POLITECNICO DI TORINO

*Master's Degree in Computer engineering*

Master's Degree Thesis

# Service-agnostic Dashboard for Kubernetes

**Supervisors:**
Prof. Fulvio Risso
Dott. Alex Palesandro

**Candidate:**
Alessandro Napoletano

Academic year 2019-2020

# Contents

# List of Figures

# Listings

# Introduction

## 1.1 The critical need of data accessibility

Since the introduction of information technologies in the past century, they have become year after year a fundamental part of our society's everyday life. A phenomenon so influential and rapid in its rise that can be considered without a doubt one of the most characterizing of human history. Huge loads of information are spread throughout the world thanks to these technologies in a way that still nowadays is hard to believe. With every new piece of technology, new information is generated, processed and most of the time that produces new information that are poured in a sea of data that is nonetheless difficult to understand for whomever will have to distill it into actual intelligence. As a matter of fact, information alone has no value if it is not used in the right way. A business manager need to analyze data in order to make decisions, and that is possible only if the data is easily accessible, readable and, most important of all, understandable. A very large amount of widespread information is little to no help when it comes to retrieve the essential out of it. The higher is the amount of information, the higher is the risk of drowning in sea of useless data.

The **information industry** has grown more and more important over the years and is now considered one the most relevant economic sectors. As the name states, it is a class of industries that relies on information, consumed or generated. To prevent being washed away with data, the information industry has put the research of managing data in a perceivable, fully explanatory yet concise way that can get the most out of an unorganized set of data volumes, as one of their most important needs. The evolution of data visualization has since come a long way, and it is now a rare occasion that a new piece of technology gets released to the public without a proper way to visualize, manage and analyze it. As it happens, business models for every kind of product (primarily software or technological pieces) are focused mostly on how their product can be used, often leaving aside all the details on how it really works, or how a feature is implemented. The world is bombarded with new products by the minute, and if something is not immediately and easily understandable, its real value can be not taken into consideration by the most.

One tool that quickly emerged from this need of easier data accessibility and understanding is the information dashboard. A dashboard is not only the point of contact between the user and the program but it is fundamentally the most powerful way to communicate with the external world. Especially when it comes to generic user oriented programs, giving a tool to make the interaction between the user and the program as understanding and intuitive as possible is as important, if not more, as the actual features claimed by the program. There are countless examples of programs with rather interesting ideas at their base that did not make it to the public because of their complexity and their lack of user oriented tool integration such a dashboard or a generic user interface. A well designed UI can really make the difference between a successful product and one that is just an interesting concept. If we look at the last decades of the past century, it is easy to understand why companies like Microsoft or Apple have emerged above all, and is all thanks to well thought designs that made from that time on complex tasks like using a computer as easy as we are now used to. Data and information are now accessible within a few clicks. This is exactly the core concept of a dashboard.

Designing a dashboard is not as easy as it gets and the keyword that need to be kept in mind is *communication*. To serve their purpose, a dashboard have to display a very wide range of data structures, a dense array of information all accessible in no more than few clicks, in a manner that communicates clearly and immediately. This requires an understanding not only of **what** is needed to be displayed, but also **how** data have to be displayed to enhance its meaning. The latter falls in an informational field that it is called *visual design*. Its importance is not to be undervalued when it comes to the development of a user interface as it is the first point of contact with the user. It should not to mention that fine-tuned visuals and a smooth user experience are key factor that makes the end-user choose a product over another.

If the information is important, it deserves to be communicated well.
*- Stephen Few, 2006*

## 1.2 General Concepts

Kubernetes is one of the most popular orchestration systems for managing containerized applications in a clustered environment, and is quickly becoming the standard choice for companies that need this kind of service. Because it is a vastly used and open-source platform, it's common for developers to define their own custom resources that extend the set of APIs that Kubernetes provides. All the best known dashboards for Kubernetes fall short when it comes to represent this kind of resources that are often left aside unacknowledged or displayed as no more than a list of names that brings no real value to the user experience. As a result, projects that relies heavily on the use of custom resources are forced to develop their own dashboards from scratch, or worse renounce to the idea of having a user interface. Because projects that use Kubernetes are countless and their number continues to grow by the day the risk is to have a number of different dashboards and interfaces that are not really necessary and will only cause confusion for their users.

This work address the problem described above, and proposes a different kind of approach in developing a generic Kubernetes web-based user interface that focuses primarily on offering the end user not only a way to manage their system (and the dashboard itself) in every aspect but also an environment that would grant easy implementations of custom components.

This thesis set its basis on three key factors:

- **Dynamic discovery of resources**: the dashboard have to be agnostic to the type of the single resource represented and does not need prior knowledge of the resources that reside in the Kubernetes cluster.
- **Modular environment**: the user can extend the dashboard functionalities, creating their own views using data from the Kubernetes server directly accessible in the dashboard with little effort.
- **Full customization of resources**: every view that represents a resource or a list of resources can be personalized to show only the information that is important to the user, avoiding useless display of data.

### 1.2.1 Dinamic discovery of resources

On a Kubernetes cluster, especially in a development environment, resources, and their definitions, are in constant changing, being updated, added or removed. Even resources that represents the core of the platform are updated when a new version of Kubernetes comes out. Because of that, the dashboard cannot rely on the implementation of a static representation for these resources. The approach presented is the following: every time the dashboard is accessed it starts the **discovery** process that is nothing more than a series of API calls to the Kubernetes API server that will respond with the resources available at the moment. That means that resources and views will always be up to the last update in the dashboard.

This solution well addresses the limitation that other dashboards have towards the exploration of custom resources. In fact, the dashboard proposed in this thesis does not make distinction between resources that are provided by default with Kubernetes and custom ones, and so the problem does not rise.

### 1.2.2 Modular environment

The dashboard comes with a set of default views that are fundamentals to the basic management of the Kubernetes system and all its resources. But that is often not enough to give the user a complete understanding of what is happening in their cluster, as many resources are related to each other and it would be quite useful to have all of them displayed in a single working space instead of multiple ones.

This project extends this idea to the concept of modular environment. That means that the dashboard offers its users the possibility to define **custom views** that are treated as empty work spaces that can host different kind of resources (single ones or lists) or custom components created by the user. This comes pretty handful in the context of custom resources that are part of complex systems developed over Kubernetes.

### 1.2.3 Full customization of resources

A completely generic and dynamic approach comes with its downsides. The knowledge of what to expect from a resource makes it easier to show what is really important to display and what not, creating ad hoc views for specific kind of resources. On the other hand, a generic code-base can only rely on what is common for all resources.

The solution that this project presents is to use a **design configuration** completely independent from the dashboard (it is, in fact, a separate YAML object) that it is used to fully customize the views of resources, granting the user the power to only display what they want and how they want. With the definition of a design configuration, not only the dashboard can have all the expressiveness that other more specific dashboards have (with less code and less time spent), but is also a step further in that regard, giving the user full control over its dashboard instead of imposing a view that often does not suite all users' needs.

### 1.2.4 Real time event responsiveness

One of the key features of the dashboard is its **real-time responsiveness**, meaning that if a resource or component gets updated (or added/deleted) in the cluster, the dashboard will react immediately and will automatically be updated without the need to refresh the page. This is obtained thanks to the use of **watches**, a powerful Kubernetes mechanism that allow the dashboard to get update notifications and therefore synchronize the state of resources. In detail, whenever there is an interest in viewing the real-time state of a resource and getting notified about its change, a watch is opened and will remain active until the view is left.

### 1.2.5 Integration with *Liqo* and *CrownLabs*

To serve its purpose and fulfill its potential, the dashboard needs a real implementation: some projects to use as test cases that could show the validity of this approach. Liqo and CrownLabs are the perfect choice: two project developed throughout the year at Politecnico of Turin that are built upon Kubernetes and relies heavily on the use of custom resources.

The idea was to use the dashboard as a framework and exploit its generic and modular approach to make custom views specific for Liqo and CrownLabs. In particular, some of the views developed that way are:

- **Liqo home view**: let the user have a general overview of its cluster, and how Liqo is working, making it easy to exploit the potential and functionalities that Liqo offers.
- **Liqo resources view**: making use of the custom view creation functionality that comes with the dashboard, this view shows all the necessary information to manage and keep track of the Liqo system and its resources, having all of them displayed in a single and compact page.
- **CrownLabs home view**: a user interface for CrownLabs, that let the user manage virtual machines on their Kubernetes cluster.

This was all possible with a relatively little amount of effort in terms of time spent and code written (roughly an hour of work per view), this because all the basic infrastructure was already made available by the dashboard (the connection with the Kubernetes cluster, a complete and secure login system, built-in real-time responsiveness). The integration with these two projects was reduced to its essentials and without needing to develop a dashboard from scratch.

# Kubernetes Integration

This chapter is meant to explain the platform that the dashboard presented in this work is designed for. Knowing what is Kubernetes and its general mechanisms will help to better understand the role of the dashboard and how Liqo*Dash* works, as well as acknowledging the needs and problems that the approaches of this work try to address and their solutions that make Liqo*Dash* significantly different from the other dashboards.

We will go through a more generic overview of the Kubernetes system, then going more in detail explaining why a dashboard is important for a system of that kind. Finally, a rapid view of the well-known dashboards that already exist, pointing the finger at their pro and cons, and how they all have some common flaws.

## 2.1 Kubernetes Overview

Kubernets is an open-source platform for deploying and managing containerized resources developed by Google and presented for the first time in 2014. Written in the Go language and supported by a great number of contributors and partnerships, its popularity has grown more and more, and it has quickly become the standard for container orchestration. Nowadays almost every big company that has to deal with the Cloud has adopted Kubernetes.

As a result, managing applications and services deployed over Kubernetes and their life cycle has become way less tedious and the fact that the platform itself provides consistency, scalability and availability with little to no effort is the reason why so many projects are relying on Kubernetes as a development platform. At this time, projects that are built over Kubernetes are countless and the number continue to grow by the day, which is only natural if we think about how powerful this platform is. Being able to deploy applications seamlessly is a real catch for developers that want to focus their resources (in terms of time, but mostly in terms of money to spend) in the development of their applications instead of the management of their deployments.

In the sections below we will briefly explain some of the resources and mechanisms that are of interest in relation to the work portrayed by this project. For more details it is recommended to check out the official documentation.

### 2.1.1 Pod

Pods are the smallest computing units that can be created in Kubernetes. A Pod is a collection of containers that share storage and network resources, as that they are running in the same execution environment. With the term *container* it is meant a unit of software that contains the application code and all of its dependencies. The most popular container runtime is Docker, although it is not a limitation of Kubernetes, that it supports also others.

### 2.1.2 Deployment

As the official documentation states, a Deployment provides declarative updates for Pods. In short, a Deployment is the manager of a deployed application, and as that, its Pods. Deployments make it easy to update versions of the code of applications running in Pods, in a reliable and safe way. As we will see in the next chapters, Liqo*Dash* itself is a Deployment, and changing through different configurations is particularly easy thanks to this object provided by Kubernetes.

### 2.1.3 Service

A Service is an easy way that Kubernetes provides to expose a set of Pods on a network, being it a closed local network or Internet. This abstraction comes in handy especially if we think about how Pods are created and destroyed regularly and their IPs change constantly. If a frontend (such as Liqo*Dash*) needs to be attached to a backend Pod, the problem is quite obvious: how to keep track of the IPs? The solution is to use a Service. The frontend will communicate with the Service (which does not change as Pods do) and the Service will handle the communication with the Pods. Different kind of Services will be described in **Chapter 4**.

### 2.1.4 API server

In developing a dashboard one of the most important parts is to access objects and data from the server, that are nothing more than API calls to said server. On Kubernetes, resources are made available thanks to the API server, a component that exposes the API of Kubernetes, intercepts API calls, validate them and, if there are no validation errors, processes them and respond with a the resources requested. The Kubernetes API server is the point of contact between the dashboard and the Kubernetes system.

## 2.2 Importance of a dashboard for Kubernetes

In the introduction we have discussed about the reason behind the concept of implementing a dashboard for a project and the added value it could bring shipping a user interface along with the main application. Kubernetes makes developers lives easier, but it is a really complex system and managing the platform itself requires a non trivial understanding of it. Most of the time, Kubernetes users interact with it through the CLI, a text-based interface that has the advantage of being immediate in its workflow (a single line command can let the user explore every resource they need in no time), but it is in fact hard to use for a non-expert, or a manager that is interested only in knowing how their cluster is performing, the status of their resources and not being drowned in the sea of details that the CLI provides, which are useful, but not all the times.

Dashboards are essentially tools that come in support of users that are not so acquainted with the use of a CLI or prefer a more graphical environment than just some line of text. But they can be so much more than just a visually pleasant way to see what can already be seen in the command prompt. Ideally, the user interface's job is to also make common operation that would require several passages on the command line, readily available with a few clicks. In the case of Kubernetes, performing CRUD operations on resources is a good example of how a UI can be put of good use. Also, in terms of customer support, the person in need of help is often requested to open the command prompt, running some commands and *describe what they see* which is not the best way to deal with problems. For these cases and more, having a dashboard to manage Kubernetes is a not only a general good recommendation, but it is fundamental when there is the need to explain what is happening in the platform in way that even user not accustomed with the system can interpret and understand.

## 2.3 Kubernetes dashboards: State of the Art

In this section we are going to explore in brief what is the state of the art of current open-source web-based Kubernetes dashboards, explaining for each the strong points and the weak ones.

### 2.3.1 Kubernetes Dashboard

Kubernetes Dashboard is a general purpose, web-based UI for Kubernetes clusters. It allows users to manage applications running in the cluster and troubleshoot them, as well as manage the cluster itself. It is the dashboard provided and maintained by the Kubernetes team, which has recently released the version 2.0. It is by far the most popular and most used dashboard thanks to its vicinity with the core project.

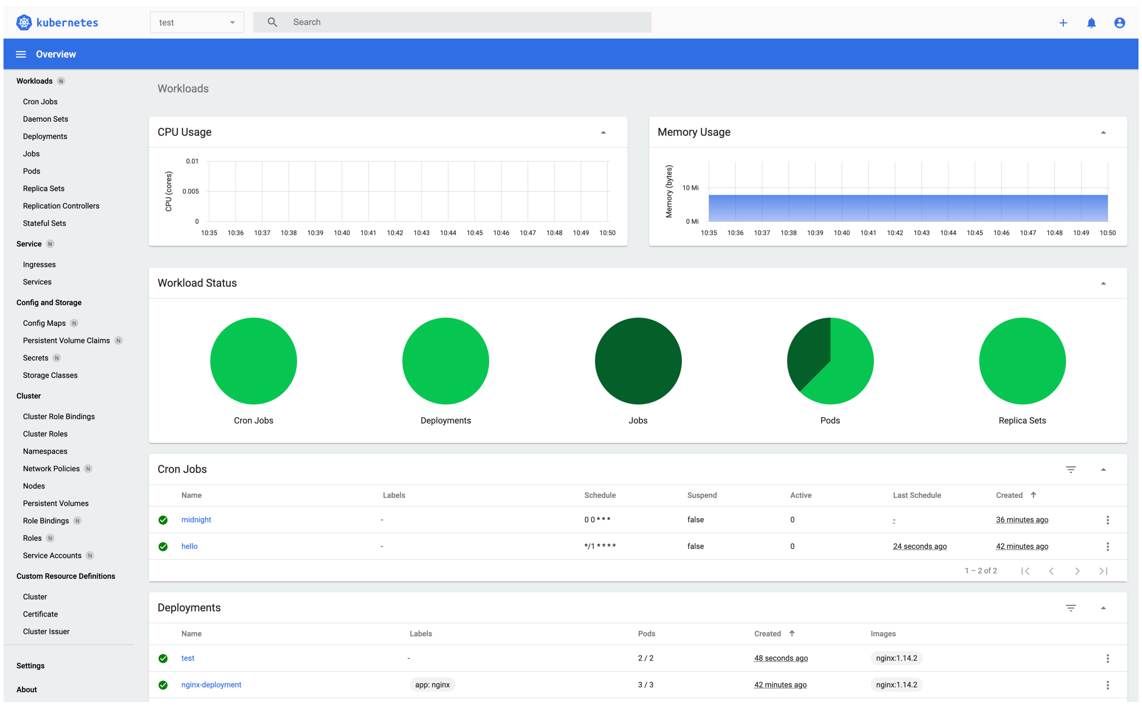
Strong points:

- Easy to install as a deployment directly in Kubernetes;

- Useful to see basic information of resources and check the general status of the cluster;

Weaknesses:

- No easy way to perform operation on resources (only deployments can be created through a form);
- No useful representation for custom resources;
- No easy way to filter by status, e.g. to see all "Pending" pods;
- No possibility to customize the dashboard;
- Only support for a single cluster;



Cron Jobs

| Name | Labels | Schedule | Suspend | Active | Last Schedule | Created |
|---|---|---|---|---|---|---|
| midnight | - | 0 0 * * * | false | 0 | - | 36 minutes ago |
| hello | - | */1 * * * * | false | 0 | 24 seconds ago | 42 minutes ago |

1 – 2 of 2

Deployments

| Name | Labels | Pods | Created | Images |
|---|---|---|---|---|
| test | - | 2 / 2 | 48 seconds ago | nginx:1.14.2 |
| nginx-deployment | app: nginx | 3 / 3 | 42 minutes ago | nginx:1.14.2 |

**Figure 2.1:** Kubernetes Dashboard.

### 2.3.2 Octant

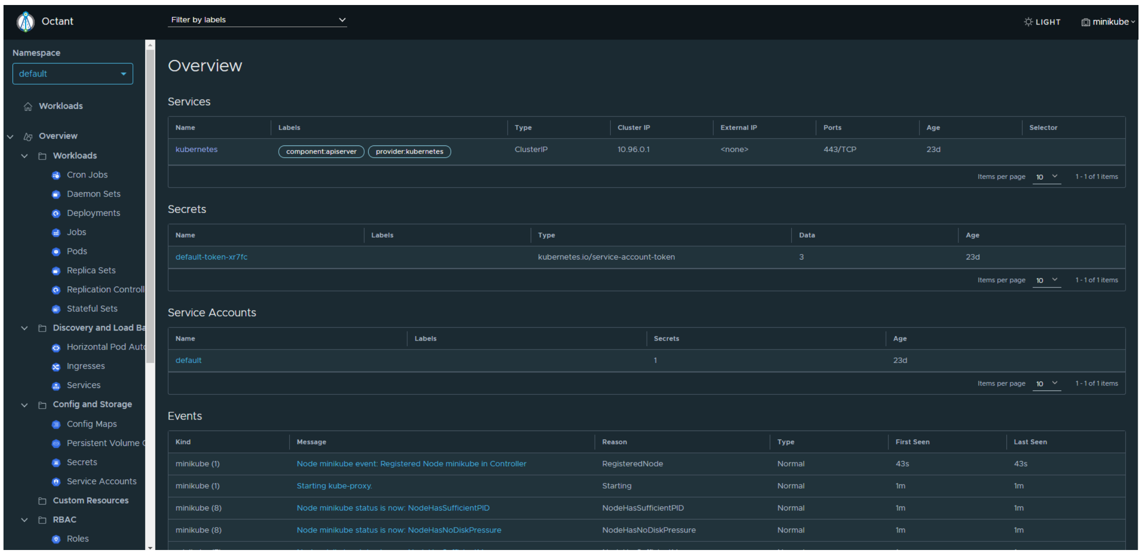
Octant is an open source developer-centric web interface for Kubernetes that lets the user inspect a Kubernetes cluster and its applications. Developed by VMWare, it has quickly become one of the favourites by the users thanks to its appealing user interface.

