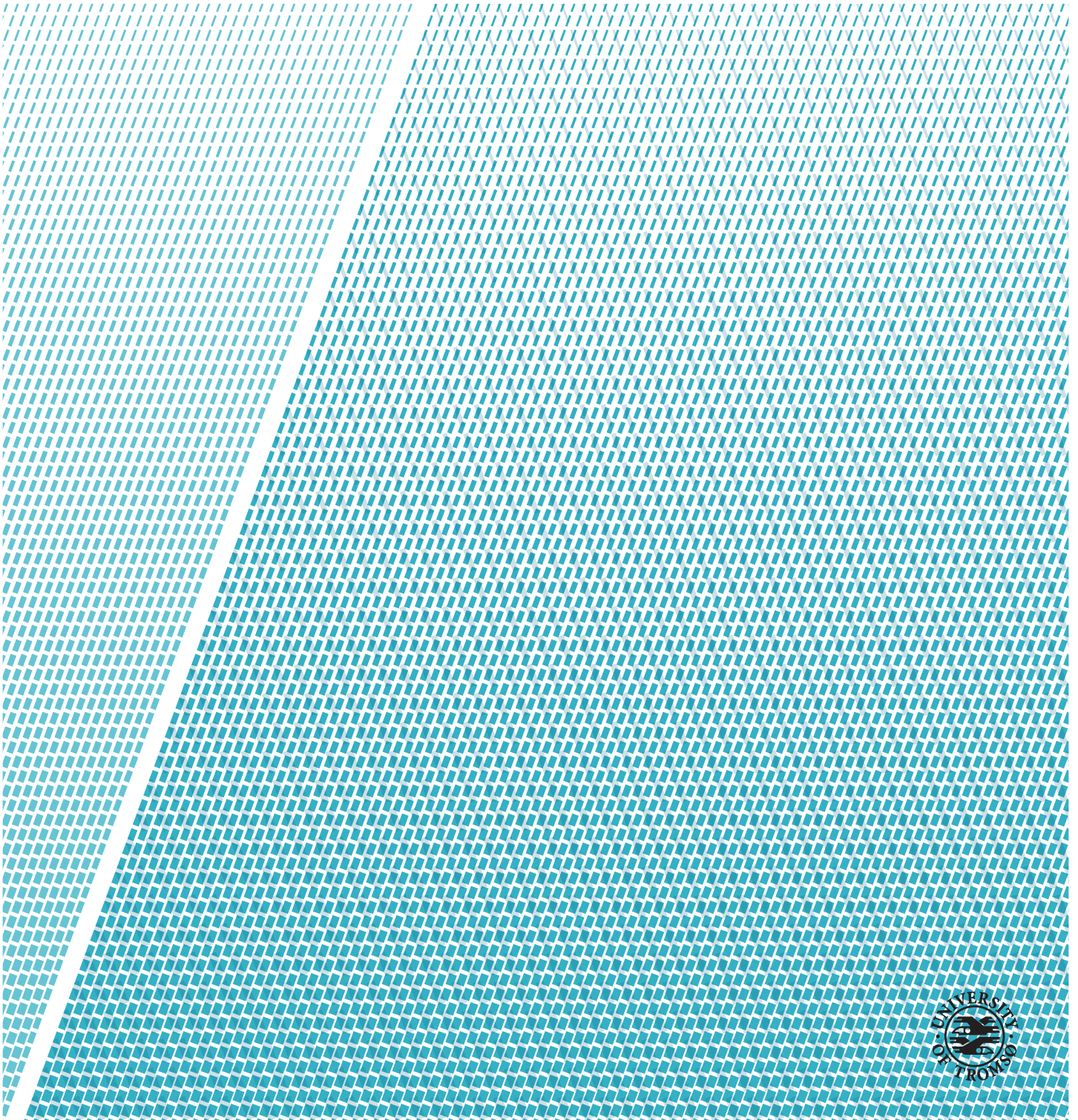


VisualBox

A Generic Data Integration and Visualization Tool

Pontus Edvard Aurdal

INF-3981: Master's thesis in Computer Science – June 2019



“The art and practice of visualizing data is becoming ever more important in
bridging the human-computer gap to mediate analytical insight in a
meaningful way.”
–Edd Dumbill

Abstract

The number of cellular Internet of Things (IoT) connections is expected to grow at a rate of 30% each year and is reaching into the billions by 2019. The world of IoT can be fragmented since data sources span a wide variety of protocols, API's, authentication methods and file formats. Data collection and processing can be complex and producing visualizations for value extraction can be a tedious task. Developers can often find themselves "reinventing the wheel" while building or using visualization libraries that present data in a specific way.

