

FINAL REPORT

CLOUD-NATIVE DATABASE MANAGEMENT AND MIGRATION THROUGH KUBERNETES OPERATORS

Hubert Stefański

20081102

Supervisor: Catherine Fitzpatrick

BSc (Hons) in Computer Forensics And Security

April 11, 2022

Contents

0.1	Introduction	4
0.2	Technologies	6
0.2.1	Glossary	6
0.2.2	Design & System Architecture	7
0.2.3	Core Technologies	8
0.2.3.1	Containerisation	8
0.2.3.2	Container Orchestration	9
0.2.3.3	Runtime	10
0.2.4	Application Level	10
0.2.4.1	Application/Service Orchestration	10
0.2.4.2	Implementation Language	11
0.3	Functionality	13
0.3.1	On Cluster Deployment	13
0.3.2	Azure Deployment	17
0.3.3	Database Migration	22
0.4	Issues Encountered	26
0.4.1	Azure	26
0.4.2	Kubernetes Cluster	29
0.5	Conclusion And Evaluation	31
0.5.1	Learnings	32
0.5.2	Further Development	34
0.5.2.1	Data Migration Between Cloud Environments	34
0.5.2.2	Extend Configuration	34
0.5.2.3	Support Other Cloud Providers	34
0.5.2.4	Support Other Types of Databases	35
0.5.2.5	Support Running OnCluster Deployments as Replicated Stateful Applications	35

0.6	References	36
0.7	Further Reading/Sources	36
0.8	Appendices	37
0.8.1	Implementation & Iterations	38
0.8.1.1	Iteration 1	38
0.8.1.2	CI/CD (Continuous Integration/Continuous Development)	39
0.8.1.3	Implementing The Controller	42
0.8.1.4	Iteration 2	47
0.8.1.5	Controller Refactoring	47
0.8.1.6	Fixed PV/PVC reconciler	48
0.8.1.7	Iteration 3	50
0.8.1.8	Reconciler and Controller Refactoring	50
0.8.1.9	OnCluster Cleanup Logic	52
0.8.1.10	Implement Secret and EnvFrom for MySQL Credentials	52
0.8.1.11	Make MySQL Accessible Through an Ingress	52
0.8.1.12	Iteration 4	53
0.8.1.13	Experimental code	53
0.8.1.14	Iteration 5	54
0.8.1.15	Implement Azure MySQL server creation and deletion	54
0.8.1.16	Add CR validation for Azure deployments	54
0.8.1.17	Extend logging	54
0.8.1.18	Fix types, previously empty fields would error out the operator on empty	55
0.8.1.19	Iteration 6	56
0.8.1.20	Iteration 7	57
0.8.1.21	Implement Azure -> OnCluster migration	57
0.8.1.22	Implement OnCluster -> Azure migration	57
0.8.1.23	Implement Readiness Checks for OnCluster Deployments	57
0.8.1.24	Implement Conditional Ingress Creation	58

Abstract

For a long time, the management of the data layer and databases, in the world of cloud applications and services has been a largely manual process, prone to human error and misconfiguration. Which could lead to expensive and time-consuming problems at runtime. Kubernetes operators provide developers with much-needed automation at the application level. This allows them to run flawlessly and maintain a state true to defined configurations for prolonged periods without the need for human intervention. Yet, data layer management remains a manual procedure for most organizations even with automation provided by Kubernetes. This project aims to resolve the issues present within the data layer of containerised applications running on Kubernetes and Docker-based cloud environments. A Kubernetes operator will be created to resolve the aforementioned issues. This project will provide DevOps engineers the ability to seamlessly and securely manage and migrate their databases between cloud environments. From public clouds, such as Azure, to private cloud and vice versa.

0.1 Introduction

The results of a recent survey carried out by the Cloud-Native Computing Foundation(CNFC) found:

“90%: Of those respondents who are running containerized applications today, nine out of 10 are doing so in production, continuing a steady upward trend: 84% were doing so last year and 67% in the 2017 survey.” (K. Casey, 2019)

For a long time database provisioning has been a tedious and flaky procedure, with many stages during which the process could fail. This combined with the rise of public cloud and cloud computing has increased the need for fast and efficient ways to set up resilient database deployments. As each computational minute costs organizations money, they will seek to minimise the time where the server or application isn't ready and actively receiving traffic. This combined with the occasional needs of database migration between environments undue pressures DevOps engineers to maintain uptime for many servers.

With cloud technologies, the likes of Docker (containerisation) and Kubernetes/Open-shift(container orchestration) becoming increasingly popular and having a year-on-year bigger market share puts into contrast the requirement for equally scalable solutions to deal with the data layer. This solution must integrable, yet loosely coupled to work flawlessly with any cloud application or service.

These days many companies have to develop their own, bespoke database management systems that suit their cloud needs. DBMMO would allow companies to reduce the time and effort required for the provisioning of databases.

The possibility of abstraction with Kubernetes custom resources would allow users to define their databases on the go. Meaning that there is no requirement of downtime to allow for manual setups, maintenance or bug fixing. Because of operators, users would have ensured that their databases would remain true to their defined state.

The solution to the aforementioned issues is DBMMO (Database Management and Migration Operator), which this project created and developed. DBMMO is a Kubernetes based operator for managing and migrating databases between cloud environments. Its mission is to provision, deploy, manage and migrate databases between a range of diverse public cloud providers. Able to do so with only minor reconfiguration by leveraging Kubernetes resources. DBMMO will control database deployments both on the same cluster and in public clouds. DBMMO utilises the Operator-SDK, which eases the burden of database management from DevOps engineers, by ensuring that the deployment will maintain its desired state throughout the application's lifetime. The operator having the ability to detect misconfiguration

or undesired states and fixing them immediately with no human intervention at any point throughout the runtime of the application. One of many advantages that DBMMO has over previous implementations of comparable systems is its ease of use. For instance with complex database setups, being able to define an entire database using a single YAML file and assigning the operator to do the rest.

The abstracted approach which DBMMO implements allows users to define a database setup that will be compatible with most cloud providers with solely minor changes in the original configuration files, thus giving users the possibility to easily switch and migrate their databases to another cloud provider without unnecessary intermediary steps. This furthermore eliminates the chance of database/data migrations going awry due to human error.

The flexible nature of Kubernetes and the Operator-SDK also allows developers to quickly and easily add support for more databases if needs be. DBMMO's code is highly reusable to further aid in this.

0.2 Technologies

0.2.1 Glossary

Apply or Create - Referring to the upsert/create action on that Custom resources undergo on the cluster.

Azure - Microsoft owned cloud environment.

Container - The specific image that Kubernetes manages.

Cloud Environments - The type of cloud/server in use, either private or public, in this context, either a locally managed Kubernetes cluster or Azure.

Custom Resource (CR) - A user-defined Kubernetes resource, this holds specific information to how a single or number of Kubernetes resources should look like.

Custom Resource Definition (CRD) - / Custom objects that can be applied to the Kubernetes cluster and be used like any other native Kubernetes objects.

(Kubernetes) Cluster - The server runtime environment in which the Kubernetes control plane lives and which it manages through Kubernetes resources such as CPU, memory and ephemeral storage.

(Kubernetes) Resource - An in-built resource or component of Kubernetes, these are generally predefined and are a building block for other resources that extend the Kubernetes API.

Multi-tenant - Clusters that are occupied by more than one user, i.e. shared clusters.

Namespace - The logical partition of the Kubernetes cluster to which resources are applied, generally custom resources will be scoped to a namespace.

Operator - In the context of this document, it refers to the implementation and logic of the application that utilises the Operator-SDK and extends the Kubernetes API to manage and run a certain set of resources relating to a product/functionality.

Route/Ingress - The endpoint address IP or URL from which a resource can be accessed, not all resources make use of or require these.

Secret - Kubernetes resource that contains sensitive information, such as cloud provider credentials, API keys etc.

The selected technologies that are utilised within this project are, at the time of writing this document, the most popular and advanced solution for cloud applications and microservices. These technologies have been proven to work time and time again across a vast amount of different infrastructures and millions of instances. This technology stack has become synonymous with the cloud, thanks to its robustness, scalability and performance.

0.2.2 Design & System Architecture

The system architecture had a strong requirement of having to be loosely fitting. A decoupled system means that support for any future providers or database types is easier to implement, this is made possible by keeping the operator logic as abstract as possible, allowing for generalised input

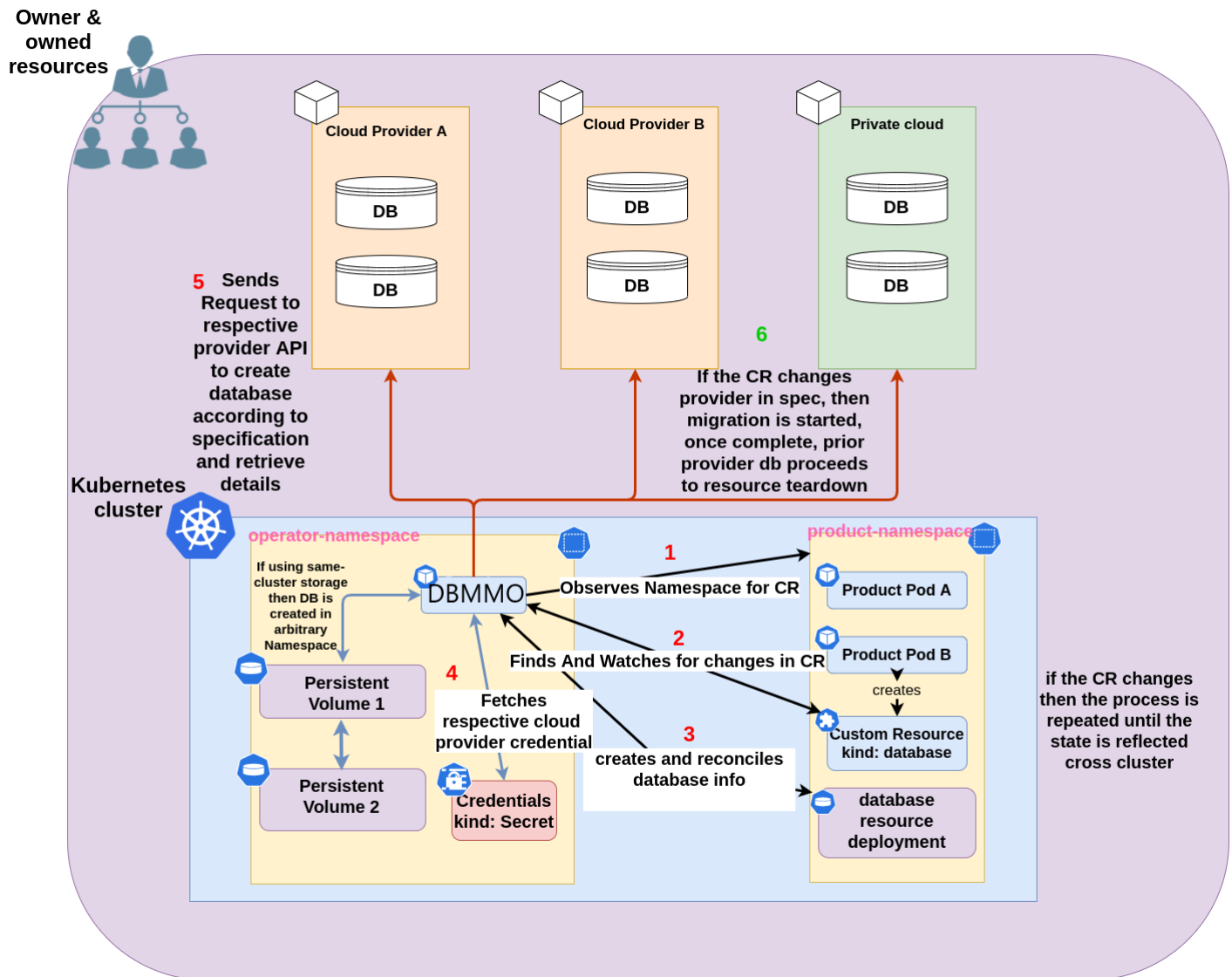


Figure 1: The Diagram illustrates the complete workflow of DBMMO and the high-level interaction between components, this shows how a new custom resource will kick off subsequent functions in DBMMO, from local creation to migration between hosts, green indicates the optional migration step

0.2.3 Core Technologies

0.2.3.1 Containerisation

Containerisation is the concept of "packaging" applications or services into images, doing so only with the necessary dependencies and libraries which the program will need at runtime, ensuring high efficiency. Containers are often compared to virtual machines, although both are quite similar, containers bring the advantage of the ability to run without the overhead for operating systems or unnecessary dependencies.