Strong points:

- Useful to see basic information of resources and check the general status of the cluster;
- Possible extensibility through custom Go plug-ins;

Weaknesses:

- Not as stable as the Kubernetes Dashboard;
- No useful representation for custom resources;
- Only support for a single cluster;



Overview

Services

| Name | Labels | Type | Cluster IP | External IP | Ports | Age | Selector |
|---|---|---|---|---|---|---|---|
| kubernetes | component:apiserver provider:kubernetes | ClusterIP | 10.96.0.1 | <none> | 443/TCP | 23d | |

Items per page 10 1 - 1 of 1 items

Secrets

| Name | Labels | Type | Data | Age |
|---|---|---|---|---|
| default-token-xr7fc | | kubernetes.io/service-account-token | 3 | 23d |

Items per page 10 1 - 1 of 1 items

Service Accounts

| Name | Labels | Secrets | Age |
|---|---|---|---|
| default | | 1 | 23d |

Items per page 10 1 - 1 of 1 items

Events

| Kind | Message | Reason | Type | First Seen | Last Seen |
|---|---|---|---|---|---|
| minikube (1) | Node minikube event: Registered Node minikube in Controller | RegisteredNode | Normal | 43s | 43s |
| minikube (1) | Starting kube-proxy. | Starting | Normal | 1m | 1m |
| minikube (8) | Node minikube status is now: NodeHasSufficientPID | NodeHasSufficientPID | Normal | 1m | 1m |
| minikube (8) | Node minikube status is now: NodeHasNoDiskPressure | NodeHasNoDiskPressure | Normal | 1m | 1m |

**Figure 2.2:** Octant.

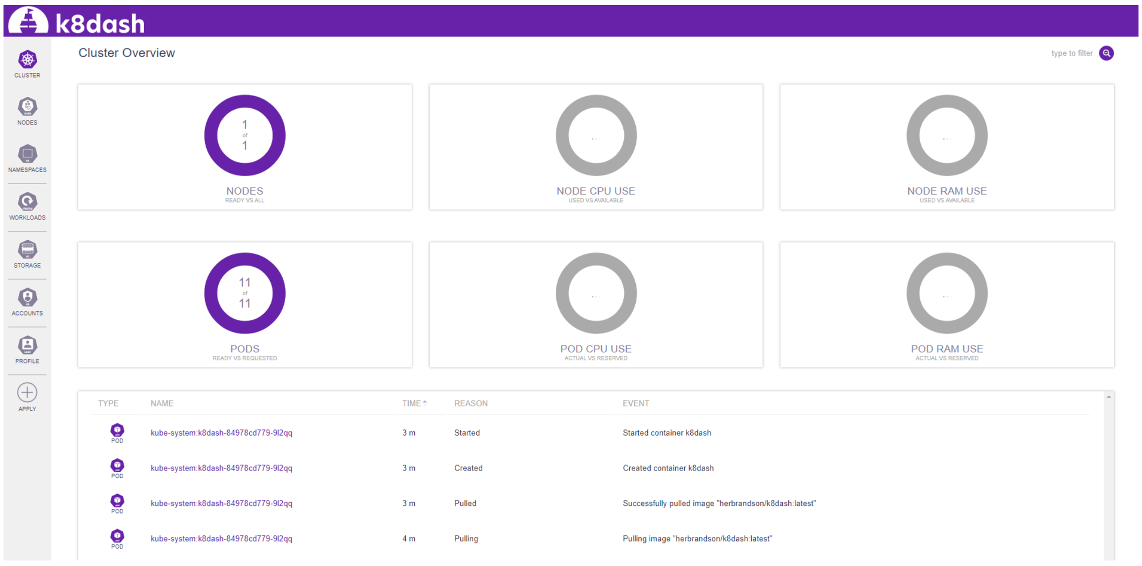
### 2.3.3 k8dash

k8dash is another popular web-based UI, that claims to be the easiest way to manage a Kubernetes cluster. Strong points:

- Immediate and compact general status visualization to keep track of the performance of the cluster;
- Really fast UI;
- Integration with OpenID to login;

Weaknesses:

- No support for Custom Resource Definitions;
- No possibility to customize the dashboard;
- Only support for a single cluster;



| TYPE | NAME | TIME | REASON | EVENT |
|---|---|---|---|---|
| POD | kube-system:k8dash-84978cd779-9l2qq | 3 m | Started | Started container k8dash |
| POD | kube-system:k8dash-84978cd779-9l2qq | 3 m | Created | Created container k8dash |
| POD | kube-system:k8dash-84978cd779-9l2qq | 3 m | Pulled | Successfully pulled image "herbrandson/k8dash:latest" |
| POD | kube-system:k8dash-84978cd779-9l2qq | 4 m | Pulling | Pulling image "herbrandson/k8dash:latest" |

**Figure 2.3:** k8dash.

## 2.4 Custom resources: the missing point

As it can be seen by analyzing what are the weaknesses that all the dashboards presented above have in common, it is clear that they all fall short when it comes to giving the user the possibility to easily customize their dashboard. Another interesting flaw that is present in all of the above solutions is the lack of a proper representation of Custom Resource Definitions (or CRDs) and their derived custom resources.

But what are custom resources in Kubernetes? Custom resources are **extensions** of the Kubernetes API. Kubernetes works through endpoints, meaning that an object that resides in the server is directly accessible (if we have the right to access it) simply performing an API call to the API server. Creating a custom resource means that we are creating a new endpoint in the Kubernetes API that is pointing at a collection of a specific kind of API objects. To add a kind of custom resources, it is firstly necessary to define it through an API that Kubernetes provides by default, which is the CustomResourceDefinition. CRDs are simple and can be created easily without having to program anything. There is also another way to add custom resources to the cluster, that is programming an API Aggregation, which allows more control over the APIs but requires more effort.

Many developers that relies on Kubernetes for their projects need to define their own set of declarative API and working with custom resources is nowadays a standard. Kubernetes is an open platform and the possibility for customization is as important as its core functions. As a result, it is not surprising to see that in a production environment such the one of a medium or big company, the use of custom resources is heavily implemented on their Kubernetes installations, and many core functions are built using custom resources. It all just makes Kubernetes more modular.

So why if the platform embraces this concept of modularity to the point that is as important as the rest of the functionalities, a dashboard, that has as a key point the job to show and explain what reside in the cluster, does not? Why using a dashboard I can easily scale the replicas of a deployment with a single click, but when it comes to a custom resource all the UI has to offer is a YAML representation of the object? Even when the object can be modified (and only in certain dashboards it is possible) as a matter of fact, it brings no real value to the user experience. Simply moving what can be done in a text editor or in the command prompt, in a same-fashioned text editor, but within a web interface, is at least useful (if you are already using the dashboard) but it does not make the user want to boot up the dashboard just to edit a line of code.

A proper way to represent custom resources, interact with them and customize the views to the likes of the end-user is the missing point that all of the above dashboards have in common, and as such it is what Liqo*Dash* sets its key lines of work, facing the problem with a completely different approach that will be explained in details in the following chapters.

# Liqo*Dash* Overview

In this chapter we are going to explain how Liqo*Dash* has been built, the choices made and why building a reliable and scalable UI for Kubernetes can be really difficult. Speaking about the latter, it is no secret that the Kubernetes APIs are not really suited to build user interfaces on top of them, and, as their implementation suggests, are mainly been designed so that creating resources and having managers working on them consumes little computational resources. That at the cost of having a poor general understanding of what is really happening in the cluster as a consequence of an action. Ideally, what the user want is the overview of a workflow that is often separated across multiple objects. For example, when we scale up the replicas of a deployment it is not immediate that this has repercussions on the status other objects (such as Pods and ReplicaSets). It is a consequence that the user (or the dashboard that uses the Kubernetes APIs) needs to know beforehand. Even though Kubernetes offers some ways to trace the links between resources, this is often not enough for a complete overview of the system and certainly not satisfying in the context of custom resources. A tool that would show at runtime the correlations between resources that would make the exploration of the cluster easier would be quite helpful in these situations, but we will discuss about it later on in the next chapters.

## 3.1 The Frontend

In this section we are going to explore the solutions proposed to build the frontend of the dashboard. The term *frontend* refers to just the user interface part of the applications, the one that runs directly in the browser of the end-user and, ultimately, what the user is going to see and interact with. It is the single point of contact between the application and the person who is going to use it. In a developer's mind, it is as important as every other component, but when it comes to the users it is the user experience what really makes the difference in choosing between one solution over another.

### 3.1.1 JavaScript Frameworks

JavaScript is one of the most popular programming languages used mainly to develop dynamic web-based applications and web browsers. The decision about using JavaScript to code the dashboard was made on the basis of the following factors:

- Iterpreted language: instructions are executed directly in the browser, without previous compilation. That gives the application platform independence, makes the language more dynamic and makes the development of plug-ins and extensions easier.
- Weakly typed: there is no need to define the type of object that are going to be processed, which comes useful for example when there is no knowledge of the kind of object that the API server will respond with.
- NPM packages: being one of the mostly used languages out there, there is plenty of open-source packages and libraries made by the community that avoid writing useless boilerplate code.

There is no doubt that JavaScript is an objectively flawed programming language, and pages could be written about the pro and cons of all the alternatives, but for the reasons depicted above, the choice of using it to develop the dashboard was considered the most fitting.

### ReactJS

React (or ReactJS) is an open-source JavaScript framework that is used for building single-page user interfaces and UI components. Released in 2013 and maintained by Facebook, it has gained lots of popularity through the years and there is an ever-growing passionate community of individuals and companies that helps making it one of the best options when working with JavaScript frontends. As their front page states *React make it painless to create interactive UIs*, allowing the user to declare components that will automatically transpiled in HTML, without the need to have splitting views that separates HTML and JavaScript code. This approach is what is so interesting about React and becomes really helpful when it comes to debugging code. Declaring components, each one managing its own state, that can be composed to create complex user interfaces makes the process of building a UI a relatively easy and painless job.

### WebPack

Applications are often split in separate chunks of code that enclose different functionalities. Each chunk is called a module. WebPack is a *static module bundler* for JavaScript applications that builds a dependency graph which maps every module and exports them as bundles. The use of WebPack (or a module bundler of sorts) was necessary because of the many plugins and external modules that Liqo*Dash* uses (for example the various loaders for images, SVGs, fonts and others) as well as to include the JavaScript library for accessing Kubernetes API which we will talk about more in detail later in this chapter.

### Ant Design

Ant Design is a React UI library that contains a set of high quality, fully responsive and highly customizable components for building rich, interactive UIs. It was chosen as design framework mostly because its simple but fresh design that is particularly well suited for a dashboard. It also proposes a very interesting set of features that will be listed below:

- Enterprise-class UI designed for web applications;

- A set of high-quality React components out of the box;
- Internationalization support for dozens of languages;
- Powerful theme customization in every detail;
- Predictable static types;

Sadly it is a relatively new framework, although it has seen a spike of popularity in recent days, so API are in constant change, and not always explained clearly. Also it is a Chinese project, and sometimes their English translation is not the best.

### 3.1.2 Javascript library for Kubernetes

The library used to connect to the Kubernetes API is kubernetes-client-javascript, our patched version of the official one. As the Kubernetes developer team states, the client of official library is implemented only for server-side, using NodeJS. This is because of various dependencies that can only be solved server-side, that would not allow the use of these API to a pure JS client. The solution was to use a patched version is the same as before, but with a slightly different build file. In fact in this project it is performed a binding of those server-side-only modules with user-accessible ones. With this library we are able to make calls to the Kubernetes API inside the dashboard, without the need of a backend.

### 3.1.3 Watches

As explained in the introduction, one of the key features of Liqo*Dash* is its **real-time responsiveness**. Resources changes constantly in a Kubernetes cluster, and there is no way to predict them. Setting up a timer that updates every resources once it reaches zero is a bad idea, because most of the time only few resources from all the ones we are keeping track of are actually updated, and it is generally a not scalable solutions. The input that something has changed need to come from the server itself, and when notification of sorts need to be intercepted by the dashboard and processed accordingly changing the information displayed. To achieve it Liqo*Dash* leverages the use of **watches**, a Kubernetes mechanism that is able to efficiently detect changes of resources and sending a notification to who is listening for them.

The way they work is simple but, as we said, quite efficient: every Kubernetes object has, in its metadata, a field called *resourceVersion* that keeps track, as the name says, of the current version of the object. When an object is updated its *resourceVersion* is updated too. When the frontend asks for a resource it will be responded by the Kubernetes API server with a resourceVersion that can be use to start a watch on said resource. If the watch is started, the server will then notify the caller of any changes that have been made to that resource after the supplied resourceVersion. From the frontend point of view, whenever we are interested in viewing the real-time state of a resource (e.g. we are on a view that shows a specific CRD and its related custom resources) and getting notified about its change, a watch is opened and will remain active until we leave the view. At its core, a watch is nothing more than a HTTP/2 connection established with the cluster's API server.

## 3.2 The Backend

In the sections below we are going to explain the solutions and how we have chosen to implement the backend part of the application. As we did for the *frontend*, let's give a more techical definition of what is it meant by the tems *bakend* before we begin the exploration. The term *backend* refers to all of the parts and components of the software that the user does not come in contact directly. It is what the user does not see and does not need to know or understand to work with the application. In contrast with the frontend which the code is often not compiled a priori, but its interpreted by the web browser and its content generated on-the-fly, the backend is often compiled code, static, and its purpose regards everything that happens before the page is displayed, such as processing of API calls and generating proper responses, handling complex workflows and storing records in a database.

All this work is done by the backend, but in the context of creating a dashboard for a platform like Kubernetes, is that really necessary? We will be discussing about that in the next paragraphs.

### 3.2.1 No need for a complex backend

We have explained that one of the roles of a backend is managing API calls to provide an object or resource to the frontend to display (or operate with it). When we talked about the Kubernetes internal in the previous chapter, we remember about a particular component called the **Kubernetes API server** and its job, among others, it to provide responses for API calls in the form of JSON objects (when we ask for resources). That said, there is no real reason for a backend implementation that takes frontend calls, processes them into other calls to the API server and then forwards the responses back to the dashboard. It seems like an unnecessary step that only makes the dashboard slower in retrieving data.

On the other hand, there are equally valid points for having a backend that implements business logic. They are well described in the official Kubernetes Dashboard documentation as the following:

- Clear separation between the presentation layer and business logic layer;
- Transactional actions are easier to implement on the backend than on the frontend;
- On certain occasions where to get the full view of a resource there is the need to perform more than one API call, the backend is faster on retrieving data, shortening the round trip time. For example, getting a list of pods with their CPU utilization timeline requires at least two requests;

Given the little advantages that a backend gives (we are talking about speed of data retrieving, and only in particular occasions), the implementation of a complex backend which would take time to develop and perform fairly well, we decided to go against this decision.

But relying on not having backend at all it is unthinkable and only leads to a series of problems (one in particular that we are going to show in detail later) that cannot be overcome. That is why Liqo*Dash* has its implementation of a backend, just a minimal one. As a result, we have avoided all the troubles about not incorporating a backend solutions, with all the advantages of having little to no delay time given by the middle step between the API call of the dashboard and the response of the Kubernetes API server.

### 3.2.2 NGINX Reverse Proxy

What does it mean **minimal backend** and how is it implemented? The backend that we opted for is just a minimal implementation of an NGINX reverse proxy. A reverse proxy is a type of proxy server that is responsible for retrieving resources from a server (the Kubernetes server in this case) on behalf of a client (the dashboard frontend). Using NGINX as reverse proxy was a natural choice because of its high-performances and lightweight structure. This proxy will receive API calls from the dashboard frontend and, without additional processing, will forward them to the Kubernetes API server. The same logic is applied for the server responses.

