable service events)
  - Only services supporting backups (say p-mysql) can be dumped, w.r.t. supporting dumps for cups or existing public services (e.g. an online public RDS/cloudant service in a public CF service)