This project utilises *Docker* for containerisation. Using Docker brings the following benefits:

- Scalability - containers can be quickly scaled up and down, failing pods can be killed and replaced in a matter of seconds, allowing organizations to maintain their application/service uptime.
- Isolation - containers are isolated, meaning that they do not have access to each other's resources, instead, if such a need arises, they can communicate with one another through secure channels.
- Performance - containers have high performance as standard, due to no unnecessary packages or libraries, services have everything they need at hand.

Other alternatives include rkt, Containers for Windows, Mesos Containerizer, However, Docker remains the market leader.

0.2.3.2 Container Orchestration

Container orchestration is the process of managing containers at runtime. This project implements Kubernetes for this purpose

The name Kubernetes originates from Greek, meaning helmsman or pilot.

(Kubernetes Documentation, 2021)

Kubernetes provides plenty of benefits, such as:

- Service Discovery - Kubernetes allows containers to be externally accessible (from outside the cluster) through DNS or IP addresses.
- Load Balancing - Kubernetes can manage and distribute traffic amongst deployments/pods such that the whole system remains stable and up.
- Automated rollouts and rollbacks - Kubernetes can systematically replace resources such as pods or containers amongst many others with newer versions, i.e updating the actual state to the desired state.
- Self-Healing - Kubernetes health checks its containers and resources constantly, ensuring that failing ones are replaced and that the whole system is stable. This also means that non-healthy containers won't be queued in the load balancer.
- Authentication and configuration management - Kubernetes allows users to manage credentials, keys and application configuration within separate resources, ensuring that container images do not have to be rebuilt for these to be changed.

Kubernetes, like docker, also has several alternatives, which include Openshift (most closely resembling Kubernetes), SaltStack or Rancher. However, Kubernetes is still the most popular.

0.2.3.3 Runtime

Kubernetes is unconcerned with infrastructure, meaning that it can run on any platform while still ensuring the same level of stability and performance across the range of possible underlying components. For this reason, it is possible to run Kubernetes locally through solutions such as *Minikube*, which is a minimalistic Kubernetes cluster provided by the people behind Kubernetes itself.

Minikube allows quick deployment and also close to no cost of testing and development, which is a valuable advantage over publicly provided clusters. Running and testing on Minikube would be no different to running on hosted clusters. Thanks to Kubernetes it is ensured that if an application or service runs on one cluster it will also run on any other Kubernetes cluster.

For actual deployment, Kubernetes can run on most public cloud providers, such as Amazon Web Services, Google Cloud Platform, Microsoft Azure amongst others.

0.2.4 Application Level

0.2.4.1 Application/Service Orchestration

Just like Kubernetes resolves issues at the cluster level through orchestration, the application layer does so with the use of the Operator-SDK. With the use of this framework, developers can define custom resources(CRs) through custom resource definitions (CRDs), the Operator-framework based operator can then manage these custom resources.

Operators can create, update and delete sub-resources as the application or service requires them. They ensure that the application stays true to defined state throughout the application's lifetime, this means that if the actual state differs from the defined state, the sub-resource will be updated such that it is no longer mismatched, this process is called *reconciliation*. They do not have to implement the Operator-SDK (Operator-Framework) to be operators, however, using this SDK provides the following advantages::

- High-level APIs and abstraction for more intuitive operator logic writing - Gives developers a more convenient and simpler way of interacting with Kubernetes and its resources.
- Tools for scaffolding and code generation - Enabling developers to quickly bootstrap their projects, increases productivity by offloading monotonous tasks to automation.
- Extensions to cover common operator use cases - With convenient inbuilt methods

such as *controllerruntime.CreateOrUpdate(object)* to eliminate the need to write bespoke creation and update functions for every new custom resource.

0.2.4.2 Implementation Language

GoLang is the native language in which Docker and Kubernetes are written and which the Operator-SDK supports, therefore it is a logical choice for DBMMO to use. Golang boasts the following benefits

- **Compiled Language** - Compiled languages are well known for their speed as machine-level code can be read directly by the computer without the need for an interpreter.
- **Extensive Error Checking** - Go has implemented extensive error checking, which ensures that the code is free of unused variables, missing packages or mistyped or invalid operations. This makes development and debugging easier.
- **Cross-Compilation** - Go implements a multi-arch compiler, meaning that developers can compile images for a multitude of different machines without having to access those machines.
- **Garbage Collection** - Golang has built-in automatic memory management, providing developers with average sub 1ms latencies.
- **Scalability** - Go has been built in mind for scalability, with multiple features being implemented to enable efficient concurrency through *GoRoutines*.

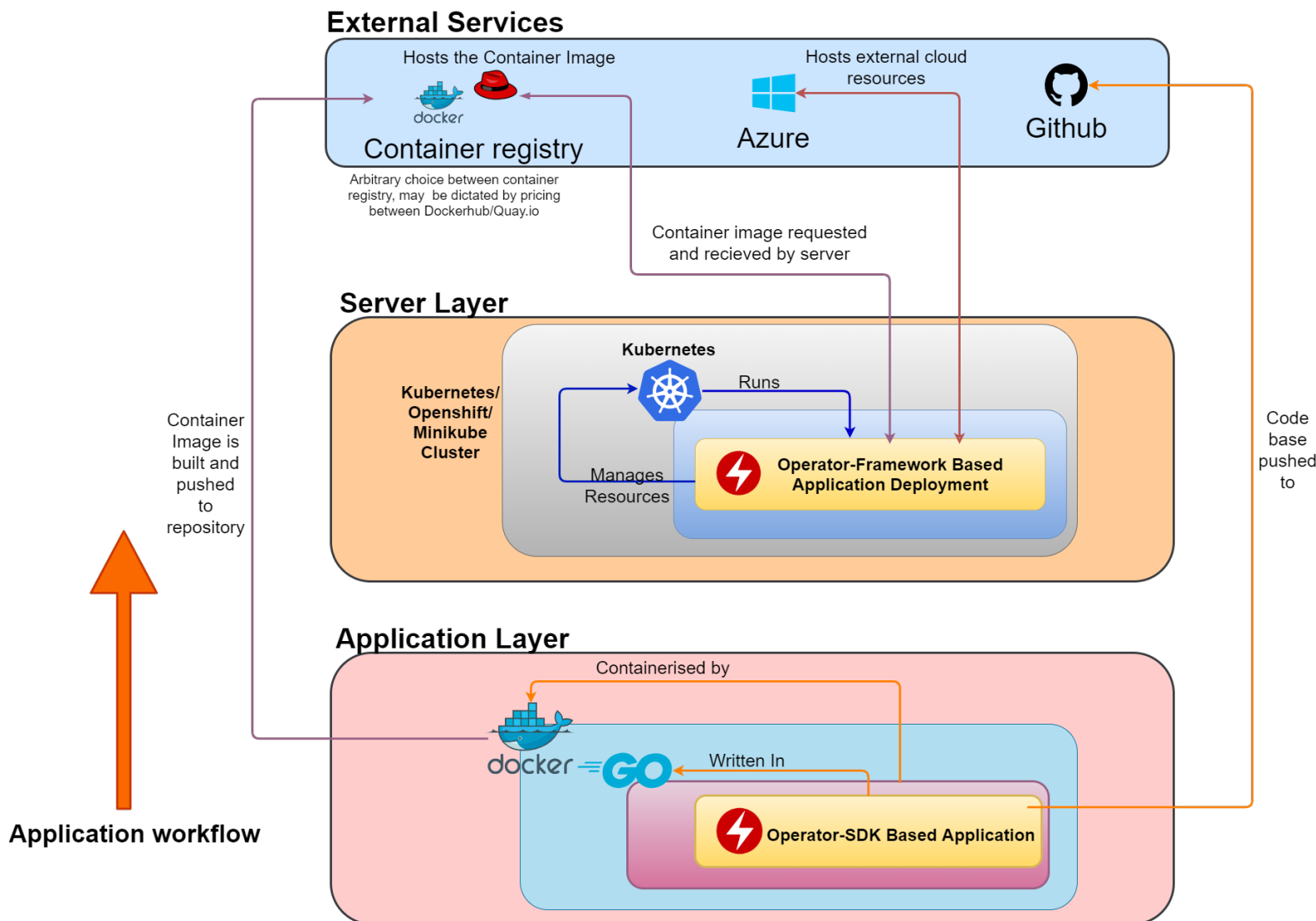


Figure 2: This Diagram illustrates the reconciler pattern and how certain technologies in the overall architecture work together.

0.3 Functionality

0.3.1 On Cluster Deployment

This functionality allows users to deploy a MySQL server running on the same Kubernetes cluster as DBMMO. This can be achieved by specifying the *Spec.Deployment.DeploymentType* and setting it to *onCluster*.

After parsing the *DBMMOMYSQL* custom resource, DBMMO will read the value provided in this field and create all necessary services, deployments etc. to deploy the MySQL server according to the specification declared in this resource. The deployed instance can then be accessed through an ingress or directly through the container running the instance. A separate persistent volume is also deployed to maintain the database and prevent information loss on server failures. This separation ensures that even during a failed deployment, the underlying database tables will be safe.