### 3.2.3 The CORS problem

As we said earlier, without even this minimal implementation of a backend, we will incur in major problems, security-wise. In detail, without a backend, and sending requests to the Kubernetes API directly from the dashboard, there could be some errors performing these requests. One kind of error regards the configuration of CORS. Cross-Origin Resource Sharing (CORS) is a standard that allows a server to relax the same-origin policy. This is used to explicitly allow some cross-origin requests while rejecting others. For example, if a site offers an embeddable service, it may be necessary to relax certain restrictions. CORS errors occur where either the browser would not accept a cross-origin request or the API server would not accept the request, responding with an error, if it does not allow cross-origin requests. The CORS mechanism is very important for the safety of both client and server, and to prevent error or avoiding configurations that would possibly put the Kubernetes server in danger, the solution was put a middleware between the dashboard and the actual API server: the NGINIX reverse proxy. With a proxy we have no CORS related errors, keeping the security that this mechanism provides.

### 3.2.4 Pod to API Server communication

In this small section we are going to take a look on how a Liqo*Dash* Pod is structured, what it contains and how the communication between it and the API server is performed through some key points.

- Init Container: this is a special kind of container that run before the app containers are started, at the start of the Pod's life cycle. Its role will be clearer after reading the next chapter. For now, what we need to know is that it shares a folder with the NGINX proxy where it retrieves the certificate needed to talk with the dashboard in HTTPS.

- Liqo*Dash*: the code of the application, in the container as a Docker image.
- NGINX Proxy: requests performed by the dashboard applications to a particular path that starts with **/apiserver/** are intercepted by the proxy and forwarded too the Kubernetes API.
- Kubernetes API: at this point, the API call is managed by the Kubernetes API validation mechanism, that verifies that what we are making is a valid request, with a valid token and the user has the privileges to access the resources it is asking for. Whatever the response is (could be a Success or a Failure) it will be forwarded the same way it arrived, and will be showed in the dashboard accordingly.

All the connections between proxy and dashboard/API server are secured (over HTTPS).

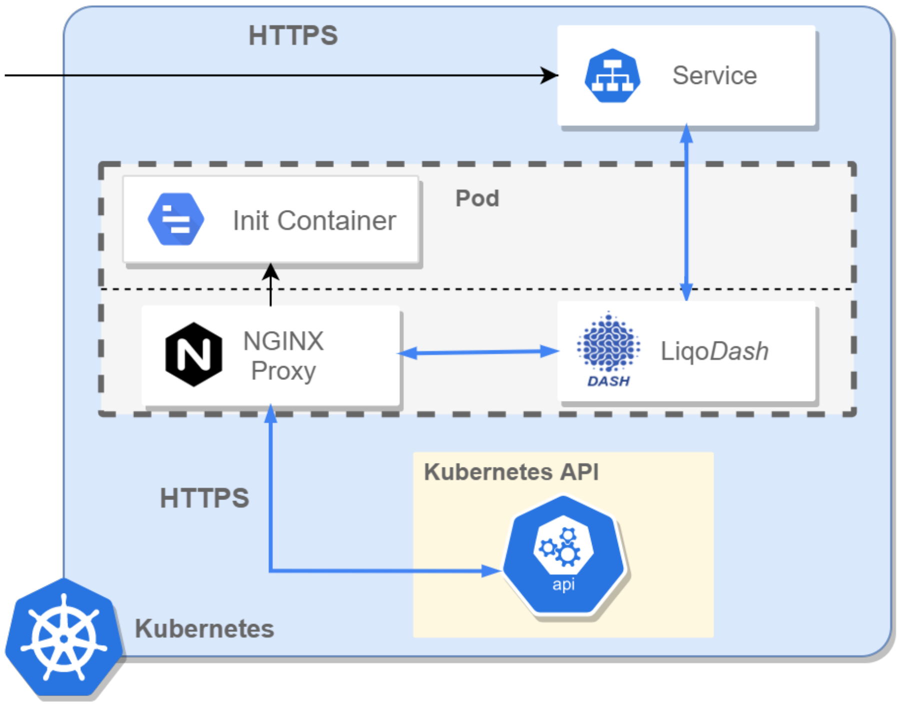


**Figure 3.1:** Liqo*Dash* Pod structure and communication with the API server.

# Authentication, Authorization and Methods of access

One of the first implementations to think about when starting developing a web-base user interface that can be potentially accessed by anyone on the internet is a secure and reliable method of login. The dashboard is no exception, as it gives its users access to resources in the cluster. This chapter is dedicated to explaining in details the solutions adopted in regard of managing authentication and authorization to a Kubernetes cluster, as well as the methods of accessing the dashboard.

## 4.1 Kubernetes Role-based Access Control

Before talking about the various methods of login, it is important to know that, whether it would be the method used, the dashboard relies on the same basic access control entity for authentication and authorization: the Kubernetes role-based access control (RBAC). Role-based access control is a method of regulating access to computer or network resources based on the roles of individual users within the organization. RBAC authorization uses the *rbac.authorization.k8s.io* API group to drive authorization decisions, allowing the user to dynamically configure policies through the Kubernetes API.

To understand more about how Liqo*Dash* leverages RBAC for authorization, we need to explain what are defined as roles and bindings in Kubernetes.

### Roles and ClusterRoles

An RBAC Role or ClusterRole contains rules that represent a set of permissions. Permissions are purely additive (there are no ”deny” rules). A **Role** always sets permissions within a particular namespace; when creating a Role, it is mandatory to specify the namespace it belongs in. **ClusterRole**, in contrast, is a non-namespaced resource. The resources have different names (Role and ClusterRole) because a Kubernetes object always has to be either namespaced or not namespaced and it can't be both. With ClusterRoles can be defined permissions that are relative to a namespace, granted across other individual or every other namespaces. Also there can be defined permissions for resources that are cluster-scoped.

### RoleBindings and ClusterRoleBindings

A role binding grants the permissions defined in a role to a user or set of users. It holds a list of subjects (users, groups, or service accounts), and a reference to the role being granted. A **RoleBinding** grants permissions within a specific namespace whereas a **ClusterRoleBinding** grants that access cluster-wide. A RoleBinding may reference any Role in the same namespace. Alternatively, a RoleBinding can reference a ClusterRole and bind that ClusterRole to the namespace of the RoleBinding.

### ServiceAccounts and Token Controller

In Kubernetes, **service accounts** are used to provide an identity for pods. Pods that want to interact with the API server will authenticate with a particular service account. By default, applications will authenticate as the default service account in the namespace they are running in.

The **Token Controller** component is part of the Kubernetes controller-manager. Its job is to observe the creation or deletion of ServiceAccounts and act consequentially, creating **Secrets** that contain tokens used to authenticate the user.

## 4.2 Liqo*Dash* Login

During the installation of the dashboard a service account and a role binding are created along the deployment of the application. The role binding binds the newly created service account with a cluster role that has admin privileges. That means that a secret containing a token will be created by the Token Controller, and this token can be used to login in the dashboard and access the Kubernetes cluster authenticated and authorized as an admin. Being an admin in the context of Kubernetes management means that every resource in every namespace is accessible and is a type of privileges that only few people should be having, especially when there are lots of cluster's users.

With that said, although the installation of Liqo*Dash* provides a cluster admin account for authentication, it is certainly not mandatory nor suggested to use it, as the dashboard is an application that is intended to be used by everyone, not only the administrators. Users with limited privileges will still be able to access the dashboard, and interact with just the resources they have the authorization for. As a matter of fact, all the dashboard do is use the token (provided by the user via ways that we will explain in the following sections) to make calls to the Kubernetes API. The process of validating authorization and authentication is all handled by the Kubernetes system. If a users do not have permission to access a resource, Kubernetes will respond with a *failure* and the dashboard will report it graphically to the user.

After the first login, the token is stored in a **Cookie**. That prevents successive interactions with the dashboard to ask again for the token. The Cookie will be deleted when the browser is closed.

Liqo*Dash* offers two distinct methods of login that are provided in different ways, but share the core aspects of validation that we have explained before. Let's explore them.

### 4.2.1 Service Account Token

This first option, which is also the easiest one, is to make use of a service account token. As we have said, every service account, being that the one created at the installation of the dashboard or one that was already existing in the cluster, is related to a secret that contains a token. This token can be used to login into the dashboard and access resources in the cluster.

```
Name:         liqodash-admin-sa-token-94v8x
Namespace:    liqo
Labels:       <none>
Annotations:  kubernetes.io/service-account.name: liqodash-admin-sa
              kubernetes.io/service-account.uid: ad421b68

Type:  kubernetes.io/service-account-token

Data
====
ca.crt:     1025 bytes
namespace:  4 bytes
token:      XXXXXXXXXXX
```

**Listing 4.1:** An example of Secret object that contains a token.

### 4.2.2 Support for OIDC

The dashboard also offer another solution of login, that is through the integration of OpenID. OpenID Connect is a flavor of Oauth2 supported by all the most famous providers, such as Azure, Google, Facebook and Keycloak. The protocol's main extension of Oauth2 is an additional field returned with the access token that is called an **ID Token**. This token is a JSON Web Token (or JWT) with well known fields, such as a user's email, signed by the server.

The login process follows some steps that we try to summarize in a few key points:

- Once the dashboard is accessed, if there is no Cookie containing the token or if the token has expired, the user will be redirected to the identity provider.
- Once the access with the identity provider is done, it will respond with an *access token*, an *id token* and a *refresh token*.
- The *id token* is sent in the Authorization header to the Kubernetes API server.
- The API server will validate the JWT by checking against the certificate name in the configuration, making sure that the JWT has not expired and that the user has valid authorizations to access the dashboard.
- The API server then respond to the dashboard.
- The dashboard provides feedback to the user.
- In case of positive response, from now on the dashboard will send every API calls with the token id in the Authorization header.

#### Liqo*Dash* Configmap

The dashboard needs to know if the user want to access with an OIDC provider or through the classic login **before** making every other call to the API server. For that it uses some static environment variables that are provided by the user before the Liqo*Dash* Deployment has started. Changing these variables, and so the method of login, will require the Deployment to be restarted. The variables are stored in a Configmap, that is a Kubernetes object that is perfect for providing simple value to key storage. The Configmap (showed in the following piece of code) has four variables:

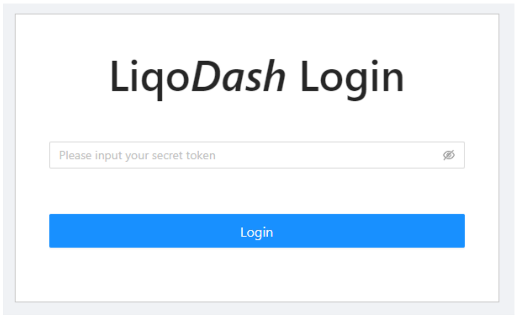
- Client ID: ID that the identity provider has registered the user with;
- Client secret: secret that the identity provider will accept;

- Provider URL: the identity provider that will be contacted (e.g Google, Keycloak);
- Redirect URL: the URL the user will be redirected to after the login is complete. This could be *localhost* in a development environment, or the name with which the user is accessing the dashboard with;

```yaml
apiVersion: v1
...
  name: liqo-dashboard-configmap
  namespace: liqo
  resourceVersion: "012345"
  selfLink: /api/v1/namespaces/liqo/configmaps/liqo-dashboard-
    configmap
  uid: 8e48f478-993d-11e7-87e0-901b0e532516
data:
  oidc_client_id: <CLIENT_ID>
  oidc_client_secret: <CLIENT_SECRET>
  oidc_provider_url: <PROVIDER_URL>
  oidc_redirect_uri: https://example.com
kind: ConfigMap
```

**Listing 4.2:** OIDC Configmap of Liqo*Dash*.

**(a)** Liqo*Dash* login page



**(b)** An example of OIDC login using keycloak

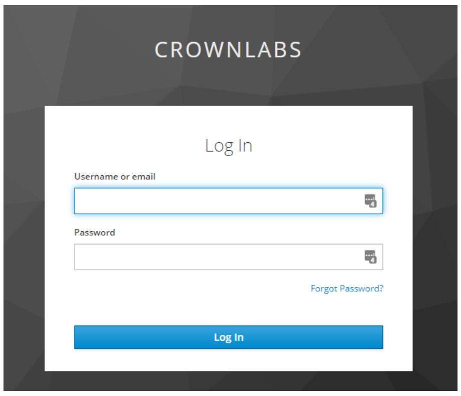


**Figure 4.1:** The two methods of login.

## 4.3 Accessing Liqo*Dash*

There are two major ways of exposing the dashboard so that it can be accessed by the users. In this section we are going to explore them in detail, understanding what are the differences between the basic and advanced configuration.

### 4.3.1 Basic access: Node Port and Port forward

When we explained the fundamentals of Kubernetes, we talked about Services. To recap, a Service is an abstract way to expose an application that is running on a Pod as a network service. There are various types of services, and we are going to concentrate on one in particular: the **NodePort**.

When setting the type of a service to NodePort, the Kubernetes control plane allocates a random port (specified from a range typically from 30000 to 32767) and each node proxies that port into the Service. With that, we are no longer dependant to the dashboard Pod actual internal IP, because the application will be accessible through the master Node IP (that supposedly does not change) via a static port, and so even if the Deployment is restarted, the dashboard will always be accessible with the same path.

Accessing the dashboard via NodePort requires, as we said, the IP of the master node. But there is another way to access the application, which is fundamentally the same concept, but allows the user to work with *localhost*, that is the machine's loopback address. To do that, we take advantage of the **port-forward** functionality that is provided with the Kubernetes CLI that exposes a service to a local port.

### 4.3.2 Advanced access: Ingress

Liqo*Dash* can also be exposed using an Ingress. An Ingress is an API object provided by default by Kubernetes that manages external access to the services in a cluster, typically HTTP. As the official Kubernetes documentation specifies, Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource. For an Ingress resource to work, the cluster must have an Ingress controller running. All the major Cloud Providers such as Amazon AWS and Azure also offers support for Ingress controllers.

The dashboard service can so be externally reachable through an URL given by an Ingress configuration and the installation provides the user with a pre-compiled Ingress resource whenever a hostname is specified. Of course, that resource can be modified at will according to the user preferences.

### 4.3.3 Differences between access methods

Liqo*Dash* offers the option of more than just a single method of accessing the dashboard, but with that comes the question about what is the difference between them and why use a method over another. In this section we will explain what are the possible cases that will help a user make that decision.

### When to use Port-Forward

Using kubectl port-forward to access the dashboard service is usually meant for testing/debugging purposes. It is not a long term solution, as it require to always run the port-forward and keep it active. However, if the user just wants to access the dashboard via *localhost*, this is the easiest method.

### When to use NodePort

Using a NodePort means that a specific port is opened on all the Nodes, and traffic sent to this port is forwarded to a service (in this case the dashboard). It is not recommended to use this method to directly expose a service on the internet, but if the user is going to use the dashboard on the same machine where their cluster is, that is a simple and valid solution.

### When to use Ingress

Using Ingress is probably the best way to expose Liqo*Dash*, especially if the idea is to access it through the internet. It can provide load balancing, SSL termination and name-based hosting, letting the users access the dashboard using a host name instead of just its IP. Because there many types of Ingress controllers, each one with different capabilities, it may require some work to properly set up.

## 4.4 Liqo*Dash* and Kubernetes communication

Now that we know how the *authorization* and *authentication* works, let us take a look at how the *connection* is established between the client (the dashboard) and the server (the Kuberntes cluster) following the workflow of an API.

- Client side: through the Kubernetes JavaScript library the dashboard makes a call to the Kubernetes API (e.g. we want to list all CRDs present in the

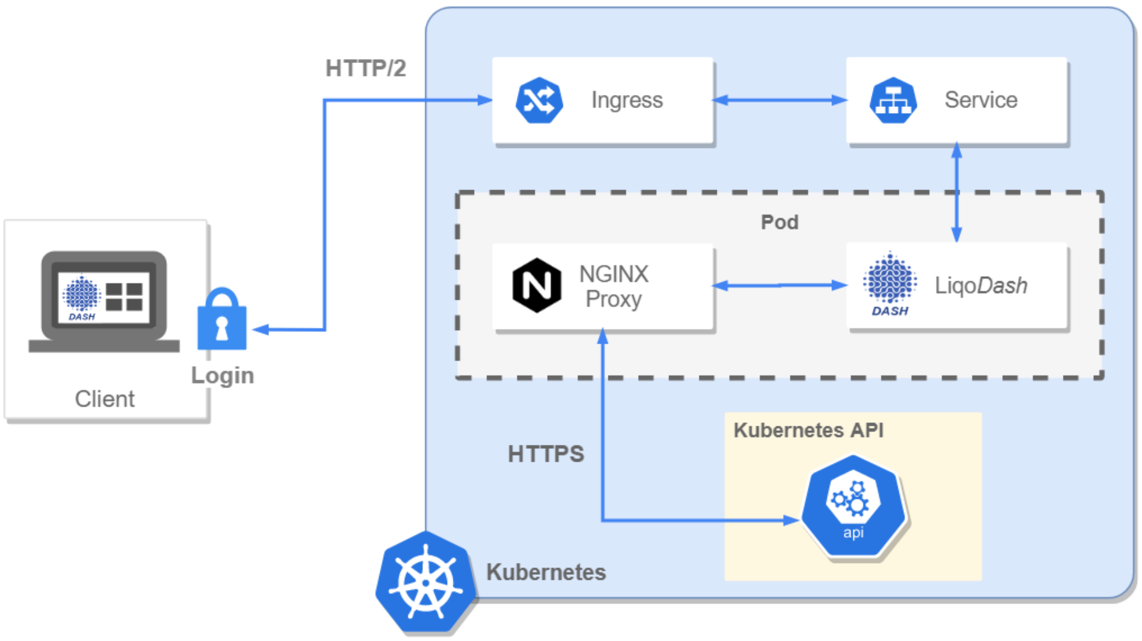


**Figure 4.2:** Liqo*Dash* API calls workflow.

cluster). As we explained in the previous chapter, this is not a direct call, meaning that we won't ask directly the Kubernetes API server, but it contains nonetheless all we need to make an API call: the bearer token (that we need to prove to the Kubernetes API that we are a valid user that has access to the resource requested), the path to the endpoint we want to access and, of course, the body and the option headers we need if we are creating or updating resources. Because we are transferring sensible data (first of all the token), the connection needs to be secure and its only natural to use HTTPS, and in particular HTTP/2, because of the multiple connections opened with the use of **watches**.