VisualBox is a generic data integration and visualization tool, built as a Software as a Service (SaaS) running a front-end web application and a back-end cloud architecture using Amazon Web Services (AWS). VisualBox is built by first defining an abstract vision where problems are divided into smaller parts that are then progressively developed into a coherent system.

An ecosystem of crowdsourced modules is used to allow developers to write software that handle data fetching and processing (called "integrations") and data visualizations (called "widgets"). These modules can be published to a registry where other users of the system can discover them for use of their own.

Modules can be added to dashboards that produce data visualizations. Integration modules output generic data models that can be connected to widget modules. By making this separation, different widgets can be used to visualize different data models and allows for rapid value extraction, even for users without any technical or programming experience.

Different approaches are explored while solving the problem of executing arbitrary user generated code and how to isolate the host system from code with malicious intent; on the client with the use of web-workers and on the back-end with the use of Docker containers. The container startup time is evaluated while using Amazon Elastic Container Service (Amazon ECS) with the Fargate launch type.

Acknowledgements

First I would like to thank my main advisor, Prof. Anders Andersen for his support, valuable input to this work, contribution of ideas for further exploration and for always providing me with the necessary working spaces and equipment; and my external advisor, Dr. Arne Munch-Ellingsen for providing the initial idea of this work, for introducing me to challenging and highly interesting IoT projects and for acting as a very supporting mentor during the final years of my academic career at UiT.

Further I would like to thank the staff at the Department of Computer Science; Ken-Arne Jensen and Kai-Even Nilssen for helping with hardware and cloud resources crucial for the development of this work; student counsellor Jan Fuglesteg for always making my study progression smooth; and all the other people responsible for a fantastic CS study programme at UiT.

Finally, thank you mom and dad for always supporting me both mentally and financially during my years in the far north and for all the visits you've made.

Contents

Abstract	iii
Acknowledgements	v
List of Figures	xi
List of Tables	xv
List of Abbreviations	xvii
1 Introduction	1
1.1 Problem Statement	2
1.2 Results	3
1.3 Methodology	3
1.3.1 Divide and Conquer	4
1.3.2 Progressively Added Functionality	4
1.4 Scope of Work	5
1.5 Scope Limitation	5
1.6 Outline	6
2 Motivation and Vision	7
2.1 An Easy-to-use Dashboard Builder	8
2.2 Crowdsourced Modules	9
2.3 Motivation and Vision: Summary	10
3 Incremental Exploration: Data Processing	11
3.1 Code Isolation is Important	12
3.2 Client Side Data Processing	13
3.2.1 JavaScript Transpilation	15
3.2.2 Module Resolution	15
3.2.3 Cross-Origin Resource Sharing (CORS)	17
3.3 Server Side Data Processing	18
3.3.1 First Approach: AWS Lambda	19
3.3.2 Second Approach: Amazon ECS Fargate	19

3.3.3	Third Approach: Kubeless	20
3.3.4	Fourth Approach: Fission	20
3.3.5	Final Approach: Kubernetes	20
3.4	Incremental Exploration: Summary	21
4	Vision Realized: Architectural Model	23
4.1	Integration Runtimes	25
4.2	Dashboard Builder	27
4.2.1	Main Panel	28
4.2.2	Adding Integrations	29
4.2.3	Adding Widgets	31
4.2.4	Connecting a Widget to an Integration	32
4.3	Architectural Model: Summary	34
5	Crowdsourcing Developer Model	35
5.1	Shareable Modules	35
5.1.1	Modules are Versioned	36
5.2	Module Development	37
5.2.1	Select Runtime Environment	37
5.2.2	Source Code Editor	38
5.2.3	Upload Module Code	38
5.2.4	Sync with External Version Control Services	38
5.2.5	Preview Module	39
5.3	Configuration Data Model	40
6	Implementation	43
6.1	Infrastructure as Code	43
6.1.1	Serverless Framework	44
6.2	AWS Lambda Functions	45
6.2.1	Creating an HTTP endpoint for an AWS Lambda	45
6.3	VisualBox Cloud Architecture	47
6.4	Launching a Container	49
6.4.1	Initial Parameters for a Container	49
6.4.2	Lambda Task Launcher (LTL)	49
6.4.3	Container Bootstrapper	51
6.4.4	Lambda File Provider (LFP)	51
6.4.5	Container Access Record (CAR)	52
6.4.6	Socket Server	53
6.5	Widget and Integration Indexing	55
6.6	Publishing a Widget or Integration	56
6.7	Implementation: Summary	58
7	Experiment and Evaluation	59
7.1	Container Startup Time	59