The code snippet for the OnCluster reconciler has been purposefully omitted as it is too large to fit reasonably within the report, instead, it can be found by following this [Github link](#)

A full resource cleanup has also been implemented, meaning that if the *DBMMOMYSQL* resource is deleted, DBMMO will kick off cleanup logic, tearing down all related resources, leaving the cluster as it were before. This is also necessary to ensure that after a migration has completed there are no remaining resources on the cluster.

```
1 //OnClusterCleanup cleans up the resources for a specific Mysql object
2 func (r *DBMMOMYSQLReconciler) OnClusterCleanup(ctx context.Context, m *cachev1alpha1.DBMMOMYSQL)
3                                     (ctrl.Result, error) {
4     pvc := model.GetMysqlPvc(m)
5     if err := r.Client.Delete(ctx, pvc); err != nil && !k8serr.IsNotFound(err) {
6         return ctrl.Result{RequeueAfter: constants.ReconcilerRequeueDelayOnFail}, err
7     }
8     svc := model.GetMysqlService(m)
9     if err := r.Client.Delete(ctx, svc); err != nil && !k8serr.IsNotFound(err) {
10        return ctrl.Result{RequeueAfter: constants.ReconcilerRequeueDelayOnFail}, err
11    }
12    dep := model.GetMysqlDeployment(m)
13    if err := r.Client.Delete(ctx, dep); err != nil && !k8serr.IsNotFound(err) {
14        return ctrl.Result{RequeueAfter: constants.ReconcilerRequeueDelayOnFail}, err
15    }
16    // Don't requeue if the cleanup was successful
17    return ctrl.Result{}, nil
18 }
19
```

```

20 func (r *DBMMOMySQLReconciler) cleanUpIngress(ctx context.Context, m *cachev1alpha1.DBMMOMySQL)
21                                     (ctrl.Result, error) {
22     ingr := model.GetMysqlIngress(m)
23     if err := r.Client.Delete(ctx, ingr); err != nil && k8serr.IsNotFound(err) {
24         return ctrl.Result{RequeueAfter: constants.ReconcilerRequeueDelayOnFail}, err
25     }
26     return ctrl.Result{}, nil
27 }

```

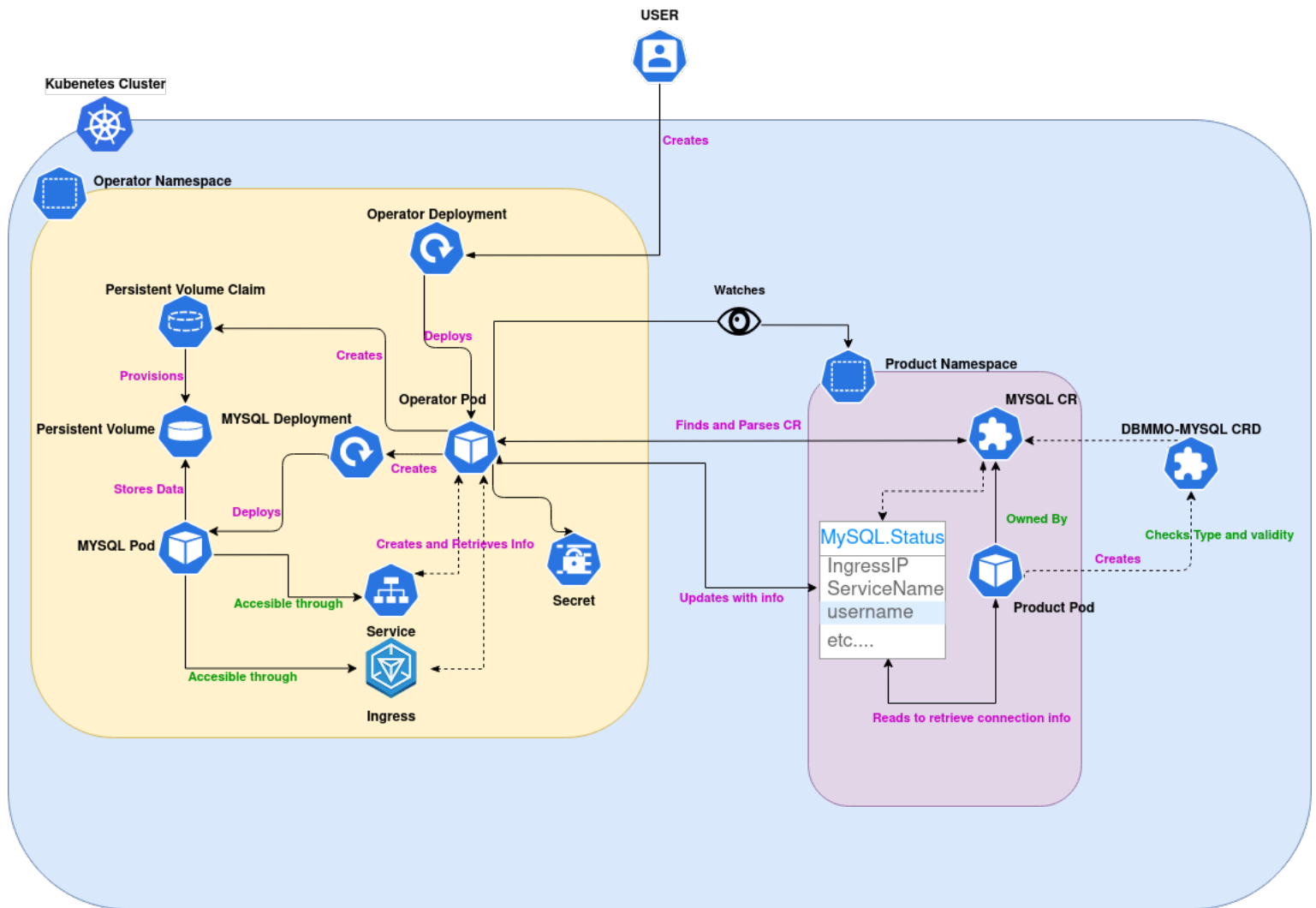


Figure 3: This Diagram illustrates the operator logic that deploys and manages the MySQL deployment within the cluster(deploymentType=OnCluster).

The following logic is executed:

1. A cluster administrator/privileged user creates a deployment specifying an external repository containing the DBMMO image.
2. Kubernetes then translates the deployment and creates a pod, in which DBMMO will run.
3. DBMMO then begins to watch a specified namespace for a known resource, in this case, MySQL.

4. An application or product pod creates a MySQL deployment, using the CRD to type and validate the YAML.
5. DBMMO finds the newly created MySQL CR and parses the configuration contained within.
6. DBMMO starts creating resources in its namespace according to the configuration. Resources such as:
 - **Persistent Volume Claims(PVC):** This specifies the requested resources in terms of storage. This information is used by the Kubernetes engine to dynamically provision a **Persistent Volume(PV)**
 - **MySQL Deployment:** This specifies the configuration for a MySQL deployment, the information for which is translated from the MySQL CR.
 - **Service:** This resource is created to enable other resources on the cluster to gain access to the MySQL instance. Services are generally used when no external access is required. Services can be accessed directly by name, which Kubernetes can resolve to an internal IP, providing a more human-readable access method.
 - **Ingress:** This resource acts as an externally facing service, allowing users and other services from outside of the Kubernetes cluster to reach the MySQL instance. This can be especially useful when a system that doesn't run on, or within the Kubernetes cluster needs to interact with a Kubernetes based system. (DBMMO has the option of enabling this if necessary)
7. Once all resources have been deployed, DBMMO, running from its pod will gather the information and state of all managed and owned resources.
8. DBMMO will update the MySQL CR in the Product namespace to reflect the status, including access information for the instance.
9. The Product pod can now interact with the MySQL instance and execute all necessary actions.

0.3.2 Azure Deployment

This functionality allows users to deploy a MySQL server running on Microsoft Azure, provided that the required fields have been correctly filled out in the *Spec.Deployment.AzureConfig*.

A check is in place to ensure that the fields have are populated, if valid credentials are provided DBMMO will begin creation and management. The shortened function can be found below:

```
1  func ValidateAzureConfig(dep *v1alpha1.DBMMOMYSQLDeployment) bool {
2      if dep.AzureConfig != nil {
3          if dep.AzureConfig.ClientID == nil || *dep.AzureConfig.ClientID == ""
4          {
5              return false
6          }
7          if dep.AzureConfig.TenantID == nil || *dep.AzureConfig.TenantID == ""
8          {
9              return false
10         }
11         if dep.AzureConfig.OAuthGrantType == nil
12         {
13             return false
14         }
15         /// SHORTENED FOR REPORT ///
16
17         return true
18     }
19     return false
```

Since Azure is an external service, DBMMO doesn't have a big workload. Mainly, it handles the creation and deletion of MySQLFlexibleServers, reconciling endpoints and connection details back to the status fields of the *DBMMOMYSQL* resource, from which other services or applications can access this information and invoke connections to the database.

The reconciliation function can be found below:

```
1 func (r *DBMMOMySQLReconciler) azureReconcileMysql(ctx context..., mysql *v1alpha1...) (ctrl.Result,
2                                                                                                     error){
3     if util.ValidateAzureConfig(mysql.Spec.Deployment) {
4         r.Log.Info("Reconciling MySQL on Azure", "Mysql.ServerName",
5             mysql.Spec.Deployment.ServerName)
6         // If Azure state doesn't indicate an error and hasn't been created, then create it
7         if !mysql.Status.AzureStatus.Created {
8             r.Log.Info("Mysql Azure instance creating, please wait",
9                 "mysql.ServerName", *mysql.Spec.Deployment.ServerName,
10                "Config:", *mysql.Spec.Deployment.AzureConfig)
11             server, err := util.CreateServer(ctx, mysql)
12             if err != nil {
13                 mysql.Status.AzureStatus.State = cachev1alpha1.AzureError
14                 if result, err := r.azureReconcileStatus(ctx, mysql); err != nil {
15                     return result, err
16                 }
17                 return ctrl.Result{
18                     RequeueAfter: constants.ReconcilerRequeueDelayOnFail}, err
19             }
20             //Update the status for future reference to the server
21             mysql.Status.AzureStatus.ServerInfo = cachev1alpha1.ServerInfo{
22                 Tags:      server.Tags,
23                 Location: server.Location,
24                 ID:          server.ID,
25                 Name:       server.Name,
26                 Type:      server.Type,
27             }
28
29             // set AzureStatus to created, prevent resource conflict on next loop
30             mysql.Status.AzureStatus.State = cachev1alpha1.AzureCreated
31             mysql.Status.AzureStatus.Created = true
32
33             // update the status to prevent next creation loop
34             if result, err := r.azureReconcileStatus(ctx, mysql); err != nil {
35                 return result, err
36             }
37             return ctrl.Result{}, nil
38         }
39
40     } else {
41         // if config isn't valid, error out, and exit the function
42         r.Log.Error(
```

```

43         fmt.Errorf("%v", "Spec.Deployment Azure field misconfiguration"),
44         "ensure data is valid",
45         "Deployment", mysql.Spec.Deployment)
46     return ctrl.Result{RequeueAfter: constants.ReconcilerRequeueDelayOnFail}, nil
47 }
48 r.Log.Info("Reconciled MySQL on Azure ", "Mysql.ServerName",
49 mysql.Spec.Deployment.ServerName)
50 return ctrl.Result{RequeueAfter: constants.ReconcilerRequeueDelay}, nil
51 }

```

Likewise, to *OnCluster* deployment, this functionality also has a full implementation of cleanup logic to ensure that all Azure resources are completely torn down. This is necessary to avoid generating costs and bills for unused services upon completing migrations.

```

1  func (r *DBMMOMySQLRec...) azureCleanup(ctx context..., mysql *cachev1alpha1...) (ctrl.Result,
2                                     error) {
3      // set the status to deleting to prevent recreation
4      mysql.Status.AzureStatus.State = cachev1alpha1.AzureDeleting
5      r.Log.Info("Deleting MySQL on Azure", "mysql.ServerName", mysql.Spec.Deployment.ServerName)
6      // delete the server
7      if _, err := util.DeleteServer(ctx, *mysql.Status.AzureStatus.ServerInfo.Name, mysql);
8      err != nil {
9          r.Log.Error(err,
10             "Failed to delete mysql Azure server",
11             "mysql.ServerName", mysql.Spec.Deployment.ServerName)
12         return ctrl.Result{RequeueAfter: constants.ReconcilerRequeueDelayOnFail}, err
13     }
14     // once deleted update the status to reflect
15     mysql.Status.AzureStatus.State = cachev1alpha1.AzureDeleting
16     mysql.Status.AzureStatus.Created = false
17     res, err := r.azureReconcileStatus(ctx, mysql)
18     if err != nil {
19         return res, err
20     }
21
22     r.Log.Info("Deleted MySQL on Azure", "mysql.ServerName", mysql.Spec.Deployment.ServerName)
23     return ctrl.Result{}, nil
24
25 }