- Ingress (optional): although highly suggested the use of an Ingress in not a requirement for the dashboard to work.

- Liqo*Dash* service: the request is then forwarded to the Liqo*Dash* service. If there is no Ingress the dashboard will be accessed directly through the service, exposing it via NodePort or port-forward. The service is the point of contact between the user and the pod (the resource that actually contains the dashboard application).

- NGINX reverse proxy and API server: the request is then forwarded to the proxy, processed and passed on to the server that will produce a response. This workflow is the same as what is explained in the Chapter 3.

# Resource Discovery and Operations

So far we have discussed about the general concepts behind the Liqo*Dash* project, and why we thought it was necessary to improve the dashboards that already exist for Kubernetes, opting for a completely different approach at its core. We have set the basis to the system we are working on and described the workflow of data exchange with the server. In the next few chapters we are going to take a look at the internals of the dashboard, the ideas that led to the birth of this application and the solutions proposed, as well as the downsides that came with their implementation.

In this chapter we will explain the **resource discovery process** that uses an agnostic-like method of resource retrieval and exploration, and the various **common operations** that can be performed on the discovered resources.

## 5.1 The generic approach

When it comes to think about the design of an application, whatever its kind or purpose may be, it is often a good choice to consider not just the immediate functionalities that the software need to implement, and so creating ad hoc programs that serve only that purpose, but being open to the possibility of supporting changes without having to rewrite a lot of the code-base. Flexibility is a high value for an application as it allows the developer to avoid spending too much time updating it to follow changes that depend on external factors, and, potentially, the company resources (in terms of budget, code maintenance, and time that could be spent on other features).

In the case of Kubernetes, a dashboard is, of course, heavily dependent on the platform and the resources it has to offer. Without a way of generic programming, we would have to keep track of the changes that occur in the system and update the code of the dashboard accordingly. If we consider, for example, the case of an API provided by Kubernetes in its *beta* version; the dashboard is designed, in order to retrieve it from the server and display it to the user, to make an API call to a path that is specific to the *beta* version; later, the API get updated an comes out of beta; until the dashboard is not updated, it will fetch and display an older version of the API, which is, in some cases, not acceptable. The same logic can be applied if we think about newly created APIs, that is, if we recollect our memories, the case of custom resources. Even the choice of the parameters of a resource to display can be subjected to a change in the resource definition.

That is why we decided that, rather than waiting for an API update and the change the code consequentially, it was a far better approach to make the dashboard able to react to these changes without needing to write or rewrite code, making the dashboard stateless and with no knowledge of what resources are in the server before accessing it. How this is obtained is fairly simple, but quite effective and will be described in the next section.

### 5.1.1 Discovery process

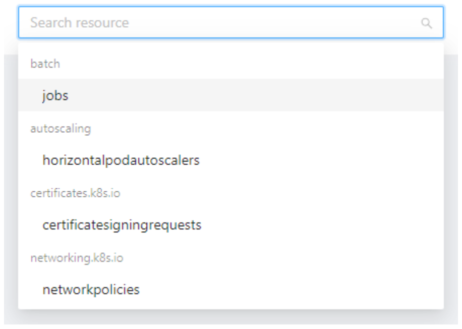
As soon as the dashboard is accessed, after the login and the process of user validation, the **discovery process** starts in background. This is one of the most important workflows of Liqo*Dash* as it is the one that will create the first wide-scoped high level representation of the dashboard, giving the user the possibility to search for and explore resources that are in the cluster, of course only if it has the rights to do so.

But what does it mean **discovery** process and how is it achieved? The dashboard, as a fundamental we impose for this project, have to be agnostic to the type of resources that reside in the cluster. That means that we cannot rely on making specific calls for specific resources because, as stated, the dashboard is not aware a priori of what kind of resources are available or not. These resources need to be discovered first, and then the dashboard can make the user interact with them. The discovery process is the process of collecting the various types of resources that the cluster has available (and the user the authorization to perform actions on) at the moment of the request and portray them in the user interface ready to be explored. Thanks to the Kubernetes API server, achieving it is just a matter of API calls. To be clearer, the discovery process workflow has been summarized in few key points:

- Question Kubernetes API sets: there are two major API sets on Kubernetes that are worthy of being displayed in the dashboard: *apis* and *api v1*. A call to these API results in a response containing a group of resource types or an array of groups of resource types.
- Iterate through groups of APIs: once obtained the references from the previous search, the dashboard iterates through the newly found groups of APIs asynchronously until the response is just a set of resources types, which is the last and deepest level of search as we, as a user, only care to know what kind of resources are available and only later the resources of a specific kind.
- Cache the server responses: at this point, to avoid having to ask again the server (which is quite costly in terms of computational time spent), the dashboard stores the essential information about the types of resources discovered (such as the name and the link to retrieve them).
- Make them available: the resources are now available to the user through the sidebar or a convenient search bar with autocomplete positioned in the header.
- Open watches: a series of watches are opened for resources or API groups that need to be kept track of because their changes have an impact on the general environment that constitutes the dashboard (for example, CRDs can be updated, added or deleted during the time the dashboard is opened and as that a watch is opened and will be kept open until the dashboard is closed).

### 5.1.2 Service-agnostic exploration of resources

The exploration of resources in a top-down fashion is one of the most unacknowledged part of Kubernetes that almost every single Kubernetes dashboard seems to not be particularly interested to display. That is because of the non-generic approach that other dashboards decided as implementation. Being able to explore APIs starting from a general level is not only useful in a variety of different cases (for example, we want to explore every Custom Resource Definition that belongs to a certain group of API), but it also comes rather easy with the adoption of a service-agnostic kind of dashboard.


**(a)** Easy exploration of resources

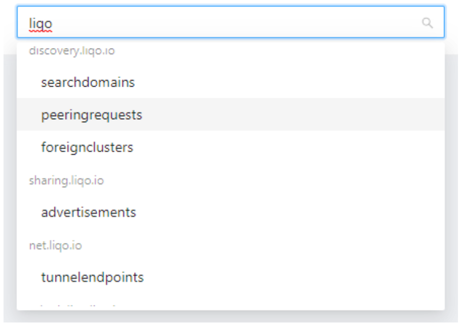

**(b)** The auto-complete functionality: searching resources that belongs to a specific set of groups

**Figure 5.1:** Search bar showing the results of the discovery process.

## 5.2 CRUD Operations on Generic Resources

The job of a dashboard is not only to represent data in a well designed and visually pleasant environment, but also to let the user interact with the resources displayed to perform more or less complex actions and workflows.

The Kubernetes API are accessed using REST calls. Representational state transfer (REST) is a software architectural style that defined a series of operations that can be performed over resources exposed through a web service. Using HTTP and Kubernetes, the operations concerned are the following:

- **GET**: retrieve a resource;
- **LIST**: retrieve a collection of resources;
- **POST**: create a resources based on the body of the request;
- **PUT/PATCH**: replace/update a resource based on the body of the request;
- **DELETE**: delete a resource or a collection of resources;

Because of the REST nature of the the Kubernetes API, once the resource is obtained (through the discovery process) and accessed, performing the classic CRUD operations over it is a rather easy task. In the next sections we will describe how these methods are implemented in Liqo*Dash* and the challenges and solutions adopted in the context of a service-agnostic and generic approach.

### 5.2.1 The challenge of a dynamic approach

With the generic method that Liqo*Dash* uses to retrieve data from the server, there are a series of challenges to take on. The first one, and by far the most concerning one, is how to handle a JSON object, that is the resource that the server responds with, which the dashboard has no idea how it is structured, and offer the user an environment where they can perform actions on it (first of all, visualize it) in an understandable and meaningful way. With this approach comes, of course, a huge loss in expressiveness, but this project provides alternative solutions that sail against the common implementations that all the other dashboards make, imposing pre-defined views to their users.

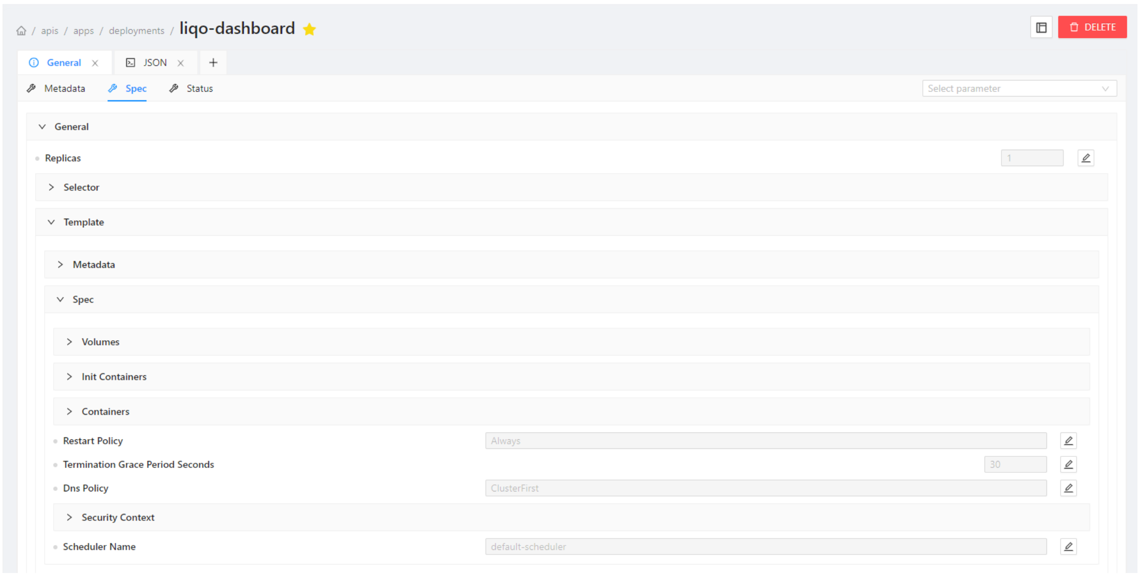
Liqo*Dash* is generic by design, and does not assume the status of the Kubernetes cluster. That means that a view that has the job of displaying a resource has to be generic too and does not have to assume the kind of resource it will be showing. As that, has to be dynamically generated according to what the interested object is and the same code needs to be the same between different kind of resources.

### 5.2.2 Generic data view

Although it is just a fraction of the architecture of a dashboard, the visual design of the user interface and the way of visual communication (also called User Experience) are one of the most interesting and difficult part to manage, as well as being extremely important for the end-user that will interact with the application. That means that the user is unaware and has no interest in knowing about the approaches that the dashboard has implemented, but what they want instead is a well designed view that would show the resource they want to explore understandable at a glance. Relying on just showing the user a YAML or JSON view of their resource is a very poor choice of design and brings little to no value to a dashboard.

Liqo*Dash* introduces a type of view that meets the need of a good visual implementation without having to renounce to its generic and dynamic approach of handling resources. The view proposed (an example of it is Figure 5.2) is dynamically generated on the basis of the resource it needs to display and offers the user an easy and intuitive way of performing rapidly all the operations that they need.

- Visualize: the resource obtained by the Kubernetes API server is processed by a component (the **dynamic form generator**) that creates a completely explorable form based on the object provided. This component is completely generic and independent of the resource it needs to display. The view can be switched to the editor mode, which will show the raw object as YAML or JSON object depending on the user preferences.
- Search: the integration of a searchbar with an auto-complete function makes the exploration of a resource an easy task.
- Update: the parameters of the resource can be updated or added by the user and validated by the Kubernetes validation system. Parameters can be modified quickly in the form or in the editor. They will also be automatically updated if they have been modified outside of the dashboard (for example directly in the cluster from someone else) thanks to the watch that will open at the opening of the resource view.
- Delete: clicking the red button will prompt a pop-up asking if the user is sure of the decision. Clicking Yes will delete the resource. There is no undo.



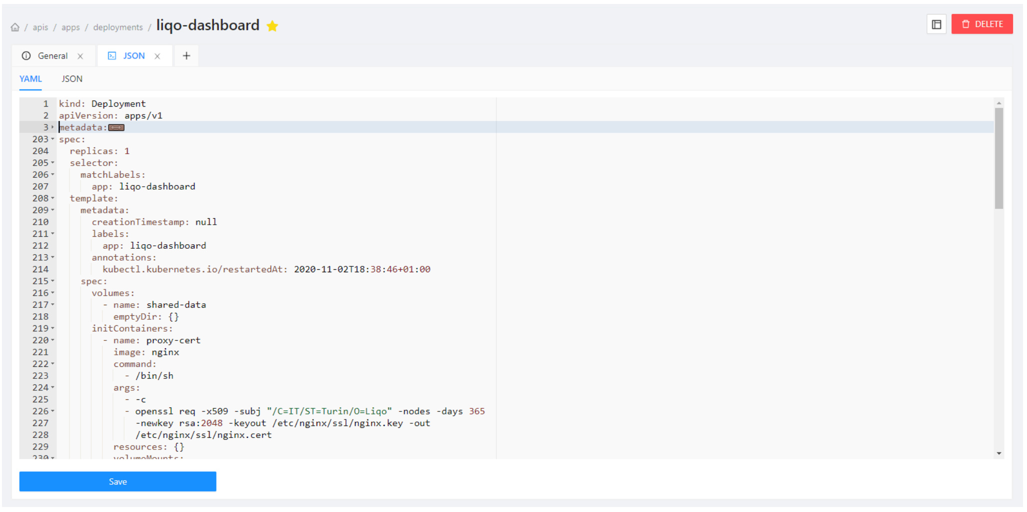
| | |
|---|---|
| Replicas | 1 |
| Restart Policy | Always |
| Termination Grace Period Seconds | 30 |
| Dns Policy | ClusterFirst |
| Scheduler Name | default-scheduler |

**Figure 5.2:** A resource of type Deployment displayed through the dynamically generated form view with ready-to-update fields.

## 5.2.3 More than just plain YAML

Although the possibility to update and manage resources through a YAML (or JSON) editor is always available, as it should be in the context of a dashboard (especially one for Kubernetes), a noteworthy amount of work has been put in the design and development of components that would make the user need to rely on it on very few occasions. The job of a dashboard, as we have said many times by now, is to give the user a sense of easiness when managing complex resources and workflows. That is why using a text editor in a UI is in part in contrast with the principles of the user experience. Sometimes there is no way to avoid it, and the translation from a text based context to a graphical and interactive one is limited or not possible, but when the possibility is there, such as visualizing generic resources, it must be grasped.

Sadly, exploring other dashboards, the trend is to have fancy and clear views of only certain Kubernetes components, leaving aside the ones that are not provided by default or are not considered important. Especially in the case of custom resources, all we can see is an editor an no more. The same could be said for creating or updating a resource. Always the editor. But the editor is not enough for a dashboard to fully serve its purpose, and the user needs more than that. This is what Liqo*Dash* offers, more than just plain YAML.