7.2	Experimental Setup	60
7.3	Results	61
7.4	Experiment: Summary	64
7.5	Distributed Arctic Observatory (DAO)	64
8	Discussion and Future Work	67
8.1	Multi-stage Docker Image Builds	67
8.2	Warm Containers	68
8.3	gVisor	69
8.4	Securing Sensitive User Data	69
	8.4.1 Initial Configuration Data Model	70
8.5	Sharing Dashboards	70
9	Conclusion	73
	Bibliography	77

List of Figures

1.1	Ericsson Mobility Report, June 2018, p. 16.	2
1.2	An abstract desired result is divided into smaller problems (A, B, C and D) that can be worked on independently.	4
1.3	Functionality to the end result is added progressively as to always keep the current product functioning.	4
2.1	Abstract representation of components involved in a typical IoT solution.	8
2.2	The complete IoT solution can be separated into two independently functional parts; the presentation and data part.	9
3.1	Running integrations must be isolated to prevent a harmful integration (colored red) to interfere with other integrations.	12
3.2	After the web application is sent to the client (1) it will fetch and process data on the client-side (2).	13
3.3	Total number of packages in each registry over time. Generated with http://www.modulecounts.com/	14
3.4	Required integration modules (dependencies) are sent to the Turbo resolver AWS Lambda function. The result is injected into the final bundle.	16
3.5	The same HTTP request done to the origin and a remote server.	17
3.6	The client sends the integration code to the back-end (1) where it is executed (2). The result is then returned to the client (3).	18
4.1	Integrations, data models and widgets together form a dashboard for data visualizations.	24
4.2	Integrations can be executed in different runtimes containers on the back-end and are orchestrated in a Kubernetes cluster.	25
4.3	An existing dashboard can be opened by clicking on it in the dashboard list, or a new dashboard can be created by clicking the plus-button in the top right corner.	27
4.4	The main panel of a dashboard builder lets the user add/remove integrations and widgets.	28

4.5	The integration explorer allows the user to add already existing integrations to their dashboard.	29
4.6	The information page is shown when an integration is opened from the integration explorer. The version can be specified before added to the dashboard.	30
4.7	Added integrations to the dashboard are shown in a list in the main panel. Their color signals if the integration code is running (red or green).	30
4.8	The configuration data model defines input parameters that the integration code can use.	31
4.9	Action buttons are shown when a widget is hovered where it can be edited, copied or removed from the dashboard grid. .	32
4.10	A widget can be connected to a data model (that has been generated by an integration) by clicking the "Data Source" button in the configuration panel.	32
4.11	The data model viewer will display generated data models by integrations that has been added to the dashboard.	33
4.12	A minimal functioning dashboard with a single widget connected to a data model by an integration.	34
5.1	The explorer allows for the discovery of published integrations or widgets which can be forked or used.	36
5.2	A runtime environment must be chosen when a new integration module is created.	37
5.3	A widget can be previewed during development by opening the preview console (right). The configuration data model can be accessed by switching to the configure-tab.	39
5.4	A config.json file has resulted in the following HTML form where the user can input values that are then injected as variables into the widget or integration code. This may alter the appearance and/or behavior of the widget/integration. . . .	41
5.5	The same gauge widget with an unconfigured (top) and a configured (bottom) configuration model.	42
6.1	Architectural overview of the VisualBox back-end composed of multiple services and AWS cloud resources.	47
6.2	Steps involved for a client to launch one or more integrations in separate containers and establish a private socket communication channel.	50
6.3	The container lifecycle is governed by the bootstrap binary which is running in each container.	51
6.4	Steps involved for a container to get its source code from the Lambda File Provider (LFP).	52

6.5	The development console with thee different messages; T_INFO (green), T_WARNING (yellow) and OUTPUT (blue).	55
6.6	Workflow when a client wants to publish a new version of a module.	57
7.1	AWS ECS Fargate startup times for containers with different image sizes using a task definition with 0.5GB memory and 0.25 vCPU.	61
7.2	AWS ECS Fargate startup times for containers with different image sizes using a task definition with 1GB memory and 0.5 vCPU.	62
7.3	AWS ECS Fargate startup times for containers with different image sizes using a task definition with 2GB memory and 1 vCPU.	63
7.4	VisualBox in use by the Distributed Arctic Observatory (DAO) research group. The middle screen is a VisualBox dashboard visualizing metrics reported by two separate IoT devices. . .	65
8.1	A socket communication is the only link between the client web application and the VisualBox back-end Kubernetes cluster, and hides away any sensitive information that may have been used while generating data.	71