```

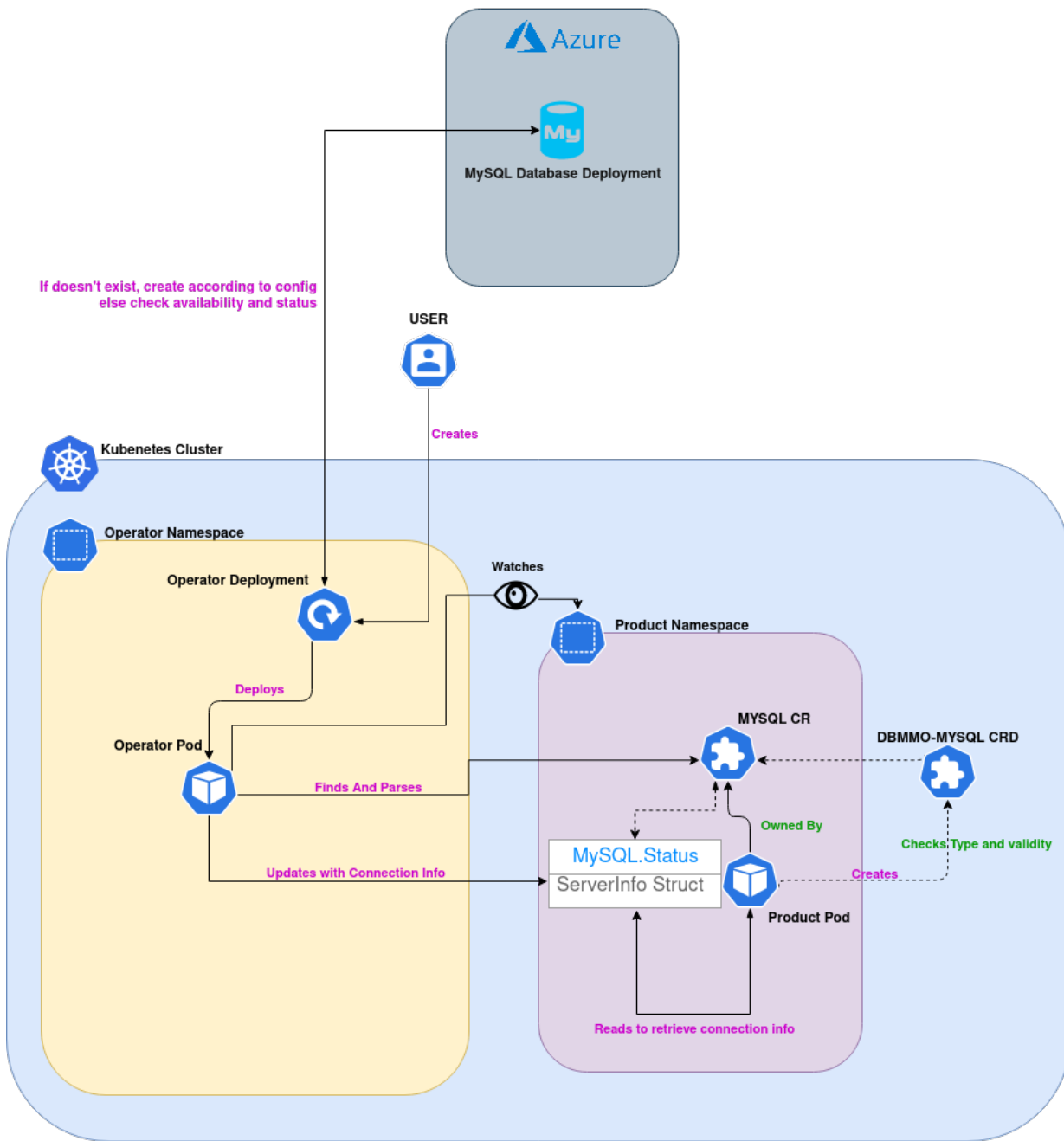


Figure 4: This Diagram illustrates the operator logic that deploys and manages the MySQL deployment on Azure.

The following logic is executed:

1. A cluster administrator/privileged user creates a deployment specifying an external repository containing the DBMMO image.

2. Kubernetes then translates the deployment and creates a pod, in which DBMMO will run.
3. DBMMO then begins to watch a specified namespace for a known resource, in this case, MySQL,
4. An application or product pod creates a MySQL deployment specifying Azure as the deployment type, using the CRD to type and validate the YAML.
5. DBMMO finds the newly created MySQL CR and parses the configuration contained within.
6. DBMMO checks whether required credentials have been provided within the resource, if not an error is returned and no further logic is undertaken.
7. If valid credentials are passed, DBMMO will then take the abstract deployment configuration and apply it to a new Azure MySQL server.
8. Once the server is ready, DBMMO will reach out and reconcile the status, returning it to the MySQL CR to enable connections to be made from other services or pods.

0.3.3 Database Migration

This functionality allows users to migrate databases between on cluster deployments and Azure and vice-versa. DBMMO handles all migration and cleanup of resources left after migration. Migration logic will only be executed if all requirements are met, mainly, confirmation and proper credentials for the respective cloud provider, Azure in this case.

The core package containing the code snippets shown below can be found on Github [\[LINK\]](#)

Migration will only take place once all resources on the desired (target "to" database) have been created and are fully available. This is done to ensure that database uptime is maintained with no interruption. Furthermore, a second check is put in place and no resource deletion will occur unless *ConfirmMigrate* is specified to *True*.

```
1  if mysql.Status.AzureStatus.Created &&
2  mysql.Spec.Deployment.ConfirmMigrate != nil &&
3  *mysql.Spec.Deployment.ConfirmMigrate {
4      r.Log.Info("Migration completed, starting Azure cleanup",
5      "mysql.Name", mysql.Name)
6      if result, err = r.azureCleanup(ctx, mysql); err != nil {
7          return result, err
8      }
9  }
```

The logic in the code snippet above is taken from the OnCluster reconcile switch. It ensures that firstly, an Azure database instance is reported as created and secondly, whether the *confirmMigration* field is specified as true. Unless these two conditions are met, no resources will be deleted. This logic is located at the very end of the switch logic, meaning that it will only be reached after all resources are created, all resources report readiness and after user deletion checks.

The other option in the switch logic is Azure, and the logic, similarly to the one for OnCluster reconciles, performs the same pre-cleanup checks.

```
1  case constants.MysqlDeploymentTypeAzure:
2      if result, err = r.azureReconcileMysql(ctx, mysql); err != nil {
3          return result, err
4      }
5      if result, err = r.azureReconcileStatus(ctx, mysql); err != nil {
6          return result, err
7      }
8      // If the object is being deleted then delete all sub resources
9      if mysql.DeletionTimestamp != nil {
10         r.Log.Info("Detected deletion timestamp, starting cleanup",
11             "mysql.Name", mysql.Name)
12         if result, err = r.azureCleanup(ctx, mysql); err != nil {
13             return result, err
14         }
15     }
16
17     if mysql.Status.Nodes != nil && mysql.Status.AzureStatus.Created
18     && mysql.Spec.Deployment.ConfirmMigrate != nil &&
19     *mysql.Spec.Deployment.ConfirmMigrate {
20         r.Log.Info("Migration completed, starting OnCluster cleanup",
21             "mysql.Name", mysql.Name)
22         result, err := r.OnClusterCleanup(ctx, mysql)
23         if err != nil {
24             return result, err
25         }
26         res, err := r.onClusterReconcileMysqlStatus(ctx, mysql, listOpts)
27         if err != nil {
28             return res, err
29         }
30     }
31 }
```

The code snippet above performs similar checks to OnCluster, additionally for the presence of nodes, whether an Azure MySQL instance is created and whether migration has been confirmed. After this, DBMMO can proceed to clean up OnCluster resources. Once the resources have been cleaned up, the status field for OnCluster will be updated to prevent further reconcile loops attempting to delete non-existent resources.

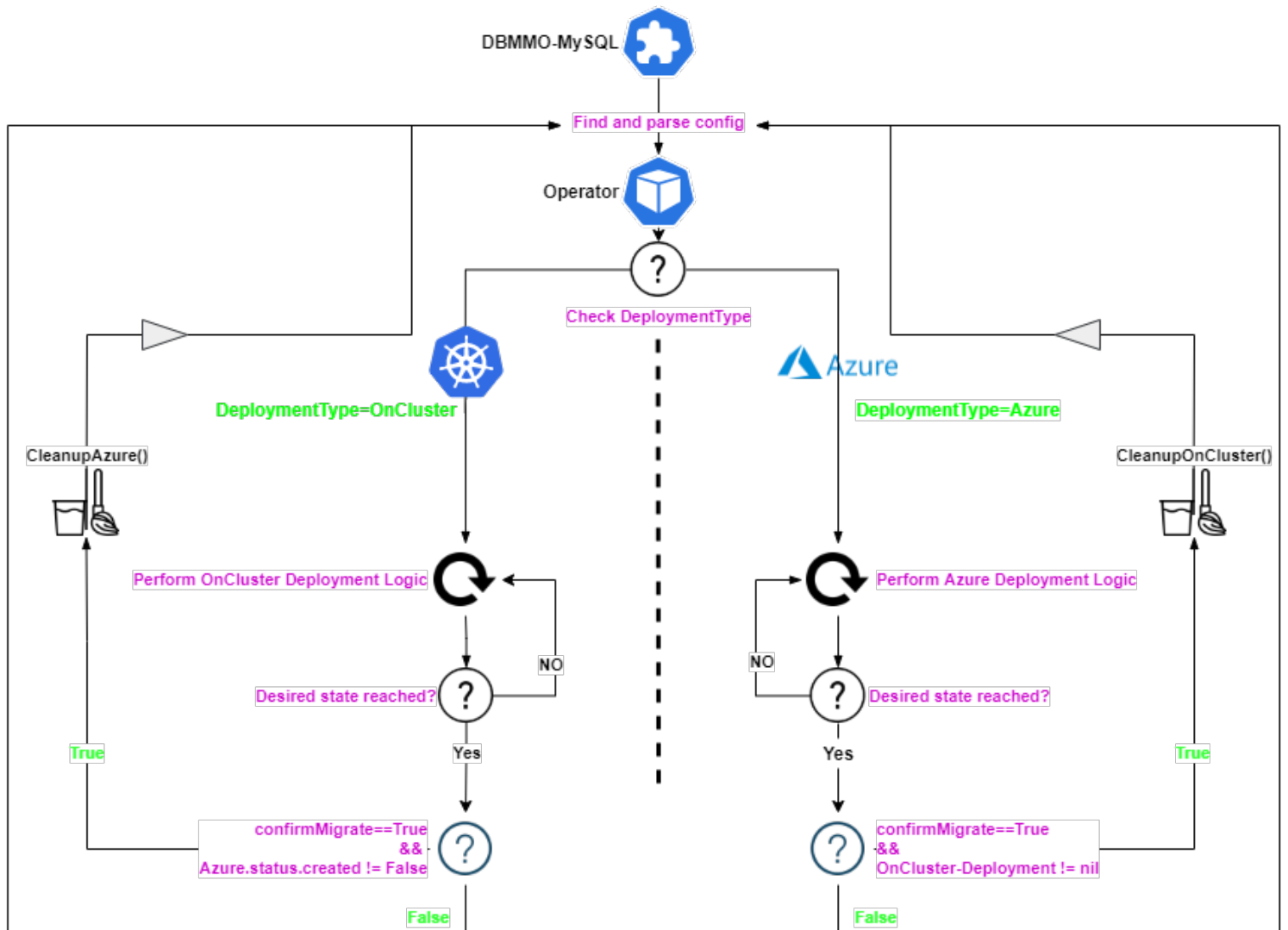


Figure 5: This Diagram illustrates the operator logic that migrates databases between environments.