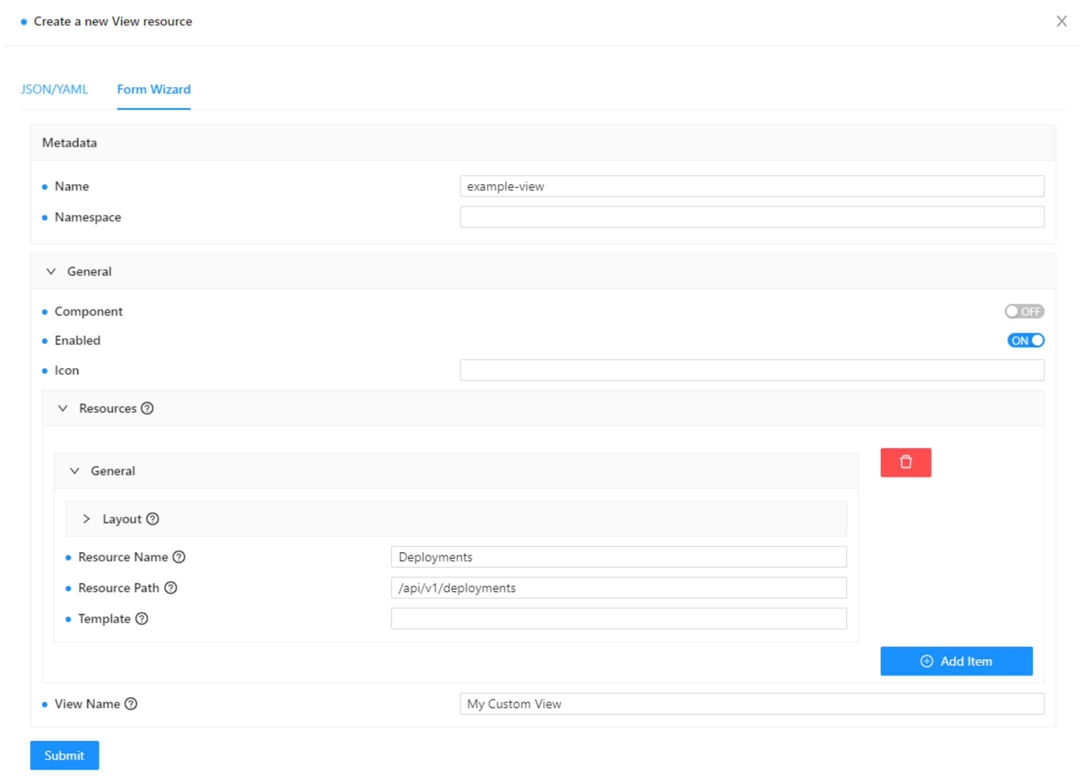
```
kind: Deployment
apiVersion: apps/v1
metadata:
spec:
  replicas: 1
  selector:
    matchLabels:
      app: liqo-dashboard
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: liqo-dashboard
      annotations:
        kubectl.kubernetes.io/restartedAt: 2020-11-02T18:38:46+01:00
    spec:
      volumes:
        - name: shared-data
          emptyDir: {}
      initContainers:
        - name: proxy-cert
          image: nginx
          command:
            - /bin/sh
          args:
            - -c
            - openssl req -x509 -subj "/C=IT/ST=Turin/O=Liqo" -nodes -days 365
              -newkey rsa:2048 -keyout /etc/nginx/ssl/nginx.key -out
              /etc/nginx/ssl/nginx.cert
          resources: {}
```

**Figure 5.3:** The editor view, used to show the resource in plain YAML or JSON.

### 5.2.4 Using the OpenAPI v3 Schema

When creating a resource (especially a custom one), the dashboard relies on the resource's validation schema, which is an OpenAPI v3 schema. With it, the form generator presented before can be expanded to a real dynamically implemented **form wizard** which is used to create, and validate the object before sending it to the Kubernetes server for validation. This saves time and, of course, the issue of writing object using a YAML or JSON editor.

Making comparisons with other dashboards, only the official Kubernetes Dashboard has made a step towards the implementation of a wizard that helps to easy the pain of creating resources, but works only for resources of type Deployment, and not at all when it comes to custom resources. In contrast, the Liqo*Dash* solution is perform extremely well in such a context and creating complex objects has become a trivial task that only requires to fill up a form.



| | |
|---|---|
| Name | example-view |
| Namespace | |
| Component | OFF |
| Enabled | ON |
| Icon | |
| Resource Name | Deployments |
| Resource Path | /api/v1/deployments |
| Template | |
| View Name | My Custom View |

**Figure 5.4:** Example of custom resource being created through the form wizard.

## 5.3 Relationships between resources

We have said how the Kubernetes API are designed to be fast and reliable and not so much aimed at an easy integration of a UI. That is because often what the user wants to see logically is scattered across multiple objects. As an example, when looking at a Deployment, the user is interested in how to access it. That requires looking at a collection of objects that are separate entities, but are all related nonetheless, like Services and Ingresses, Endpoints, Pods and Nodes.

Liqo*Dash* offers its users a way to overcome this limitation through a view that, for each resource, is able to build a dynamic dependency graph that let the user visualize and explore the objects that are in some ways related to the one we are on. Each object can be selected on the graph and its content will be described in a separate column.

### 5.3.1 Different kind of relationships

We are about to describe three different methods that Liqo*Dash* uses to understand the relationship between the resource selected and the others that reside in the cluster, and as that, create the dependency graph. Each one of these approaches are enabled by default, meaning that the dashboard will apply all of them when trying to create the graph, but there is always the possibility to disable the ones that are not needed, as the three methods are independent one from the other. The methods depicted below are ordered from the more general to the more specific in terms of resources correlation.

#### Using labels

Labels enable users to map their own organizational structures onto system objects in a loosely coupled fashion, without requiring clients to store these mappings. When objects share the same label there is a probability that they have some kind of relationship. This kind of resource linking is of course the most loose between the three we are going to explain because there is no way to guarantee that two objects that share the same label are indeed related. This is also the slowest method of generating links between resources, as it require to check every resource type in the Kubernetes cluster and apply a label selector in the API request, so that the server will respond with a list of items that contains the selected label.

### Owner References

Some Kubernetes objects are **owners** of other objects and that ownership can be used to detect correlations between objects. For example, a ReplicaSet is the owner of a set of Pods. The owned objects are called dependents of the owner object. Every dependent object has a *metadata.ownerReferences* field that points to the owning object. These values can be set automatically by Kubernetes or can be specified manually by setting the proper field. This is a mechanism by which the Kubernetes garbage collector controller works. Cross-namespace owner references are disallowed by design. This means that namespace-scoped dependents can only specify owners in the same namespace, and owners that are cluster-scoped and cluster-scoped dependents can only specify cluster-scoped owners, but not namespace-scoped owners.

### Direct object reference: a new approach

A new method, developed in this thesis, to specify the relationship between two object is to directly include a reference to a resource in the schema of the resource. Of course, because the user can only define the schema of a resource through the Custom Resource Definition API, only custom resources can implement this method of linking. The way it works is rather simple: in the resource schema the reference is created including an object which name that is formatted in a specific fashion: *resource-group/resource-kind*. It has two parameters which are the *name* of the resource referenced and the *namespace*, in the case of a cluster-scoped resource. With that, there is no cross-namespace limitation, but the user need to have authorization to access all the namespaces interested in the relationship for it to be displayed.

Liqo*Dash* also handles this kind of references when a resource is being created or updated, letting the user select a resource of the type referenced instead of having to input the name and the namespace, avoiding common and mistakes.

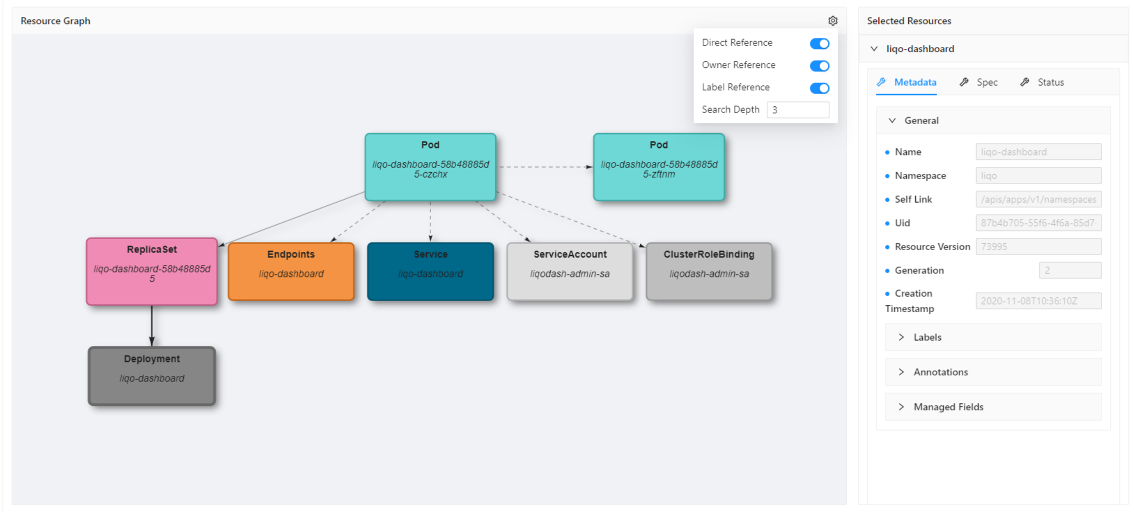
```yaml
openAPIV3Schema:
  properties:
    ...
    spec:
      type: object
      properties:
        reference:
          type: object
          properties:
            dashboard.liqo.io/MiddleRef:
              type: object
              properties:
                name:
                  type: string
                namespace:
                  type: string
```

**Listing 5.1:** Example of a custom resource's schema with direct object reference to a resource of type **Middle**.

### 5.3.2 Design of data trees and level of search depth

The view is generated using a custom version of the **Vis.js** library for React to display networks consisting of nodes and edges. Nodes represent the resources, and a node can indicate one or multiple (clustered) objects. Edges represent the relationship between resources and are designed in different ways associated to the different methods that they represent (e.g. dashed line for the label method, a straight line for the owner reference). The graph is automatically updated every time a direct link is created. To avoid computational overhead and establishing an excessive amount of open connections with the server, the dashboard will not react to indirect changes happening in regarding of the first two methods of exploration of relationships. Only if the main resource (the one that we are visualizing) is modified (i.e. a new label has been applied or a new owner reference has been selected) the graph will be updated.

The graph generation inherits the service-agnostic nature of the dashboard, and the three methods of links search are only dependant on the actual resource we are currently viewing. That means that, through the use of recursive functions that only need a starting resource, a graph could have, in theory, an infinite depth. The right compromise between good performances in terms of computational time spent in generating the graph and clear exposition on all the most important relationships between resources is to consider depth the of the graph of two layers, as such it is designed to be the default one. As a result, the view will be showing, by default, the resources that are directly related to the resource in question and also the one ones that are related to them (as shown in the Figure 5.5). Depth of search can be changed by the user in the settings.



| | |
|---|---|
| Direct Reference | ✓ |
| Owner Reference | ✓ |
| Label Reference | ✓ |
| Search Depth | 3 |

| | |
|---|---|
| Name | liqo-dashboard |
| Namespace | liqo |
| Self Link | /apis/apps/v1/namespaces |
| Uid | 87b4b705-55f6-4f6a-85d7- |
| Resource Version | 73995 |
| Generation | 2 |
| Creation Timestamp | 2020-11-08T10:36:10Z |

**Figure 5.5:** Example of a dynamically generated dependency graph of a Pod resource with all the methods of relationship search activated and a level 3 depth of search. Its Deployment is selected on the right column.

# A Fully Customizable Dashboard

As we have said in the introduction, a completely generic and dynamic approach comes with its downsides. The knowledge of what to expect from a resource makes it easier to show what is really important to display and what not, creating ad hoc views for specific kind of resources. On the other hand, a generic code-base can only rely on what is common for all resources. In this chapter we explain the solutions adopted to grant Liqo*Dash* the same level of expressiveness that other, more specific dashboards have and how this is, in fact, one of the strong points of the dashboard that is capable of solving numerous resource-displaying related problems.

## 6.1 The importance of customization

A concept that other dashboards do not have taken into account is the one of view customization. This is one of the key flaws that we have identified in all of the current solutions for a Kubernetes user interface. Reading through various blogs around the internet it seems that there are more than few users that lament the fact that there is no way to add useful parameters that are not displayed in the default views to the page, or, for example, add custom columns to filter resources through certain object fields.

This project addresses that problem and proposes a way to give the user the possibility to use the default view that the dashboard provides, or completely overturn it into something that is more appropriate for the user's needs. We know well that a dashboard is designed for the users, but different users may need different visualization of the same resource, and who better than the user itself knows what is the best for their cases? What Liqo*Dash* offers is a powerful, fully customizable environment that the user can modify at will, only keeping what is necessary, and if there are changes to be made, these are extremely easy to make. All without writing a single line of extra code: it is all managed by the dashboard.

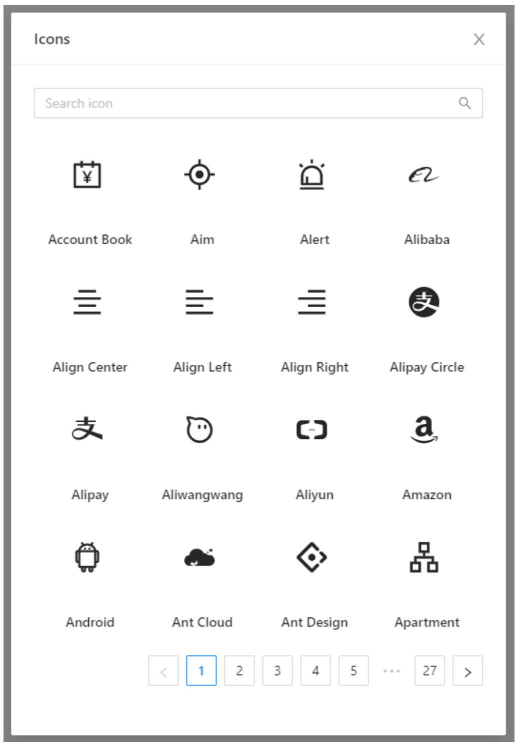
## 6.2 Icons and Favourites management

Before talking about the implementations and the solutions adopted between different customizable views, it is noteworthy taking a more general overview of the basic management systems shared through all the views. There are two features that are relatively easy to build, and give a huge contribution in improving the user experience. Having them is a great advantage with a little amount of work, and are two of those features that the end-user does not really know they need, but once they are accustomed to it, it is difficult to not rely on them. We are talking about the **favourite management** and the **customizable icon system**.

#### Icons

At first impact, having the possibility to customize the icon of a view may seem to be a nice feature, but rather useless for the user experience and just an aesthetic improvement of the dashboard. It is reasonable to consider useful only what it makes some jobs done (e.g. the implementation of a tedious workflow in just one click). But let us not forget about the elements that many times they are overlooked by the programmer or the architecture manager, but are nonetheless important in granting the end-user a good quality of experience. The visual design is fundamental in a dashboard, and there is no way to deny that. That is why Liqo*Dash* comes with a feature that let the user customize their resource's icons.

**(a)** The user can select between a set of over 400 icons



**(b)** Favourite resources are displayed in the sidebar

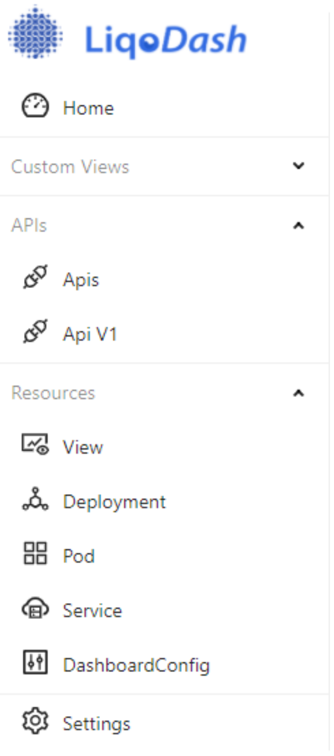


**Figure 6.1:** Select and implementation of icons and favourites.

### Favourites

The user can mark every resource or group of resources as favourite which is a way to pin the resources to make them readily available for use. What happens behind the scenes when marking a single resource as favourite is that it will create a "*favourite: true*" annotation in the resource metadata. When a resource is removed from the favourites, the annotation will be deleted. When a list of resource is marked as favourite, we need a more high level way of keeping track of it, and this is done thanks to the use of a particular custom resource called the Dashboard Configuration that we will explore deeply in the next sections. At the start of the application, Liqo*Dash* will check if there are resource list marked as favourite in the Dashboard Configuration and will act accordingly displaying them in the sidebar, easily accessible by the user.

## 6.3 The Dashboard Design Configuration

In this section we are going to explain how it is managed the full customization of views that Liqo*Dash* provides. First, we need to introduce a Custom Resource called **DashboardConfig** that is the object in which the **design configuration** of the dashboard's views are stored. Because it is a separate YAML object, it is completely independent from the dashboard code, which only contains generic components that process and display the design depicted in the configuration. This grants the user the power to only display what they want and how they want. With the definition of a design configuration, not only the dashboard can have all the expressiveness that other more specific dashboards have (with less code and less time spent), but is also a step further in that regard, giving the user full control over its dashboard instead of imposing a view that often does not suite all users' needs.

### 6.3.1 Resource List view customization

The **Resource List** view let the user have a look at all the resources they have the right to interact with, exposing them in a table with full customizable columns which purpose is to show general details that give the user an overview of each resources in a compact fashion. In this view the user can also create a resource of the type visualized.

The table is composed of three static columns, that represents the essential details that represents a resource of every kind, and as such they cannot be modified or removed:

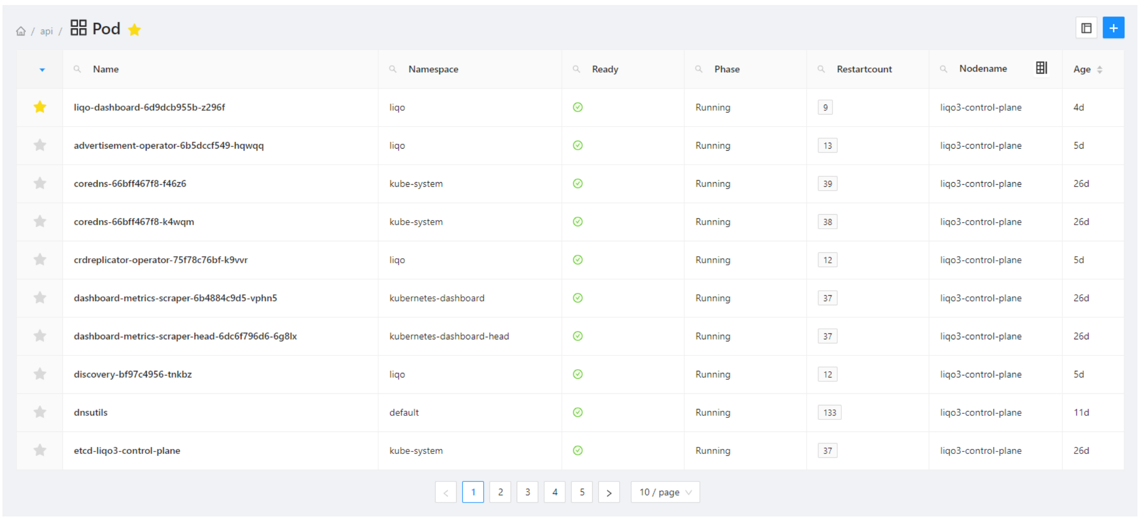
- Favourite column: through this column, resources can be marked or unmarked as favourites. It is always the first column.
- Name column: is the name of the resource as given by its metadata field. Resources in Kubernetes cannot exist without a name, and names cannot be changed after the resource is created. So this column is treated as static and always present.
- Age column: this column shows how much time has passed from the creation of the resource. It is always the last column.