List of Tables

4.1	Prepare and run commands that are executed in respective runtime after launch.	26
6.1	Initial parameters for a container.	49
6.2	Socket Message Types sent between the client application and containers.	53
6.3	Possible Status Types for the STATUS message.	54
7.1	Image sizes used in the experiments.	60
7.2	AWS ECS Fargate task definition configurations.	60

List of Abbreviations

AWS Amazon Web Services

CAR Container Access Record

CD Continuous Deployment

CORS Cross-Origin Resource Sharing

CRUD create, read, update and delete

DAO Distributed Arctic Observatory

DoS Denial of Service attack

EC2 Elastic Compute Cloud

ECR Elastic Container Registry

ECS Elastic Container Service

FaaS Function as a Service

IaC Infrastructure as Code

IAM Identity and Access Management

IDE Integrated Development Environment

IoT Internet of Things

IST Instance Session Token

LFP Lambda File Provider

- LTl** Lambda Task Launcher
- MIC** Managed IoT Cloud
- MVP** Minimum Viable Product
- NPM** Node Package Manager
- PWA** Progressive Web App
- REPL** read–eval–print loop
- S3** Simple Storage Service
- SaaS** Software as a Service
- SemVer** Semantic Versioning Specification
- SPA** Single Page Application
- UiT** University of Tromsø
- VM** Virtual Machine



Introduction

The Internet of Things (IoT) is on the rise and more and more data is generated by ubiquitous sensors located all around us. Ericsson is forecasting the number of cellular IOT connections to grow at a rate of 30 percent each year, and reach 3.5 billion in 2023 [1]. By connecting physical things to the internet, industries can reduce production costs and increase the production efficiency as more monitoring data becomes available.

With the introduction of network technologies such as LoRaWAN and NB-IoT¹ – small battery driven microcontrollers can extend their life-time to span years [2] and the cost of production is going down. As network infrastructure is upgraded, extended and a higher bandwidth can be established, data pipes can be streamlined to push information directly to end-consumers which enables a constant stream of updated live data. Nevertheless, with all of this comes increasing amounts of data.

1. Long Range Wide Area Network (LoRaWAN) is maintained by the LoRa Alliance.
Narrowband IoT (NB-IoT) is developed by 3rd Generation Partnership Project (3GPP).

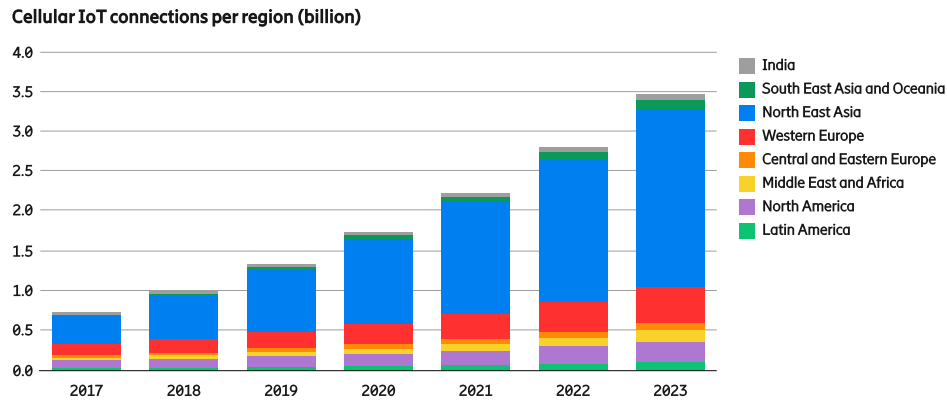


Figure 1.1: Ericsson Mobility Report, June 2018, p. 16.

Data visualization has long been an application for computer systems and data scientists. Data often needs to be pre-processed before actual value can be extracted. As humans, we can get a better understanding of larger quantities of data by transforming, aggregating, splitting it into smaller pieces and visualize it. As such, you can find graphical data visualization libraries in many programming languages. Although the tools to create data visualizations exists, it can often be a tedious task to write the software necessary to achieve this goal, especially if data sources span a range of different protocols, devices, services and authentication methods. This is especially true in the world of IOT.

Related work, such as Grafana [3], come in the form of bundled services that are downloaded, installed and hosted on infrastructure provided in-house by the customer and are often tailored for a specific application use-case.

1.1 Problem Statement

A tool for creating data visualizations as a collection of graphs in a dashboard would reduce time spent on building IOT prototypes and applications.