The following logic is executed:

1. A DBMMO-MYSQL resource is found and read by DBMMO.
2. DBMMO parses the *deploymentType* field and decides on which logic to use for the respective deployment.
3. DBMMO proceeds to create the specified deployment and resources on the respective environment, either OnCluster or Azure.
4. DBMMO fully reconciles and waits for all resources of the specified type to reach full readiness.

5. Once all resources reach full readiness, DBMMO will check for the *confirmMigrate* flag, if set to True and if resources of another, currently not-specified resource exist, DBMMO will proceed to execute cleanup logic.
6. Once the resources have been cleaned up, the operator will continue to reconcile specified resources as normal.

0.4 Issues Encountered

0.4.1 Azure

During the development of this project, multiple issues have arisen with regards to Azure.

Azure was quite difficult to get started with, the lack of documentation was quite striking, and if something was documented, then it was lacking information, at worst it was counter-productive, especially spending hours searching through official documentation websites and not finding even a mention of the information that was needed.

Overall, most questions regarding Azure were answered on stack overflow or other forums.

The number of ID's, secrets, and other configurations just to connect to Azure was difficult to set up, especially when the documentation that was there, was largely out of date.

Often resources created through the API were only visible when called through the API, despite being created using the same *subscriptionID* (which would mean that they are assigned to the same account). This meant that vast amounts of Azure credit were wasted because the resources couldn't be accessed and deleted, neither through the CLI or Azure portal.

This meant that the limited trial credit was being used up significantly faster than expected, through no fault or effect of actions by the developer.

Another issue encountered with Azure was the lack of permissions when trying to use college assigned credit on the respective *subscriptionID*. The role automatically assigned to the account by the college was lacking in permissions and privilege to fully avail of Azure. The project required user permissions to manually create applications in the *Azure Active Directory* as well as secrets to allow applications to be authenticated through a specified role (which also had to be manually created as a prerequisite). These actions weren't possible with the default permissions assigned to the student account. Getting these permissions assigned was a lengthy process, as there was no person in charge of the WIT organization that could be contacted.

Another option that was explored was attempting to use friends or siblings student accounts to use the same free trial that the initial Azure development could proceed with. Yet again, being unsuccessful due to account limitations, mainly, The "borrowed" account could only be logged in on one machine at a time, limiting access for the owner which wasn't a viable solution especially when this would block others from their college work.

Initially, development was possible because of the two separate subscriptions. The subscription that had full access was granted by Azure in a 90 day, credit limited free trial, which would cease to work after either the day or credit limit was reached. This subscription was separate from the student subscription, which is vastly limited resource access: the 100USD

which is assigned to it couldn't be used.

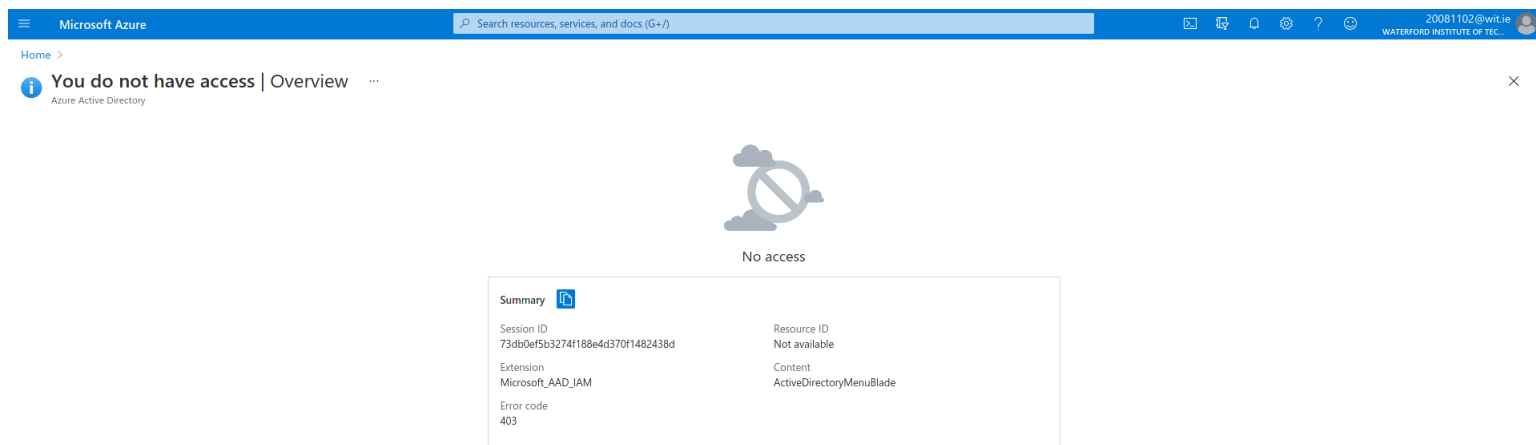


Figure 6: no permissions to access Azure Active Directory

The figure above shows the error that is returned when attempting to access the Azure Active Directory from the student subscription, preventing any further development and resource set up.

Overall dealing with Azure proved to be challenging, but not in aspects that were expected before development started. The lack of clarity in all aspects of it was frustrating, with a vast amount of time having to be dedicated to getting permissions, configure specific settings and get the proper documentation, which as mentioned above, was lacking. While the time spent on programming the solution was minuscule in comparison to the amount that was essentially wasted on factors that largely couldn't be influenced.

In an attempt to resolve the account access problem the head of computing in WIT was contacted, in hopes to find out who was in charge of the WIT Azure organisation, so that proper roles and permissions could be assigned to the student subscription and account to grant access to required resources for authentication. However, that process was also quite time-consuming and eventually proved unfruitful.

Another attempted resolution was to try and gain another free trial and credit by attaching a private debit card to a new account. This again proved unsuccessful as can be seen from the figure below. Azure most likely verifies machines and addresses such that they cannot be perpetually exploited for free credit.

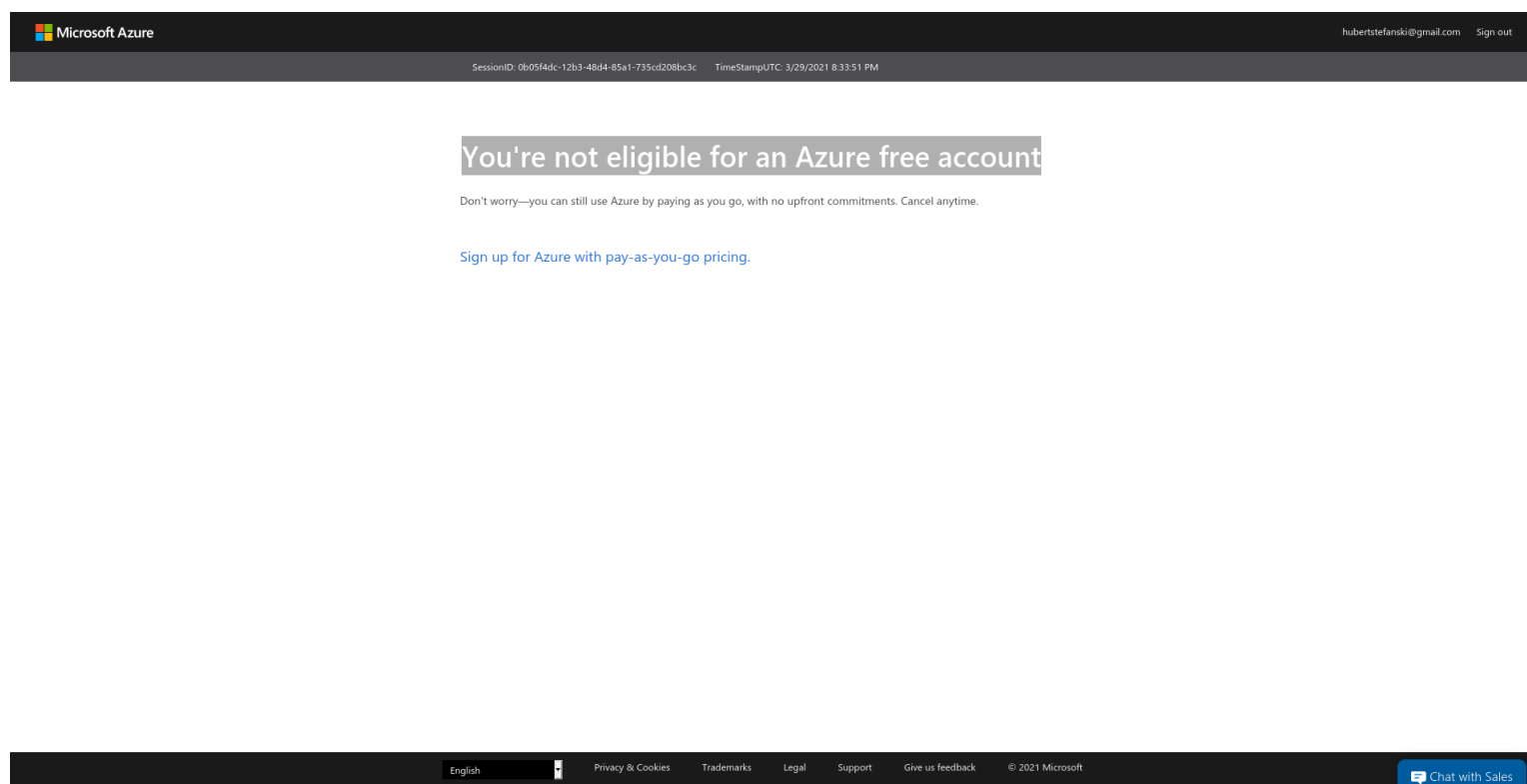


Figure 7: not eligible for an Azure free account

Numerous attempts were also made to authenticate DBMMO through alternative ways, trying to bypass resources to which the student subscription didn't have access. This again proved unsuccessful, as the common denominator for all authentication methods was the requirement to perform create or read actions on certain fields from the *Azure Active Directory*.

The final and ultimate solution to this issue was to add a personal debit card and pay for the services. This solution was the last option as it was never clear why the original credit was used up so quickly, therefore there was significant hesitation to take this step as without proper supervision the Azure services bill could escalate into hundreds of Euro. Proper controls had to be put in place around cost alerts and budget limiting to prevent Azure from overcharging the debit card.