The other columns of a resource list table are can be modified, removed, or added at the user's own discretion. The dashboard provides a default configuration for all the common and most used resources of a Kubernetes cluster, but in cases where that is not possible (e.g. custom resources that only the user know about) the dashboard offers a way to implement the parameters that are more appropriate with ease and just in a matter of few steps.

Clicking the *add column* button will create a new column as the second-to-last. A searchbar with autocomplete will let the user select the parameters that the resource kind has available as described in its OpenAPI v3 schema. The autocomplete function does not let the user select a parameter that is not defined in the resource's schema to prevent the input of wrong data. The newly created column can be saved and will now show, for each resource displayed, the corresponding value of the parameter. For each column there can be specified more than just one parameter and even some text to show in each cell, or a basic arithmetic calculation between parameters. For example if we want to create a column that shows the ratio between *available replicas* and *total replicas* of a Deployment, it can be done providing the first parameter and the second with a divisor (such as "/") between the two. That way we have created a column that shows a more complex situation and is more clearly understandable than just single parameters in different columns.

Saving the column will also save the new design in the **dashboard configuration CRD**. If the resource kind has not already an entry in the configuration, meaning that no customization has been performed on it or its resources, it will be generated and added. The column is saved in an array of objects simply called *columns* and is composed of two inner objects: an list of parameter to display and the name of the column. The name of the column is a string auto-generated at the act of saving the column, it is meant to describe the content of the column and can be changed by the user at any time just by clicking the column header. Every column of the table also comes with a filtering option next to its name, and it is a feature available regardless of the column being a default or a custom one. Simply clicking its relative search icon will open a pop-up that will let the user input a search query. Resources can also be filtered by favourites for an easy search.

In the picture below (Figure 6.2) we can see a custom-made view that describes the resource of kind Pod. Aside from the default columns, we have added the *Namespace* that shows the namespace that the Pod belongs to and the *Phase* that describes where the Pod is in its life cycle. It is also worthy to notice that two columns are not represented as strings. That is because the parameter *Ready* is a boolean, and boolean are represented with a green check if true or a red cross if false, to help the user understand the situation with a quick glance. The column is also able to manage an array of parameters as input, and they are represented as *Tag* components, and an example of that is the *Restartcount* column.



| | Name | Namespace | Ready | Phase | Restartcount | Nodename | Age |
|---|---|---|---|---|---|---|---|
| ★ | liqo-dashboard-6d9dcb955b-z296f | liqo | ✓ | Running | 9 | liqo3-control-plane | 4d |
| ☆ | advertisement-operator-6b5dccf549-hqwqq | liqo | ✓ | Running | 13 | liqo3-control-plane | 5d |
| ☆ | coredns-66bff467f8-f46z6 | kube-system | ✓ | Running | 39 | liqo3-control-plane | 26d |
| ☆ | coredns-66bff467f8-k4wqm | kube-system | ✓ | Running | 38 | liqo3-control-plane | 26d |
| ☆ | crdreplicator-operator-75f78c76bf-k9vvr | liqo | ✓ | Running | 12 | liqo3-control-plane | 5d |
| ☆ | dashboard-metrics-scraper-6b4884c9d5-vphn5 | kubernetes-dashboard | ✓ | Running | 37 | liqo3-control-plane | 26d |
| ☆ | dashboard-metrics-scraper-head-6dc6f796d6-6g8lx | kubernetes-dashboard-head | ✓ | Running | 37 | liqo3-control-plane | 26d |
| ☆ | discovery-bf97c4956-tnkbz | liqo | ✓ | Running | 12 | liqo3-control-plane | 5d |
| ☆ | dnsutils | default | ✓ | Running | 133 | liqo3-control-plane | 11d |
| ☆ | etcd-liqo3-control-plane | kube-system | ✓ | Running | 37 | liqo3-control-plane | 26d |

**Figure 6.2:** Example of the resource list view displaying Pod with custom columns.

### 6.3.2 Single Resource view customization

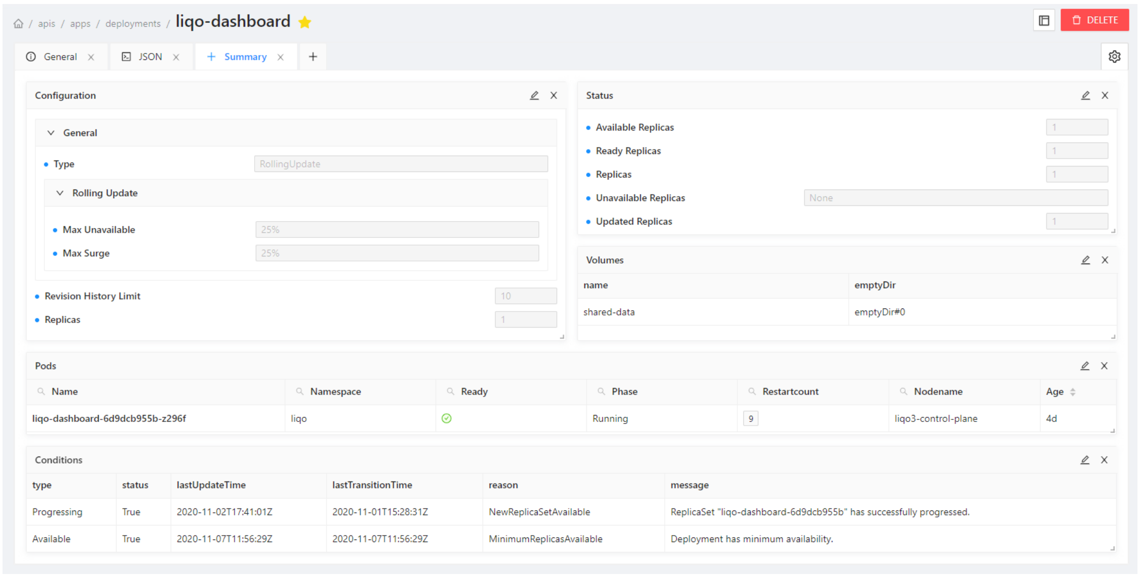
Regarding the view that shows the details of a single resource, in the previous chapter we have described that it contains two default tabs: the one containing a dynamically generated form created from the JSON object returned from the server and another one containing an editor view, displaying the object in plain YAML or JSON. Sometimes that is not enough as a view describing the most important details, leaving aside the ones that are not as interesting, in a compact and immediate way can really improve the user experience.

Because of that, Liqo*Dash* offers a tool to create custom tabs specific for single resources kind in order to give the user full controls in managing their resources, especially if we are talking about custom ones. A custom tab is essentially a blank working space where the user can add, remove and manage their own contents in the form of cards components. This working space is wrapped in a customizable grid layout and every card on it can be dragged around, resized at any time and its layout is automatically saved in the dashboard configuration resource. A card is the singular component that is stored in a tab. More cards can be composed together to create a single tab and are generally used to tie in a single space parameters that are somewhat related (for example Metadata fields or Status fields).

Creating a new card is easy thanks to custom context menu of the tab, which let the user add objects simply with the mouse right click. A card can be of various types:

- List: a simple list of parameters chosen by the user. Useful when we want to display a list of separate parameters or an object;
- Table: useful when we want to display an array of objects;
- Reference: shows a different kind of resource chosen from the list of resources available in the cluster. Useful when we want to display a resource list related to the resource (e.g. a deployment and its pods);

Creating a new tab will also save the new design in the **dashboard configuration CRD**. If the resource kind has not already an entry in the configuration, it will be generated and added. The tab is saved in a collection of objects (each one represents a single tab) and the object itself is composed of a list of cards that the tab contains and the name of the tab, which can be changed by the user at any time simply clicking the tab name in the resource view. Each card that belongs to a tab is also an object composed of a collection of parameters, the kind of the card (list, table or reference) and its name, and all can be modified directly in the dashboard.



Configuration

General

- Type: RollingUpdate

Rolling Update

- Max Unavailable: 25%
- Max Surge: 25%

- Revision History Limit: 10
- Replicas: 1

Status

- Available Replicas: 1
- Ready Replicas: 1
- Replicas: 1
- Unavailable Replicas: None
- Updated Replicas: 1

Volumes

| name | emptyDir |
|---|---|
| shared-data | emptyDir#0 |

Pods

| Name | Namespace | Ready | Phase | Restartcount | Nodename | Age |
|---|---|---|---|---|---|---|
| liqo-dashboard-6d9dcb955b-z296f | liqo | ✓ | Running | 9 | liqo3-control-plane | 4d |

Conditions

| type | status | lastUpdateTime | lastTransitionTime | reason | message |
|---|---|---|---|---|---|
| Progressing | True | 2020-11-02T17:41:01Z | 2020-11-01T15:28:31Z | NewReplicaSetAvailable | ReplicaSet "liqo-dashboard-6d9dcb955b" has successfully progressed. |
| Available | True | 2020-11-07T11:56:29Z | 2020-11-07T11:56:29Z | MinimumReplicasAvailable | Deployment has minimum availability. |

**Figure 6.3:** Example of a custom tab created for the Deployment resource kind.

# Modular Environment

Liqo*Dash* is born to give the users an alternative when choosing the proper UI for their Kubernetes system. But it has come with the time a more wide-spread project, and constraining it with just the definition of *dashboard* would be an understatement. That is because Liqo*Dash* is indeed a **platform** and a **framework** that allows users and developers to create their own user experiences that have Kubernetes at the core. The goal is to also help teams reduce the time spent on developing interfaces and incorporate different ones with different workflows all in a single working space.

Trying out different tools and dashboards, each of them was missing one, extremely important thing: the ability to seamlessly extend them and create custom views to visualize and interact with resources in the Kubernetes cluster. That is where Liqo*Dash* comes in to play: the easily extensible platform that this project offers, the ability to override the default views of the dashboard, is what makes Liqo*Dash* a highly powerful tool.

In this chapter we describe the solutions adopted in these regards, what can be done directly in the dashboard just through the user experience, and what instead requires a little bit of code-work by developers to create custom and complex workflows to build smart, insightful and actionable interfaces around resources provided by Kubernetes.

## 7.1 Concept of modular dashboard

The default views provided by the dashboard describes just the fundamentals of the basic resources the Kubernetes makes available, and are meant to give the user a general idea of the system, allowing then to keep exploring one resource at a time if interested. We talked about how Kubernetes API design is not particularly fitted for a UI exploration of that kind, because often what the user really need to view is scattered through various object, and that makes understand the correlations between resources quite difficult, if we just can view one resource at a time. The customization offered by Liqo*Dash* is helpful in that sense, as it allows to reference other resources to the one we are currently on, but we need a more generic way of organizing objects in a way that is meaningful for the user and, as for everything that needs interaction in the dashboard, rather easy to do.

That is the reason behind the implementation of a modular dashboard: a dashboard that can be expanded by the user through the creation of custom view, that can be either collection of resources, of completely custom components designed ad hoc by the developers which implement custom workflows.

## 7.2 The Custom View CRD

We introduce now the means through which a custom view life cycle is managed. Every custom view in the dashboard is stored as a custom resource of type **View**, defined by its Custom Resource Definition. This resource describes each custom view individually and is unique in its namespace, so that in the context of a large used cluster, users with different privileges will have access only to certain custom views that they have the right to visualize, or simply to keep a generic user out of the scope of certain custom workflows that are meant for a restricted group (for example developer that are working on a new custom component for the dashboard and do not want it to be used by non-developer's hands).

The **View** CRD is divided in two main sections: the first one describes generally the custom view, while the second describes the resources that the view should contain. More in detail, for the first part we have:

- *viewName*: the generic name of the view, which is different from the one gave at its creation, which is stored in the metadata and is immutable. In contrast, this name can be modified at any time.
- *enabled*: when a custom view is created, but no longer needed, there is no need to delete it, as it can be disabled just by setting this boolean to false. When disabled, a custom view will not be considered in the pool of accessible custom views, and as that will not be displayed in the sidebar.
- *component*: this boolean is used to differentiate between a custom view that contains only resources or groups of resources (such as a list of Deployment, a specific Service or a custom resource) and a custom view that contains a custom component. That because the loading method between the two types of custom view is different, and will be described later.
- *icon*: as we have said in the previous chapters, defining custom icons is particularly important for the user experience, and as that Liqo*Dash* offers the possibility, although it is not a requirement, to personalize a custom view by defining an icon to represent it.

Each custom view allows an indefinite amount of resources and components that can be displayed in a single page, and it is up to the end-user choosing the right balance between what should be shown and what not. What the custom view offers is a way to describe the resource and its layout in the view, that brings us to describing the second section of the CRD. For every resource we have the following parameters:

- *resourcePath*: The path of the resource or the custom component we want to display in the view. In case of a resource, it has to be a valid path which refers to a valid API endpoint or the resource will not be considered. In case of a custom component, it has to reference an existing component, or an alert will be prompted.
- *resourceName*: this is the name of the resource displayed in the view. It can be whatever name the user choose and is not bound by any constraints. If it not specified, an auto-generated one derived from the *resourcePath* parameter will be used.
- *layout*: defines the layout of the resource in the view, which is an object that describes the height, width, and coordinates of the card wrapping the content of the resource. It is automatically updated every time it is changed.

## 7.3 More resources per view

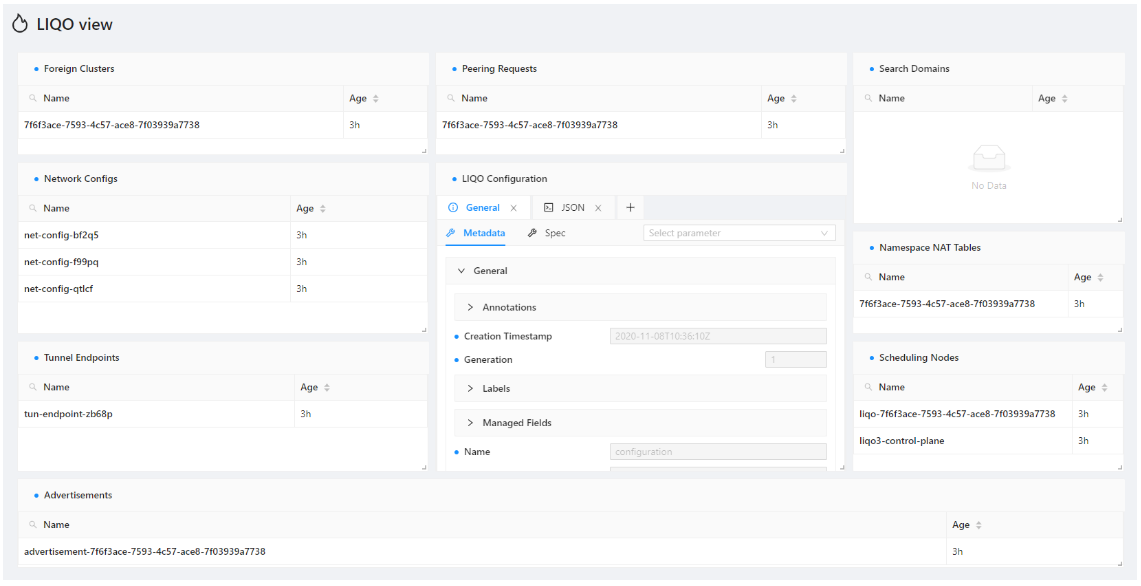
Liqo*Dash* offers the user the possibility to create its own custom views as a page that displays a set of resources of choice, useful to keep track of different resources at the same time, without the need to switch from one page to another. This can be all done directly in the dashboard an without needing to write a single line of code or having to deal with YAML.

As it can be seen in the figure below (Figure 7.1), we have together in a single view three different resource lists. Every card in the view has its own layout that can be changed at any time. The general feature that comes with the workplace set up by default by the custom view are:

- Drag and drop: every card can be moved in the space of a custom view just by dragging it through the title bar and then placed where the user finds more suited. The cards are automatically vertically packed.
- Resizable: the size of card is in the hand of the user, which can use the *resize handle* at the bottom-right corner of the component to make it bigger or smaller.
- Easy resource exploration: everything that is displayed in a card has a reference link to its own single view. In the case of a resource list displayed, every resource can be accessed directly through the custom view as the dashboard itself manage these links.
- Automatic save: every change in the view's layout is automatically saved, so every time the user access a custom view they can expect every card to be in the same position as they left it the last time.

The view is responsive and support separate layouts per responsive breakpoint, that means that when resizing the browser window (changing breakpoint) the new layout will be saved as a different one and will not overwrite the previous.

One of the key feature of Liqo*Dash*, as we have said many times, is the real-time reaction to cluster events. That means that if something is changed on the cluster it is automatically reflected on the view you are on, if we are interested on it. The custom view is no exception, so if a resource that is part of the set of resources that we are displaying in the view is updated, the changes will be reflected in the view without the need to refresh the page. This also applies if a resource is added/deleted from the set of resources in the custom view.