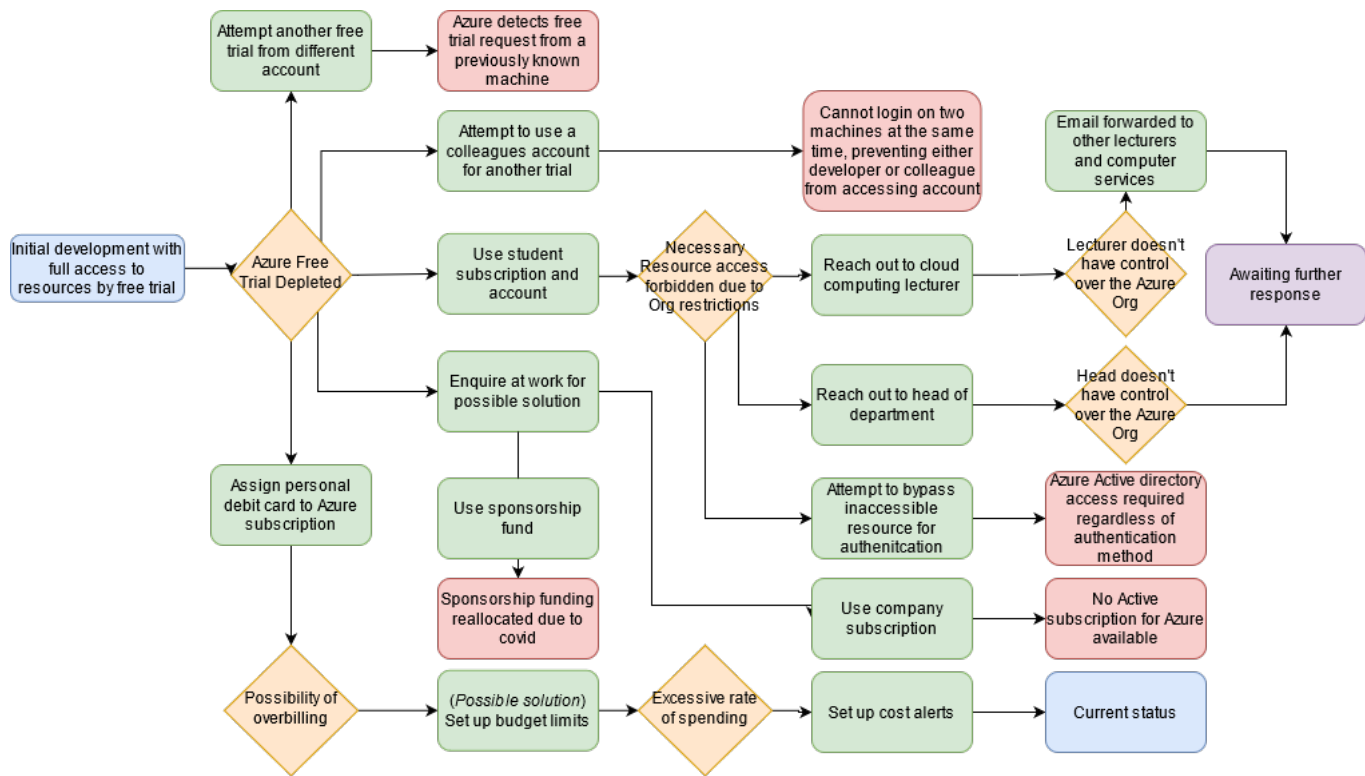


Figure 8: explored solutions to mitigate azure credit issues

0.4.2 Kubernetes Cluster

For the duration of this project, Minikube was used for local Kubernetes cluster deployment, this was a decision that made more sense than attempting to deploy a cloud-hosted server, saving both time and money during the development process. Minikube has the advantage of being lightweight, allowing for local development and deployment for testing. Minikube clusters can be provisioned within a few minutes and torn down almost immediately, which makes it an ideal sandbox environment. However, choosing to use a locally deployed Kubernetes cluster means that the administration and subsequent issues will also have to be resolved manually. An issue arose with Minikube after updating operating system packages, which as a result started breaking Minikube cluster deployments, the cluster started having issues pulling down images such as MySQL or *Nginx-ingress* add-on controllers for the cluster itself from Docker Hub, however, this issue did not exist locally, therefore, it was determined that the issue lies within the automated docker configuration within Minikube itself. Encountering this issue meant development couldn't proceed as DBMMO depends on images

such as *mysql:5.6* to create deployments of MySQL servers. After hours of investigation into what could be causing this issue, a choice was made to completely wipe the Minikube cluster and reinstall the application again, after which the cluster worked properly, however, this consumed a significant amount of time and was a hindrance to the development for several days while troubleshooting and investigating possible solutions. Even after finding a fix, the issue appeared again after a few days. The solution after this was to simply implement conditional ingress creation to avoid using the non-working controllers.

0.5 Conclusion And Evaluation

In Summary, this project set out to resolve the identified gaps, issues and shortcomings of current database management systems within cloud-native environments, specifically in cloud applications or services based on containerisation. The project was successful in doing so by implementing the Database Management and Migration Operator (DBMMO). DBMMO provides users with the ability to manage their deployments both on the Kubernetes cluster as well as off-cluster, in public cloud environments the likes of Azure. DBMMO proved that it is possible to automate the provisioning and migration of databases between environments. DBMMO has also shown that it is capable of handling databases of all types, doing so by giving developers an abstract interface to declare non-environment specific options for database configuration. The generalised configuration options open the door for future expansion into the support for other database engines and cloud providers. DBMMO delivered on all goals set out prior, reaching full functionality as outlined in the planning stage. The project, as it stands currently is fit for purpose and ready for real-world use.

0.5.1 Learnings

I found myself significantly deepening my knowledge on all of the technologies that have been used during the development of this project. There is an immense value to be gained from developing an entirely new and fresh project with new technologies, it offers an entirely different perspective and experience than that offered by an already existing project. Perhaps the most significant learning experience was achieved by learning through trial and error. Despite coming into this project with *reasonable* knowledge of some of the technologies, I still found myself making many mistakes through, not necessarily wrong assumptions, but through assumptions that were formed through prisms of experience with other projects that utilised similar technologies. Starting this project from the very beginning, from planning design and architecture through to code implementation brought an entirely new perspective. Certain aspects that seemed difficult turned out to be quite straightforward and others that seemed trivial turned out to be anything but, especially when starting to consider different use and error cases.

Starting a project from scratch, especially of this magnitude, is quite overwhelming. When looking at the overall scale of the project and the goals set out in the plan it was difficult to imagine a clear path forward. However, I found that splitting up these goals into smaller and smaller objectives was the easiest way forward. Initially having a look at the end goal, and then asking a series of questions until a clear task could be defined to push the project closer to the goal. At all times during development there were always three questions that were posed at every new stage, mainly:

1. Where and what is the goal?
2. Where is the project now?
3. How to get a step closer to the goal? (Identify gaps and tasks in between)

There were many unknowns during the development of this project but the iterative process helped to get through and closer to completion.

The Agile methodology chosen for this project showed its worth during development. Using Agile allowed plenty of flexibility, which proved especially useful when implementing more recent features by giving the ability to return to a certain feature implemented in a previous sprint/iteration and refactor it to be more suitable for current requirements. As expected, planning for specific implementation details proved useless, many of the things that were planned stopped making sense once the project started taking shape, however, the general ideas remained the same, thus showing the value of keeping an open mind to alternative

solutions and not boxing oneself into a corner with concrete decisions made prior.

Sprint reviews allow to identify gaps, but also to re-prioritise certain functionality, as well as reject some. An example of a rejected piece of functionality was MySQL table support, where the basic idea was to allow the operator to create and manage tables within the database deployment. Table management seemed like a great idea initially, but after revisiting it during a sprint review, a few issues became apparent:

1. Security, adding table support and the ability to invoke queries provides another attack vector on possibly sensitive data.
2. Resource conflict, the operator could potentially delete or overwrite data or changes made by external services that require this database.
3. Out of scope, the operator should only be concerned with databases and migration and nothing else to do with the data within.
4. Feature creep, introducing more and more obscure pieces of functionality will bog down the operator. Instead, the operator should be kept light and dedicated to one purpose.

This would be a likely feature that would be supported by DBMMO, but the sprint review identified that the risks and drawbacks outweighed the benefit. This was an overall lesson in software engineering, showing that engineering is more than just writing code and that first and foremost things must be thought through.

In conclusion, the knowledge taken away from this project is a mixture of soft and hard skills, from learning to better understand code, its implications, use and misuse cases and learning to think more architecturally design-wise to gaining a deeper understanding of Kubernetes, Docker, Operator-SDK and all the other technologies that are currently the cutting edge in the cloud.

0.5.2 Further Development

0.5.2.1 Data Migration Between Cloud Environments

For future development, the aspect of data migration between cloud environments should be addressed. There are many ways of approaching the specific implementation for this functionality, however, most likely the easiest solution would be to directly make use of the Azure Migration Service (AMS), for which DBMMO has been already prepared through its database migration logic. The Azure Migration Service works by continuously syncing the Azure-deployed database with the previous database, once all data is synced, the previous database endpoint will deactivate and traffic will be redirected to the Azure managed database. At the time of writing, this couldn't be automated as the Azure-SDK lacked the API for interaction with this service but could be achieved with a manual step. Attempting to automate the manual procedure (UI navigation etc.) would prove too fragile and unfeasible for the operator.

0.5.2.2 Extend Configuration

Currently, DBMMO is quite configurable, the most popular and necessary options are available to be for user definition, however, there are still options that are yet to be made configurable through custom resources. Currently, DBMMO works on a best guess basis for setting default values to fields that weren't defined by the user. Further development would add more configuration options for a wider range of fields to be overridden.

0.5.2.3 Support Other Cloud Providers

The current state of the project leaves a lot of room for other cloud providers, the likes of Google Cloud Platform or Amazon Web Services, which could all be eventually implemented. Extending this support to other cloud providers gives DBMMO a wider user base to which it would appeal, as well as giving existing users the possibility to extend or migrate their databases to a broader selection of cloud providers. This extended support would come at an increased complexity as all use cases would have to be considered for migrations between various combinations of cloud providers, thus possibly, an ideal solution would be to dedicate support to only the top N amount of cloud providers to decrease the exponentially increasing workload that would be involved with onboarding of more and more providers.

0.5.2.4 Support Other Types of Databases

The Operator-SDK and Kubernetes itself isn't affected by what type of application is deployed using it, as long as it can be compiled into a runnable image. Therefore this lends DBMMO to be vastly expanded to a broad selection of other types of databases, from blob storage, NoSQL, object-oriented or networked amongst many, given that they can be containerized. To achieve this, new controllers would have to be implemented for each type of database.

0.5.2.5 Support Running OnCluster Deployments as Replicated Stateful Applications

The current implementation of DBMMO only supports a single stateful application. In further development this should be refactored to support a replicated stateful deployment, meaning that the state of a single instance of the database should be reflected amongst all replicated deployments. As is usually the case with other replicated and stateful applications, the topic begins to gain exponential complexity with scale, hence why it was omitted in initial development. Including it in the initial development would have likely taken a significant effort, not to mention even greater labour in further development when dealing with migration.

0.6 References

Casey, K. 2019. Kubernetes By The Numbers: 13 Compelling Stats. [online] Available at: <<https://enterpriseproject.com/article/2019/7/kubernetes-statistics-13-compelling>> [Accessed 5 October 2020].

Kubernetes. 2021. Kubernetes Documentation. [online] Available at: <<https://kubernetes.io/docs/home>> [Accessed 2 May 2021].

Reference 3 —————

0.7 Further Reading/Sources

osdk-getting-started[LINK]

5-container-alternatives-to-docker[LINK]

what-is-kubernetes[LINK]

golang-get-started[LINK]

0.8 Appendices

0.8.1 Implementation & Iterations

The development methodology used for the development of DBMMO was agile. This is an iterative approach based on weekly sprints. At the beginning of each week (sprint) a certain set amount of tasks was created to be completed during the sprint. Tasks were created and tracked using JIRA. Initially, the project was tracked using Trello, but a decision was made later on to migrate to JIRA as I felt that this provided a better user experience. Each sprint focused on the breakdown and incremental implementation of separate tasks to build on to functionality, primarily the three functional requirements:

1. On cluster deployment - A locally accessible MySQL instance, hosted on the same cluster as DBMMO.
2. Cloud database deployment - A database created and reconciled on a public cloud provider, such as AWS or Azure.
3. Database migration - Fully implemented upgrade logic to allow movement of databases between environments, OnCluster -> Azure, Azure -> OnCluster, Azure -> *Other cloud*(in the future).

0.8.1.1 Iteration 1

Week : 1

Sprint : 1

Release Notes:

- Set up CI/CD and other automation.
- Generated Custom Resource Definitions.
- Generated API.
- Implemented Local Mysql Deployment controller and reconciler.
- Implemented all necessary subresources (Deployment, Container, Pod, Persistent Volume & Claim, Service)

This sprint focused on implementing basic provisioning and reconciliation logic to deploy MySQL using Kubernetes and the Operator-SDK. The main goal being for DBMMO to be able to create the necessary resources, manage them and make them accessible in such a way that they can be used if they were deployed manually.

0.8.1.2 CI/CD (Continuous Integration/Continuous Development)

Part of the setup for this project was to include CI/CD tools and automation to aid in the development and testing of DBMMO. The implemented automation consisted of the following:

Go Builder - This Github workflow will check that the project can be successfully built with the included changes, this workflow is run on every pull request and commit to the pull requests branch, as well as on every commit to the master branch.

```
1     name: Go
2     on:
3       push:
4         branches: [ master ]
5       pull_request:
6         branches: [ master ]
7     jobs:
8       build:
9         runs-on: ubuntu-latest
10        steps:
11          - uses: actions/checkout@v2
12          - name: Set up Go
13            uses: actions/setup-go@v2
14            with:
15              go-version: 1.15
16          - name: Build
17            run: go build -v ./...
18        #TODO Reactivate this once tests are implemented
19        #   - name: Test
20        #     run: go test -v ./...
```

This workflow also enables unit/end-to-end testing, for now, this was disabled as there was no test implemented at this stage of development.


Go Releaser - This Github workflow automatically builds multi-architecture images on release with a binary for each architecture, along with a checksum of the project to ensure integrity.


```
1       name: goreleaser
2   on:
3     push:
4     tags:
5       - '!*'
6   jobs:
7     goreleaser:
8       runs-on: ubuntu-latest
9       steps:
10        - name: Checkout
11          uses: actions/checkout@v2
12          with:
13            fetch-depth: 0
14        - name: Set up Go
15          uses: actions/setup-go@v2
16          with:
17            go-version: 1.15
18        - name: Run GoReleaser
19          uses: goreleaser/goreleaser-action@v2
20          with:
21            version: latest
22            args: release --rm-dist
23          env:
24            GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }
```

DockerHub & Github Integration - This automation is a feature shipped with hub.docker.com Which is the designated container image repository hosting this project. Dockerhub provides users with the ability to set integrations and rules for Github commits, this means that an image will be automatically created and hosted based on the *latest* state of the master Github master branch.

Automated Builds

Autobuild triggers a new build with every **git push** to your source code repository. [Learn More.](#)

 [HubertStefanski/database-management-and-migration-operator](#) | Use Docker Hub's infrastructure | Autotests: Off

Docker Tag	Source	Latest Build Status	Autobuild	Build caching	
latest	master	SUCCESS	✓	✓	

Recent Builds




 Build in 'master' (c2bce140)  latest [c2bce14](#)  24 minutes ago

Figure 9: Dockerhub automated build section

0.8.1.3 Implementing The Controller

Initially, the first controller that was to be implemented was the DBMMO-Controller (database management and migration operator). This controller would be responsible for managing the controller deployment on the cluster, however, after extended consideration, it became evident that this controller would be largely out of scope for this project and would add unnecessary complexity to an already complex subject. Instead, the controller would be run from a single *deployment* kind resource, rather than its unique kind. Instead, the first controller to be implemented was the *DBMMOMYSQL-Controller*, this being the newly created custom resource kind for a MySQL database. Once DBMMO is run, it will start looking for a resource of this kind in its namespace, if one is not found, it will log an adequate message and continue onto the next loop until a resource is found, once found, the resource will be read by DBMMO and have it's configuration read from the *yaml*. This iteration still being an early stage of development, means that DBMMO will simply read the resource and create adequate minimum resources for the MySQL instance to be available, with the configuration being a goal for future iterations. This controller, as all future controllers is called from the *main.go* file, which is the file that sets up all controllers with the controller manager.

```
1     if err = (&mysql.DBMMOMySQLReconciler{
2         Client: mgr.GetClient(),
3         Log:     ctrl.Log.WithName("controllers").WithName("DBMMOMySQL"),
4         Scheme: mgr.GetScheme(),
5     }).SetupWithManager(mgr); err != nil {
6         setupLog.Error(err, "unable to create controller", "controller",
7             "DBMMOMySQL")
8         os.Exit(1)
9     }
```

The code block above being responsible for adding the DBMMOMYSQL reconciler to the manager controller. The controller code for this iteration can be found *here*

```

1 foundDeployment := &appsv1.Deployment{}
2 err = r.Get(ctx, types.NamespacedName{Name: mysql.Name, Namespace:
3 mysql.Namespace}, foundDeployment)
4 if err != nil && errors.IsNotFound(err) {
5     // Define a new deployment
6     dep := r.getMysqlDeployment(mysql)
7     log.Info("Creating a new Deployment",
8         "Deployment.Namespace",
9         dep.Namespace, "Deployment.Name", dep.Name)
10    err = r.Create(ctx, dep)
11    if err != nil {
12        log.Error(err, "Failed to create new Deployment",
13            "Deployment.Namespace", dep.Namespace,
14            "Deployment.Name", dep.Name)
15        return ctrl.Result{}, err
16    }
17    // Deployment created successfully - return and requeue
18    return ctrl.Result{Requeue: true}, nil
19 } else if err != nil {
20     log.Error(err, "Failed to get Deployment")
21     return ctrl.Result{}, err
22 }
23 size := mysql.Spec.Size
24 if *foundDeployment.Spec.Replicas != size {
25     foundDeployment.Spec.Replicas = &size
26     err = r.Update(ctx, foundDeployment)
27     if err != nil {
28         log.Error(err, "Failed to update Deployment",
29             "Deployment.Namespace",
30             foundDeployment.Namespace, "Deployment.Name",
31             foundDeployment.Name)
32         return ctrl.Result{}, err
33     }
34     return ctrl.Result{Requeue: true}, nil
35 }

```

As can be seen from the code snippet above, the code is quite complex and needs to account for and update any field separately, this is something that will be improved in future iterations, for an initial and non-configurable deployment this is satisfactory. At a high level, the code above attempts to retrieve the *Deployment* object from the cluster using its declared name and namespace from the DBMMOMYSQL resource to which the controller belongs, If one isn't found it will be created using the hard coded configuration from the following function:

```

1  func (r *DBMMOMySQLReconciler) getMysqlDeployment(m *cachev1alpha1.DBMMOMySQL)
2  *appsv1.Deployment {
3      ls := getLabels(m.Name)
4      replicas := m.Spec.Size
5      dep := &appsv1.Deployment{
6          TypeMeta: metav1.TypeMeta{},
7          ObjectMeta: metav1.ObjectMeta{
8              Name:      m.Name,
9              Namespace: m.Namespace,
10             },
11             Spec: appsv1.DeploymentSpec{
12                 Replicas: &replicas,
13                 Selector: &metav1.LabelSelector{
14                     MatchLabels: ls,
15                 },
16                 // FURTHER CONFIGURATION, CUT FOR REPORT
17             },
18             },
19             },
20             },
21             },
22             },
23         }
24         // Set Mysql instance as the owner and controller
25         _ = ctrl.SetControllerReference(m, dep, r.Scheme)
26         return dep

```

This Function will eventually host calls to other functions that will be responsible for the configuration of each field according to the declared DBMMOMYSQL resource. Once the deployment resource is created, further calls will compare the resource found with the actual state, if these differ they will be updated such that they are no longer mismatched.

The created deployment can be seen in the figure below

mysql-deployment-7967fb4fcd-9cv8z

Metadata

Name

mysql-deployment-7967fb4fcd-9cv8z

Namespace

dbmmo-ns

Created

Feb 14, 2021

Age

31 seconds ago

UID

d67534df-b3fd-4185-8104-3a28be154cca

Labels

app: dbmmo-mysql
mysql_cr: dbmmo-mysql
pod-template-hash: 7967fb4fcd

Resource information

Node

minikube

Status

Running

IP

172.17.0.8

QoS Class

BestEffort

Restarts

0

Conditions

Type	Status	Last probe time	Last transition time	Reason	Message
Initialized	True	-	31 seconds ago	-	-
Ready	True	-	26 seconds ago	-	-
ContainersReady	True	-	26 seconds ago	-	-
PodScheduled	True	-	31 seconds ago	-	-

Figure 10: DBMMO managed deployment created and running a pod on the Kubernetes cluster

mysql-deployment-7967fb4fcd-9cv8z > Shell

Shell in mysql-container ▾ in mysql-deployment-7967fb4fcd-9cv8z

```

root@mysql-deployment-7967fb4fcd-9cv8z:/# mysql -ppassword
Warning: Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.6.51 MySQL Community Server (GPL)

Copyright (c) 2000, 2021, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> 
```

Figure 11: Mysql available in the container

In the image above we can see that MySQL is available and running, the executed command starts a MySQL shell. This pod is available to all other pods on the cluster through the attached service, which is also managed by DBMMO.

0.8.1.4 Iteration 2

Week : 2

Sprint : 2

Release Notes:

- Added Roles, ServiceAccounts and general permissions for DBMMO enabling it to run from within the cluster.
- Refactored controllers to use `controllerruntime.CreateOrUpdate()`
- Removed bespoke create or update logic.
- Implemented time delay for reconcile loops to save resources and network usage.
- Fixed PV reconciler.
- Implemented Mysql status reconciliation.
- Improved logging in reconcilers/controller.

This weeks sprint focused primarily on improvements to the reconcilers and controller, with some minor bug fixing and testing.