LIQO view

Foreign Clusters

| Name | Age |
|---|---|
| 7f6f3ace-7593-4c57-ace8-7f03939a7738 | 3h |

Peering Requests

| Name | Age |
|---|---|
| 7f6f3ace-7593-4c57-ace8-7f03939a7738 | 3h |

Search Domains

| Name | Age |
|---|---|

No Data

Network Configs

| Name | Age |
|---|---|
| net-config-bf2q5 | 3h |
| net-config-f99pq | 3h |
| net-config-qtlcf | 3h |

LIQO Configuration

General | JSON | +

Metadata | Spec | Select parameter

General

Annotations

Creation Timestamp: 2020-11-08T10:36:10Z

Generation: 1

Labels

Managed Fields

Name: configuration

Namespace NAT Tables

| Name | Age |
|---|---|
| 7f6f3ace-7593-4c57-ace8-7f03939a7738 | 3h |

Tunnel Endpoints

| Name | Age |
|---|---|
| tun-endpoint-zb68p | 3h |

Scheduling Nodes

| Name | Age |
|---|---|
| liqo-7f6f3ace-7593-4c57-ace8-7f03939a7738 | 3h |
| liqo3-control-plane | 3h |

Advertisements

| Name | Age |
|---|---|
| advertisement-7f6f3ace-7593-4c57-ace8-7f03939a7738 | 3h |

**Figure 7.1:** Example of a custom view that includes different list of custom resource types and single resources, used to keep under control a custom system.

## Creating a custom view

Creating a custom view can be done in three ways:

- From the **sidebar**: clicking the *New Custom View* button will open a menu where the user can specify a name for the custom view as well as choose the resources groups they want to include with a convenient selector. Custom views created this way will be placed in the default namespace, or in a namespace where the user has the right to create resource of type View.
- From the **resource view**: clicking the *layout button* will pop-up a dropdown menu. Selecting *New Custom View* will open the same menu specified in the first method, but this time the resource viewed will be already included in the set of resources in the custom view. As said previously, creating a custom view this way will be place it in the default namespace, or in a namespace where the user has the right to perform such action.
- From the **View CRD**: because every custom view is, in fact, just a resource of type View, the user can create one as they would do with every other resource, exploiting the capabilities of the dashboard. This way the user has more freedom in the creation of their resource.

### Adding and removing a resource from a custom view

While on the page of a resource or group of resources, the user can select the *layout button*. This will show a dropdown menu where the user can all the custom views they have access to in the cluster. If the name of a custom view is red it means the resource is already in that custom view.

- Clicking on a neutral Custom View will add the current resource to the selected custom view.
- Clicking on a red Custom View will remove the current resource from the selected custom view.

As we said earlier, a custom view is nothing more than a resource of type View. As such, it can be modified at any time in the resource of type View page. Users can add or remove a resource in the editor.

### Deleting a custom view

Deleting a custom view is the same as deleting a resource of type View. Going in the resource's page will let the user delete the resource clicking the appropriate button.

## 7.4 Dynamic loading of custom components

Liqo*Dash* offers the possibility for developers to implement custom workflows that require writing their own custom components and that can be integrated in the dashboard as plugins managed by a custom view resource. That comes within the concept of *modularity* and *dashboard as a framework*, exploiting the many widgets the dashboard provides as well as an easy to use API manager that handles all the interactions with Kubernetes and the watches, granting always full responsiveness of the resources involved in the workflow.

Although managing custom components through a custom view resource is not at all mandatory for the dashboard to be used as a framework and work with the implementation of user's plugins, it has some advantages that may be useful to a developer:

- No need for a hard-coded route path to access the view, it's all managed by the custom view loader;
- Custom components are directly accessible in the sidebar and they can be enabled or disabled like every custom view;
- The layout of the view can be be managed through the custom resource;
- Custom components are loaded only if their custom view is enabled, making the loading of components completely generic and independent from the component itself. It is useful if, using the dashboard as a framework, we want to expand the set of custom components, making the linking between the dashboard and the custom component as easy as creating a custom view resource.

We will see some examples of integration of custom components in the later chapters, and how it is an easier an reliable solution compared to developing a dashboard from scratch.

# Integration with existing projects and Performance

To serve its purpose and fulfill its potential, the dashboard needs a real implementation: some projects to use as test cases that could show the validity of this approach. Liqo and CrownLabs are the perfect choice: two project developed throughout the year at Politecnico of Turin that are built upon Kubernetes and relies heavily on the use of custom resources. The idea was to use the dashboard as a framework and exploit its generic and modular approach to make custom views specific for Liqo and CrownLabs.

It this chapter we are going to explore these two use cases and the custom views and components created and integrated in Liqo*Dash*. Finally we are going to give a general overview on the performance of the application in relation to other dashboards that we have listed in Chapter 2.

## 8.1 Liqo

Liqo is an open source project started at Politecnico of Turin that allows Kubernetes to seamlessly and securely share resources and services, so you can run your tasks on any other cluster available nearby.

Thanks to the support for K3s, also single machines can participate, creating dynamic, opportunistic data centers that include commodity desktop computers and laptops as well.

Liqo leverages the same highly successful “peering” model of the Internet, without any central point of control. New peering relationships can be established dynamically, whenever needed, even automatically. Cluster auto-discovery can further simplify this process.

Sharing and peering operations are strictly enforced by policies: each cluster retains full control of its infrastructure, deciding what to share, how much, with whom. For security we leverage all the features available in Kubernetes, such as Role-Based Access Control (RBAC), Pod Security Policies (PSP), hardened Container Runtimes Interfaces (CRI) implementations.

With Liqo, there is no disruption neither in the common Kubernetes administration tasks nor from the user perspective because everything happens as your cluster gets bigger. And, for Liqo admin tasks, a dedicated GUI (Liqo*Dash*) will bring users to their objective in a few clicks.

Finally, according to the sharing economy principles, Liqo is also more energy efficient, for the benefits of our planet as well.

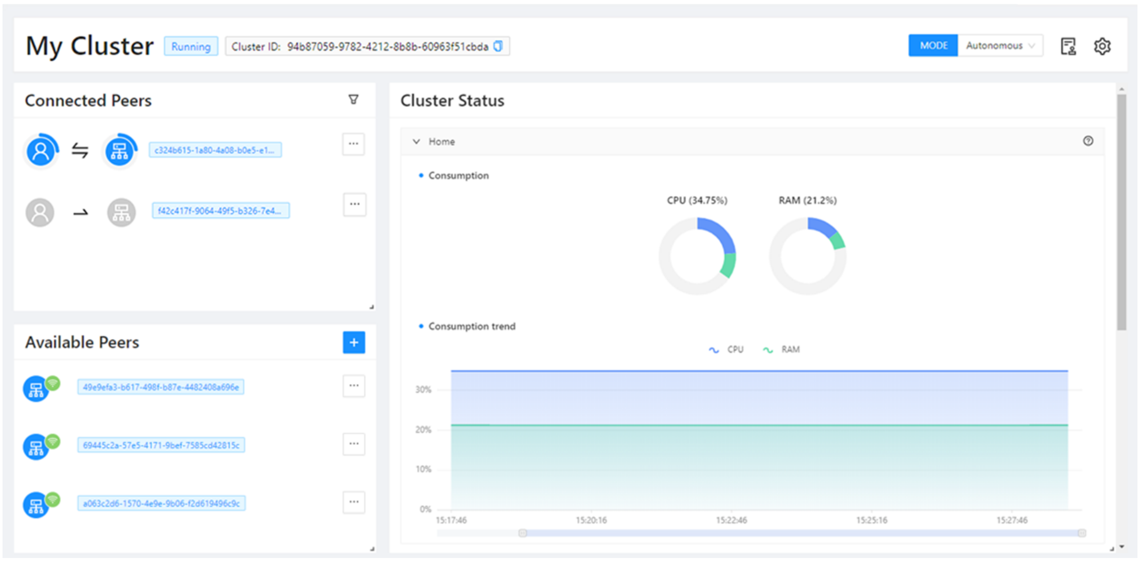
### 8.1.1 Liqo*Dash* meet Liqo

The first view presented is the Liqo Home. This page contains general information about the status of the user’s cluster and their Liqo resources, such as clusters you are connected to and the available ones. It is divided in various sections that will be discussed in detail.

#### Header

This first section is mean to show the status of the Liqo system and its working mode. It includes:

- **Cluster Name**: the name of the cluster. Can be modified at any time just by clicking on it.
- **Status**: it can be Running or Stopped.
- **Cluster ID**: a unique identifier of your cluster.
- **Mode**: in this dropdown menu the user can choose between various working modes that Liqo supports.
- **Policies**: a link to the Liqo policies page, where the user can set their preferred policies.
- **Settings**: a link to the Liqo configuration page, where the user can set their preferred configuration for their system.


My Cluster Running Cluster ID: 94b87059-9782-4212-8b8b-60963f51cbda

MODE Autonomous

Connected Peers
c324b615-1a80-4a08-b0e5-e1...
f42c417f-9064-49f5-b326-7e4...

Available Peers
49e9efa3-b617-498f-b87e-4482408a696e
69445c2a-57e5-4171-9bef-7585cd42815c
a063c2d6-1570-4e9e-9b06-f2d619496c9c

**Figure 8.1:** Liqo Home view that shows available and connected clusters, as well as the status of the cluster itself.

## Connected Peers

In this section the user can see the clusters they are connected to. For each peering it is displayed the direction of the connection, that let the user know if they are connected to the foreign cluster, if someone is connected to them or both. Having established a connection doesn't mean that the user is actually sharing resources, but just that *they can*. To show if resources are actually shared between the user's and the foreign cluster, either offering and/or consuming, connections are colored in different ways: **blue** if there is actual resource sharing or **gray** if there is not.

If there is resource sharing, the amount of memory consumed in a cluster, in relationship to how much is made available to share, can be seen around the interested cluster, and hovering over it with the mouse will show the actual percentage of memory consumed. For example, in the picture above, the first peering represent a bidirectional connection where both the home cluster and the foreign one are using resources of each other. In the second case, we can see that the connection is only from the home cluster to the foreign, meaning that the user could use resources of the foreign cluster, but the other way around is not possible. Also the connection is gray, meaning that there is no actual resource sharing.

The user can disconnect from a peer just by clicking *Disconnect* in the peer's menu or selecting the connection and clicking the *Disconnect* button.

### Details

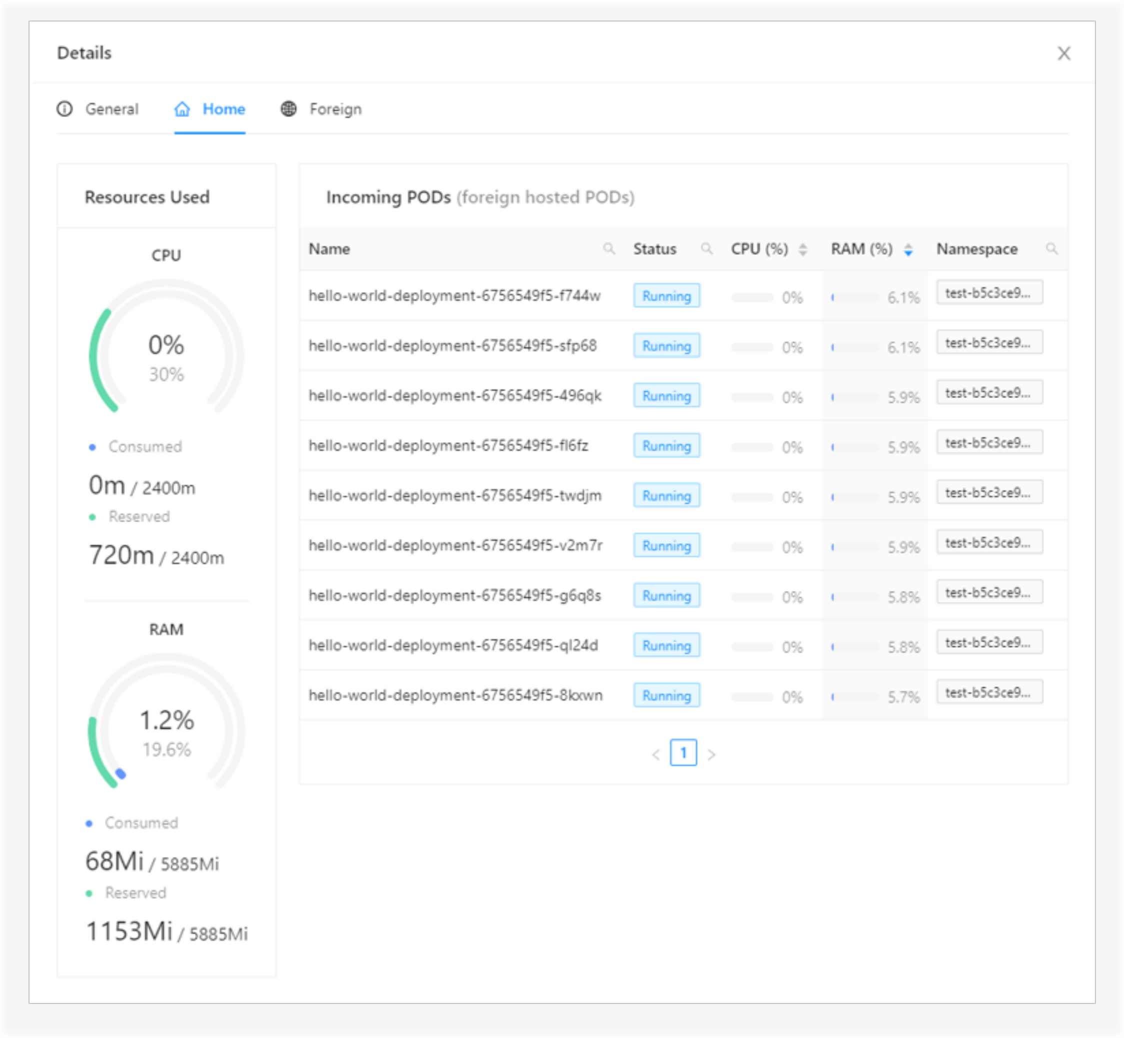
The details of a particular peering can be viewed selecting it from the dropdown menu, placed right beside the name of the foreign cluster the user is connected to. This view displays some general information about the peering, such as the direction of the connection, and some more in depth details about the status of both the home and the foreign cluster. In particular we can see:

- Two double gauge-shaped progress bars that represent how much CPU and RAM is both consumed and reserved in percentages and in units (m for CPU and Mi for memory);
- A table that lists all the Pods offloaded to the other's cluster (on the Foreign tab) or the Pods that the one the connected cluster is offloading to the user's one (in the Home tab), along with the status of each POD;

Metrics are calculated in different ways:

- The **Reserved** percentage is the sum of each Pod's requested resource allocation, in relationship to how many shared resources are established during the peering. These resources are guaranteed to be available. If a Pod (or one of its containers) does not specify requested resources will always be considered as if it has zero resources reserved.

- The **Consumed** percentage is the sum of each Pod's used resource over the total of shared resources agreed. Information about the real usage of CPU and memory are given by a metrics server deployed in the cluster to which Liqo*Dash* has an integration with. If there is no metrics server available the Consumed metrics will be the same as the Reserved ones, as the dashboard assumes the user is consuming as much as requested.



Incoming PODs (foreign hosted PODs)

| Name | Status | CPU (%) | RAM (%) | Namespace |
|---|---|---|---|---|
| hello-world-deployment-6756549f5-f744w | Running | 0% | 6.1% | test-b5c3ce9... |
| hello-world-deployment-6756549f5-sfp68 | Running | 0% | 6.1% | test-b5c3ce9... |
| hello-world-deployment-6756549f5-496qk | Running | 0% | 5.9% | test-b5c3ce9... |
| hello-world-deployment-6756549f5-fl6fz | Running | 0% | 5.9% | test-b5c3ce9... |
| hello-world-deployment-6756549f5-twdjm | Running | 0% | 5.9% | test-b5c3ce9... |
| hello-world-deployment-6756549f5-v2m7r | Running | 0% | 5.9% | test-b5c3ce9... |
| hello-world-deployment-6756549f5-g6q8s | Running | 0% | 5.8% | test-b5c3ce9... |
| hello-world-deployment-6756549f5-ql24d | Running | 0% | 5.8% | test-b5c3ce9... |
| hello-world-deployment-6756549f5-8kxwn | Running | 0% | 5.7% | test-b5c3ce9... |

**Figure 8.2:** The details view shows the detail of a single cluster connected.

### Properties

This menu shows the Liqo resources associated with a specific peer (either available or connected). For every resource the user can modify its spec directly in the view (the status and metadata are readonly, as the user is not supposed to modify them). The resources are:

- **Foreign Cluster**: it is the resource that describes the peer and contains information like the cluster ID or the status of the peering. It is always present.
- **Advertisement**: it describes what the other cluster is offering in terms of hardware capabilities or software.
- **Peering Request**: a request sent by a cluster to create a peering with another cluster.

### Available Peers

In this section are listed the peers that are available to connect to. The user can see some general information regarding the peer, such as its name or if the connection is made through LAN or an external network. Clicking on the peer (or the dropdown menu in the right) will let the user see more details about the available peer and the possible connection. If the user wants to connect to an available peer, they can do so just by clicking the *Connect* button or select *Connect* in the peer's menu. This will trigger the connection process, and its various phases will be described as a side note in the peering section. The user can also decide to stop the connection process by clicking the *Stop Connecting* button.