0.8.1.5 Controller Refactoring

After reviewing the implementation of the reconciler that was introduced in the first sprint it was decided that there was a need to clean up the logic and find a more robust solution. This functionality can be achieved through

`controllerutil.CreateOrUpdate()`

provided by the controller-runtime Kubernetes library. The use of this function will significantly decrease the amount of code necessary to manage each resource, improving efficiency both in runtime as well as the development of future resources. The Reconcile function is now significantly shorter, as can be seen from the figure below.


```

1  func (r *DBMMOMySQLReconciler) reconcileMysqlService(ctx context.Context,
2  m *cachev1alpha1.DBMMOMySQL, listOpts []client.ListOption) (ctrl.Result, error) {
3      // Define a new service
4      service := model.GetMysqlService(m)
5
6      _ = ctrl.SetControllerReference(m, service, r.Scheme)
7
8      r.Log.Info("Reconciling service", "Service.Namespace", service.Namespace,
9      "Service.Name", service.Name)
10     _, err := controllerutil.CreateOrUpdate(ctx, r.Client, service,
11     func() error {
12         service.Spec.Ports = []corev1.ServicePort{
13             {
14                 Name:      constants.MysqlContainerPortName,
15                 Port:      constants.MysqlContainerPort,
16                 Protocol:   corev1.ProtocolTCP,
17                 TargetPort: intstr.FromString(
18                     constants.MysqlContainerPortName
19                 ),
20             },
21         }
22         return nil
23     })
24
25     if err != nil {
26         r.Log.Error(err, "Failed to reconcile Service",
27         "Service.Namespace", service.Namespace, "service.Name", service.Name)
28         return ctrl.Result{}, err
29     }
30
31     return ctrl.Result{Requeue: true}, nil
32
33 }

```

0.8.1.6 Fixed PV/PVC reconciler

The persistent volume and persistent volume claim reconcilers required bug fixing. The initial implementation would error out after the initial reconcile/creation loop has finished running, this is since persistent volumes aren't supposed to be changed once they are created, primarily due to their resource management and potential problems therein. After further consideration and investigation into how PVs and PVCs worked, it was decided that the persistent volume reconcile had to be removed as it was no longer necessary as Kubernetes

supports dynamic persistent volume creation, meaning that a persistent volume will be created with the resources requested from a persistent volume claim. The persistent volume claim reconciler still required refactoring, this was to account for the fact that some of its fields cannot be updated, the new logic will instead check if this PVC already exists on the cluster, if not, it will create the resource. Note the code snippet below still uses bespoke logic to reconcile the resource, rather than using the CreateOrUpdate() function, which would not be suitable for this purpose.

```
1      // Define a new PersistentVolume
2          pvc := model.GetMysqlPvc(m)
3          r.Log.Info("Reconciling PVC", "Pvc.Namespace", pvc.Namespace,
4              "Pvc.Name", pvc.Name)
5          _ = ctrl.SetControllerReference(m, pvc, r.Scheme)
6          r.Log.Info("Creating a new PersistentVolumeClaim",
7              "PersistentVolumeClaim.Namespace", pvc.Namespace,
8              "PersistentVolumeClaim.Name", pvc.Name)
9          if err = r.Client.Create(ctx, pvc); err != nil {
10              r.Log.Error(err, "Failed to create
11                  new PersistentVolumeClaim", "PersistentVolumeClaim.Namespace"
12                  , pvc.Namespace,
13                  "PersistentVolumeClaim.Name", pvc.Name)
14              return ctrl.Result{}, err
15          }
16
17          r.Log.Info("PersistentVolumeClaim created",
18              "PersistentVolumeClaim.Namespace", pvc.Namespace,
19              "PersistentVolumeClaim.Name", pvc.Name)
20
21          return ctrl.Result{Requeue: true}, nil
```

0.8.1.7 Iteration 3

Week : 3

Sprint : 3

Release Notes:

- Refactor reconciler and controller to separate out onCluster and Cloud reconciliation logic
- Implement cleanup logic for OnCluster deployment.
- Make Mysql externally accessible through Ingress.
- Move credentials to EnvFrom, Credentials can now be passed in through this option from a secret to avoid plaintext.
- Restructure Mysql CRD , deploymentType is now a subfield of Spec.Deployment.

This sprint focused on further improvements to the code base and structure to facilitate the eventual implementation of cloud database and provisioning.

0.8.1.8 Reconciler and Controller Refactoring

This refactoring added code to check the type of deployment that is being requested by the MySQL resource, which will then subsequently run the corresponding reconciler logic, either *OnCluster* or *Cloud*(TBA).

```
1      switch depType := *mysql.Spec.Deployment.DeploymentType; depType {
2          case constants.MysqlDeploymentTypeOnCluster:
3              if result, err := r.onClusterReconcileMysqlPVC(ctx, mysql);
4              err != nil {
5                  return result, err
6              }
7              if result, err := r.onClusterReconcileMysqlService(ctx, mysql);
8              err != nil {
9                  return result, err
10             }
11             // ..... SHORTENED FOR REPORT
12         case constants.MysqlDeploymentTypeAzure:
13             r.Log.Info("Specified deployment type is not currently supported,
14             please enter a supported type",
15                 "DeploymentType", mysql.Spec.Deployment.DeploymentType)
16         default:
17             r.Log.Error(fmt.Errorf("%v", "Unrecognized deployment type"),
```

```
18         "ensure correct spelling or supported type",
19         "DeploymentType", mysql.Spec.Deployment.DeploymentType)
20     }
21     return ctrl.Result{Requeue: true,
22     RequeueAfter: constants.ReconcilerRequeueDelay}, nil
```

0.8.1.9 OnCluster Cleanup Logic

This feature was required for the cluster to be able to clean up subresources of MySQL, this is done to ensure that all resources will be removed once a database has been migrated and there won't be any future conflicts in the case of another migration. Cleanup is done conditionally and only when a deletion timestamp has been created on the parent resource.

0.8.1.10 Implement Secret and EnvFrom for MySQL Credentials

In general, it is bad practice to store credentials in plain text. In-built Kubernetes resources called *Secrets* are responsible for maintaining a level of security in deployments, they can be changed at any time and passed into the application as a resource, or in this case, as an *EnvFrom* (Environment Variable From) field in the MySQL resource, Kubernetes will read in the resource and pass it as an environment variable to the pod.

0.8.1.11 Make MySQL Accessible Through an Ingress

Ingresses are In-built Kubernetes resources that allow externally facing addresses to act as endpoints for users and services outside of the Kubernetes cluster. The Ingress can be retrieved from outside the cluster by:

```
kubect1 get ingresses --all-namespaces
```

which will return the following:

dbmmo-ns	dbmmo-MySQL-ingress	<none>	MySQL-host-name	192.168.39.146	80
----------	---------------------	--------	-----------------	----------------	----

0.8.1.12 Iteration 4

Week : 4

Sprint : 4

Release Notes:

No release this sprint.

Bug fixing and experimenting with Azure.

This sprint focused mainly on experimentation and research into Azure, with some prototype code written to test the required fields and variables that will be used to create the server through DBMMO.

0.8.1.13 Experimental code

This week, the sprint focused mainly on researching and experimenting with the Azure software development kit for Go. a few helper and setup functions were created to aid in a future implementation, such as:

```
1      func getServersClient(m *v1alpha1.DBMMOMySQL) MySQL.ServersClient {
2
3          serversClient := mysql.NewServersClient(
4              *m.Spec.Deployment.AzureConfig.SubscriptionID)
5          a, _ := GetResourceManagementAuthorizer(m)
6          serversClient.Authorizer = a
7
8          return serversClient
9      }
```

The code snippet above creates a new server-client for Azure, such that it can be passed into other functions and used to invoke actions on the cloud. The *SubscriptionID* is a unique identifier that has to be created manually by the user on the Azure portal, it is then passed into the Custom resource and read by DBMMO.

0.8.1.14 Iteration 5

Week : 5

Sprint : 5

Release Notes:

- Implement Azure MySQL server creation and deletion
- Add helper methods for Azure deployment
- Add CR validation for Azure deployments
- Extend logging
- Fix types, previously empty fields would error out the operator on empty

This sprint focused on implementing Azure deployments for MySQL, mainly create and delete functionality.

0.8.1.15 Implement Azure MySQL server creation and deletion

This functionality implemented the ability to create and delete MySQL flexible servers on Azure, provided that the credentials and details passed into the object were configured correctly on Azure itself. The code for which can be found [here](#)

0.8.1.16 Add CR validation for Azure deployments

This implementation provides a simple validator to ensure that the fields passed into the *AzureConfig* struct are valid (non-empty and non-nil). This is required to prevent DBMMO from making unnecessary calls to the API with non-valid or missing credentials, thus preventing rate throttling or timeouts from Azure. The code for this validation can be found [here](#)

0.8.1.17 Extend logging

This addition is largely due diligence. Previously, certain functions in DBMMO were lacking adequate logging, thus making troubleshooting difficult and hard to pinpoint bugs and errors, with these changes DBMMO has become more verbose when running, thus making troubleshooting easier in the future. However, this will have to be refactored in the future to

provide more control over the verbosity of the logs, i.e users may want to set DBMMO to only log *Warn* or *Error* lines as opposed to every *Info* or *Debug* call.

0.8.1.18 Fix types, previously empty fields would error out the operator on empty

This fix addressed certain errors that would appear in runtime, freezing the operator if certain status fields were left empty on status reconciles. This was simply fixed by adding *omitempty* to the JSON annotations in respective structs, thus DBMMO will no longer freeze if a field is empty.

0.8.1.19 Iteration 6

Week : 6

Sprint : 6

Release Notes:

No release this sprint,

Encountered issues with table implementation for Azure.

The work this week was mainly focused on a bug discovered with the azure controller. This bug exists in the pre-create checks, it fails to return an active and running instance, hence it cannot determine if a server exists already and attempts to create one, but fails due to conflicts in resource names.

0.8.1.20 Iteration 7

Week : 12

Sprint : 7

Release Notes:

Delayed sprint due to issues with Azure credits. work resumed after resolution.

- Implement Azure -> OnCluster migration
- Implement OnCluster -> Azure migration
- Implement readiness checks for OnCluster deployments
- Implement conditional ingress creation
- misc. "comfort of life" changes, clearing up logging etc.

0.8.1.21 Implement Azure -> OnCluster migration

This piece of functionality implements the logic to allow users to change deployment types on a running operator custom resource deployment. Once the operator detects changes in the *deploymentType* field, it will kick off logic to create OnCluster resources, once all resources report as ready and available and the migration has been confirmed with the *confirmMigration* field, the operator will proceed to Azure teardown. After which the operator will solely dedicate reconciliation to OnCluster resources.

0.8.1.22 Implement OnCluster -> Azure migration

This new logic focuses on the migration of OnCluster databases to Azure. DBMMO will create an Azure deployment once a change in the *deploymentType* field is detected, proceeding to provision on Azure and waiting until the Azure deployment has reached full readiness, after which DBMMO will update appropriate status fields in the DBMMOMYSQL resource with the details of the Azure MySQL instance, such that other applications (or possibly other operators) can read the information contained within and from which they can invoke connections.

0.8.1.23 Implement Readiness Checks for OnCluster Deployments

This feature ensures that all OnCluster resources are declared as ready before the operator proceeds further with reconciliation logic. This feature is especially important to ensure that no resources of another type are deleted before the newly created resources are fully available. This logic was a prerequisite to ensure the safe migration of databases.

0.8.1.24 Implement Conditional Ingress Creation

Conditional Ingress creation was a feature that was implemented to mitigate issues within the Minikube cluster (as outlined in the issues section, the Minikube cluster had issues pulling in external images). Ingress is an externally facing endpoint that resolves to a running pod within a Kubernetes cluster, however, they are not necessary for the application itself to run. Other applications or resources within the cluster can use a service instead, which is essentially an internally facing endpoint that can be resolved by name rather than by IP. Kubernetes already handles service name resolution, therefore no further logic was necessary within DBMMO. With this feature, an ingress will only be created if specified so in the custom resource