### Cluster Status

This view is meant to show the total of resources consumed in either your cluster and all the foreign clusters you are connected to. Here are displayed, for both Home and Foreign clusters:

- **Consumption of CPU and RAM**: this donut chart shows you the percentage of resource consumed. In the *Home* cluster section, it shows the user's consumption as well as every other peer consumption. Hovering over each slice of the chart will pop-up a tooltip that let the user know the peer that is using their resources and the percentage of use. In the *Foreign* clusters section, it shows how much resources the user is using on the clusters they are connected to. Hovering over each slice of the chart will pop-up a tooltip that let the user know the peer they are using their resources from and the percentage of use.

- **Consumption trend**: a chart that displays the difference in consumption of CPU and RAM over time. It is updated every 30 seconds.

Unlike the Connection Detail view (which tells all the details about a single connection), the Cluster Status view is a more generic exposure of how and who is using resources on the user's cluster, as well as how much they are using others'.

## 8.2 CrownLabs

CrownLabs is a set of services that can deliver remote computing labs through a per-user virtual machine.

Instructors can provision a set of virtual machines, properly equipped with the software required for a given lab (e.g., compilers, simulation software, etc).

Each student can connect to its own set of (remote) private environments without requiring any additional software, just a simple Web browser. No space problems on the student hard disk, no troubles in setting up the environment required to support multiple subjects on the same machine, and more.

In addition, each student can share his remote desktop with their groupmates, enabling multiple students to complete their labs in a team.

Finally, CrownLabs supports also instructors, who can connect to the remote desktop of the student and play directly with his environment, e.g., in case some help is required.

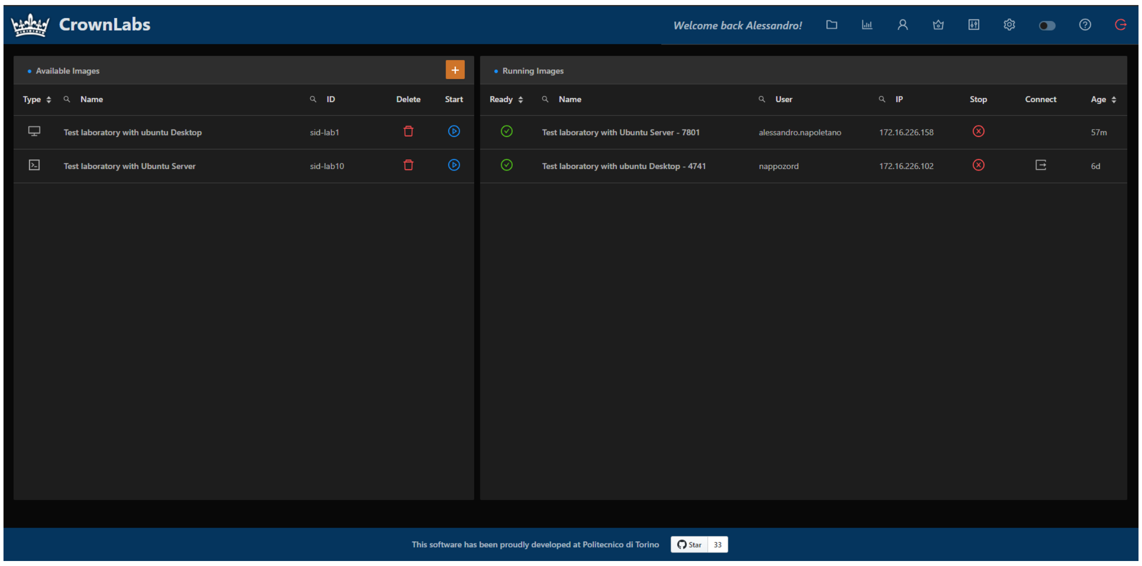
### 8.2.1 Liqo*Dash* meet CrownLabs

While the integration with Liqo is just about the expansion of the functionalities of the dashboard through the implementation of custom views that performs workflows specific for the Liqo system, the integration of CrownLabs is a completely different use case that really demonstrates the high extensibility of Liqo*Dash* and the powerful tools that it offers.

CrownLabs is used by both system admins and generic users (students and professors) which possibly know nothing about how it works and the fact that it is built over Kubernetes. Because of that, the user interface needs to be clean and simple for the latter kind of users, stripped to its basic functionalities, avoiding all the superfluous actions that the dashboard provides by default that would only cause confusion to the end-user. With that in mind, exploiting the modularity of Liqo*Dash* and the possibility to enable or disable each general component, we created two dashboard configuration for the two roles, one that only implements what the end-user could and should interact with, and another configuration that maintains all the features that are useful in managing the cluster.

Because the generic user does not need to explore the cluster, the discovery process has been disabled in the configuration, as well as the sidebar and the possibility to select a namespace or search for a resource. Some useful external links have been added to the header (Drive and Grafana) with the links to the custom views that implements the components specific for CrownLabs. Also a footer with a Github reference to the CrownLabs project has been enabled.

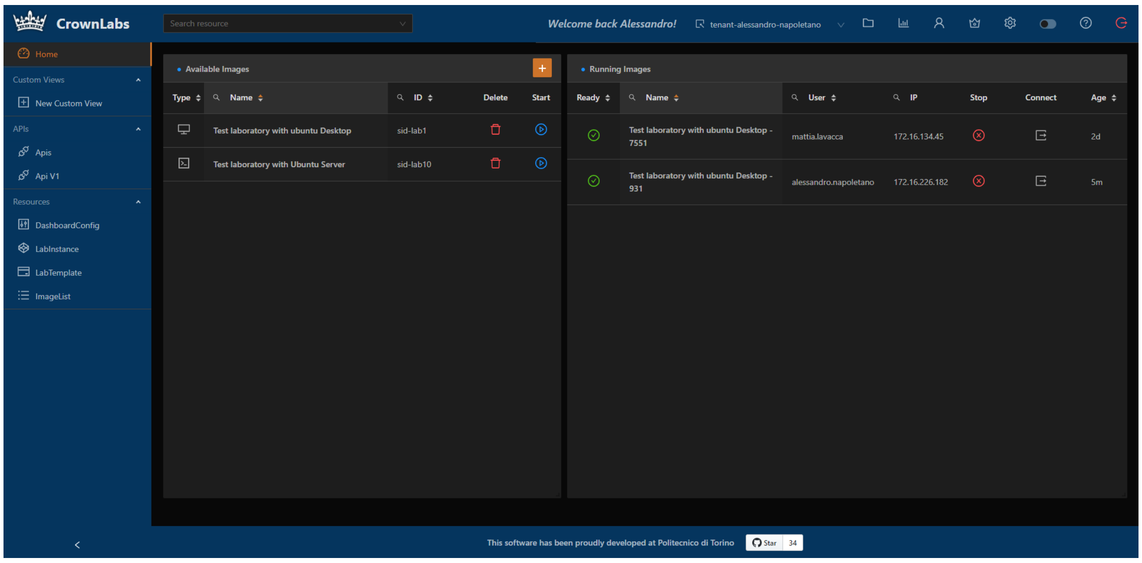
That shows how a single dashboard can act differently with two kind of users. Technically, if every user of the cluster has a role (e.g. different service account or identity to access the dashboard), that means that a different dashboard configuration can be assigned to every user.



| Type | Name | ID | Delete | Start |
|---|---|---|---|---|
| 🖥 | Test laboratory with ubuntu Desktop | sid-lab1 | 🗑 | ▶ |
| ▣ | Test laboratory with Ubuntu Server | sid-lab10 | 🗑 | ▶ |

| Ready | Name | User | IP | Stop | Connect | Age |
|---|---|---|---|---|---|---|
| ✓ | Test laboratory with Ubuntu Server - 7801 | alessandro.napoletano | 172.16.226.158 | ✕ | | 57m |
| ✓ | Test laboratory with ubuntu Desktop - 4741 | nappozord | 172.16.226.102 | ✕ | ⇥ | 6d |

**Figure 8.3:** The CrownLabs view for generic users. Designed to be clean and simple.



| Type | Name | ID | Delete | Start |
|---|---|---|---|---|
| 🖥 | Test laboratory with ubuntu Desktop | sid-lab1 | 🗑 | ▶ |
| ▣ | Test laboratory with Ubuntu Server | sid-lab10 | 🗑 | ▶ |

| Ready | Name | User | IP | Stop | Connect | Age |
|---|---|---|---|---|---|---|
| ✓ | Test laboratory with ubuntu Desktop - 7551 | mattia.lavacca | 172.16.134.45 | ✕ | ⇥ | 2d |
| ✓ | Test laboratory with ubuntu Desktop - 931 | alessandro.napoletano | 172.16.226.182 | ✕ | ⇥ | 5m |

**Figure 8.4:** The CrownLabs view for admin users. Let the admin explore the cluster.

## 8.3 Performance

In this section we analyze the performances of Liqo*Dash* in relation to the other dashboards that represent the state of the art of the Kubernetes web-based user interfaces, described in Chapter 2. The tests were performed using a test Kubernetes cluster environment in which all the dashboards were deployed, expect Octant which is an external program and therefore does not run directly in Kubernetes. In the cluster were also present a considerate amount of custom resources to test the scalability of the **discovery process** of Liqo*Dash* and deployments to simulate a production environment (although a really small one).

For these tests were considered three dashboards other than Liqo*Dash*: **Kubernetes Dashboard**, **k8dash** and **Octant**.

#### Initial Load Time

This first chart represent the average time that the dashboard takes to:

- Load the application: all the process needed to start the dashboard and the creation of the environment (assuming the login is already done);
- Load the page: that means the general structure of the a page, understanding which content is needed to load and the resources to ask the server, as well as all the initial scripting executed before the view rendering;
- Load the resources to show: generating requests to retrieve resources from the Kubernetes API server and waiting for the responses;
- Load and create the DOM content: process data retrieved from the API server and load components to create the views;

Because the home page vary from dashboard to dashboard, all the tests have been performed using a view that is common to all four of them: the Deployment resource view, that shows all the deployments in a namespace of the cluster.

As we can see from the Figure 8.3, the fastest UI to load is k8dash with just less than a second. One of the key feature advertised by the k8dash team is indeed

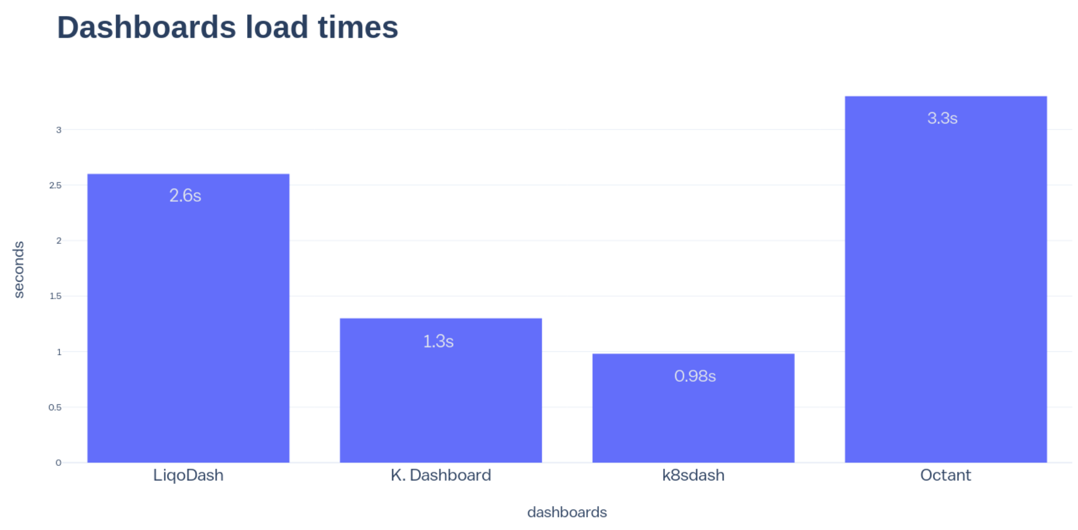


**Figure 8.5:** Comparison of the dashboards' load time.

its speed, and it shows in the benchmarks. The Kubernetes Dashboard follows with 1.3 seconds of overall loading time. It is, however, noteworthy to specify that almost 80% of the loading time of a user interface is about the rendering of the graphic components, and the components of these two first dashboards (tables, menus, lists and buttons) are extremely simple in their visual design and, as such, highly performant speed-wise at the expense of their aesthetics.

Liqo*Dash* needs almost double the time to load compared to the official dashboard. This is partly because of the **discovery process** that starts as soon as the dashboard is accessed and, although it is a background process, because JavaScript is designed as a single-threaded language, this process affect the first loading of the dashboard. Finally, the rendering of the components forming the view is what slow the creation of the DOM contents the most. This is tied to the choice of using **ant design** as design framework which is highly customizable and well designed visually, but it is still a relatively new framework and the performances are not as good as the one given by others design framework such as material-ui.

The slowest one, Octant, takes 3.3 seconds to fully load its initial page.

### API call time

This test was performed switching through various views of the dashboards, and its purpose was to validate the approach that Liqo*Dash* has implemented in regard to the use of a minimal backend instead of the full ones that all the other dashboards have. The chart shows the RTT (round trip time) of an API call from the moment the request has been sent by the dashboard to the moment a response (only positive ones in this case) is received. The four resources chosen for test are:

- Namespaces: a cluster-wide resource;
- Nodes: a cluster-wide resource;
- Deployment (all namespaces): a namespaced resources, but the request is to get all resources of type Deployment in all namespaces;
- Deployment (default namespace): in this case, only the request is to get the resource only in one namespace;

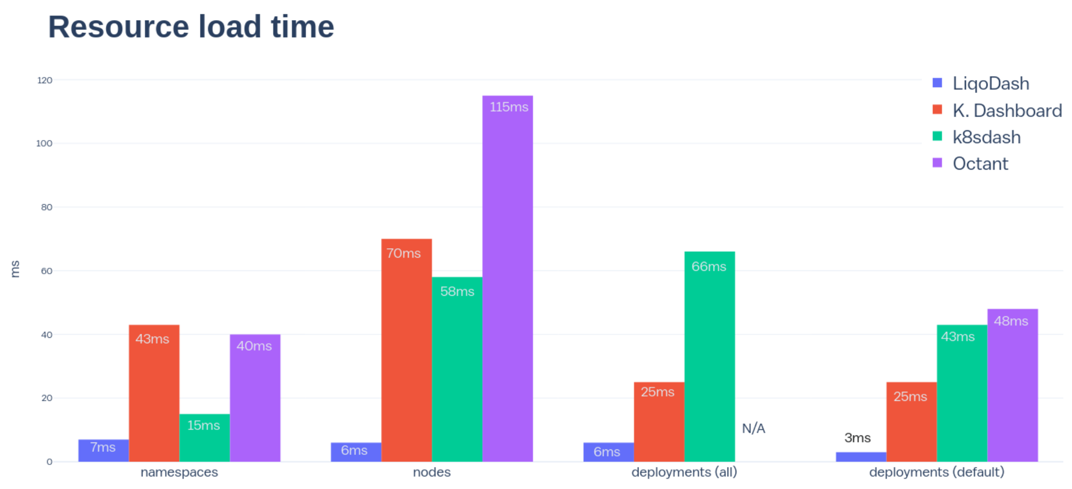


**Figure 8.6:** Load time of four kind of resources. The minimal backend gives Liqo*Dash* a significant advantage.

As we can clearly see in Figure 8.4, the absence of a complex backend that process API calls and forwards them to the API server gives Liqo*Dash* a significant advantage. The data is self-explanatory: even though the resource requested (the

API called) is the same, the response time of the other dashboards is always more than doubled, and in case of the Node resource, the RTT is ten times lower for the Liqo*Dash* than its counterparts.

# Conclusions

The work of this thesis aims at defining a new kind of approach in regards of improving the user experience of a platform as complex as Kubernetes. The focus is not only to give system administrators a suited environment in which they can manage their resources and monitor the cluster with ease, but also, and perhaps most importantly, offering developers that choose Kubernetes as a base for their projects and applications a way to integrate their work in a user interface, which is fundamental when said application has to be used by generic users that are not really accustomed with the powerful but rather complex platform that is Kubernetes.

We have seen how it is possible for the dashboard to accommodate easily every user's needs and how it is designed for simplicity, but at the same time capable of representing complex systems and workflows. It is all in the user's hands and this is one of the quality of this project.

As Kubernetes gains more and more attraction, and its use has slowly shifted from just being a *simple* orchestrator to a real echosystem, a simple dashboard that let the user only watch their resources in a very disconnected fashion is not enough anymore. That is where Liqo*Dash* finds its spot. Like Kubernetes is designed to be a customizable platform in which developers can create their own applications, the dashboard is designed the same way: a platform in which developers can create their frontend for generic users to interact with the applications, and for developers themselves to monitor their resources, workflow and in general, their cluster.

Like every other application, the dashboard is always in constant changing and trying to improve the user experience and the tools it offers its users. As an open-source project, there is always room for improvement.

# Bibliography

1. Stephen Few, Informational Dashboard Design (2006)
2. Octant. URL: `https://octant.dev/`
3. k8dash. URL: `https://github.com/indeedeng/k8dash`
4. Kubernetes Official Dashboard. URL: `https://github.com/kubernetes/dashboard`
5. Kubernetes Official Website: URL: `https://kubernetes.io/`
6. React. URL: `https://reactjs.org/`
7. Ant Design. URL: `https://ant.design/`
8. Kubernetes Client for Javascript. URL: `https://github.com/liqotech/kubernetes-client-javascript`
9. Cross-Origin Resource Sharing. URL: `https://developer.mozilla.org/it/docs/Web/HTTP/CORS`
10. OpenID Connect. URL: `https://openid.net/connect/`
11. Liqo Official Website. URL: `https://liqo.io/`
12. CrownLabs. URL: `https://github.com/netgroup-polito/CrownLabs`