The world of IOT can be fragmented in terms of data sources that span different protocols, devices etc. It can be a tedious task to build a system which integrates and collects data from various data sources. There are also only so many ways e.g. a two-dimensional graph can be made, so a lot of time could potentially be wasted on implementing the same types of graphs for each new data visualization application.

This thesis will handle how to safely execute arbitrary user generated code in a containerized environment, explore different container orchestration methods and how to tie together a coherent system where data visualizations can be made from crowdsourced modules that can be arranged in dashboards. The use of containers is motivated by the execution isolation it provides, and will be used to run crowdsourced module code. The container efficiency, in terms of startup time, will be evaluated and different levels of execution isolation will be explored on the 1. browser level in the form of utilizing web workers and 2. the back-end in the form of a cluster of containers.

1.2 Results

The result of this thesis is a complete Software as a Service (saas) solution for a generic data visualization tool called *VisualBox* (<https://visualbox.io>) where virtually any data source can be attached. To achieve flexibility for the user to attach data sources, an experimental cluster of containers is used to host a pool of sandboxes to run arbitrary user generated code in isolation. Containers support a range of environments where different programming languages can be used. Widgets can be arranged in a highly configurable dashboard which gives maximum customizability.

Crowdsourced data source integrations and visualization widgets allows for a complete data presentation application to be built in a short time without any technical competence. A model for how to optimize containers by using a pre-build step is proposed and how it compares against the implemented model. The feasibility of such a system is tested in cooperation with research projects at the UiT.

1.3 Methodology

The methodology used in the development of this project is based on a divide-and-conquer approach for problem solving combined with progressively added functionality.

1.3.1 Divide and Conquer

When working towards an abstract vision for what the desired end result should be it can be helpful to divide it into smaller, more digestible problems. This method is widely used during the thesis work to not only concretize problems, but later compose a larger functioning end result.

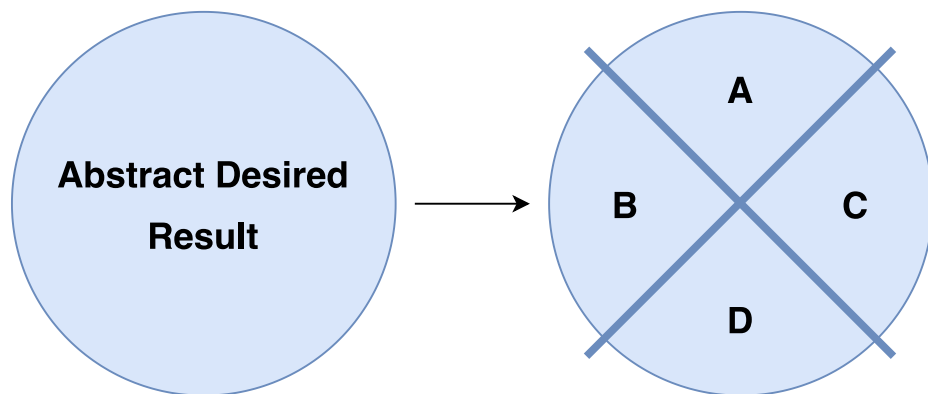


Figure 1.2: An abstract desired result is divided into smaller problems (A, B, C and D) that can be worked on independently.

1.3.2 Progressively Added Functionality

The project start with a working solid base foundation. As new problems are solved they are added to the base as an addition. The goal is to always have a working product at any given time. This is somewhat comparable to what is found in Agile methodologies where instead of a waterfall approach the product should always be in a working state (although possibly lacking functionality).

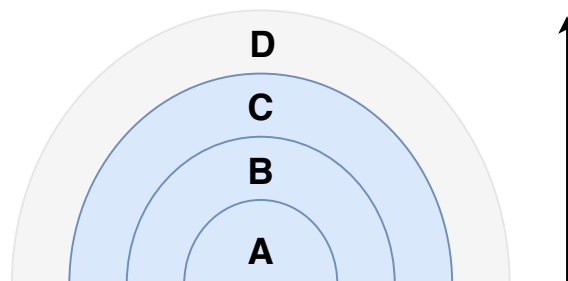


Figure 1.3: Functionality to the end result is added progressively as to always keep the current product functioning.

1.4 Scope of Work

This thesis focuses on the integration of different types of data sources across protocols and authentication methods, and how to efficiently and safely create visualizations in generic dashboards with crowdsourced modules such as:

- **Data source integrations** that fetch, process and generate data models, written by integration developers in their programming language of choice.
- **Widget visualizations** that consume data models generated by data source integrations and display the result in dashboards running in a web browser.

One of the first problems to solve is to find out how and where these crowdsourced modules will be executed. In the case of integrations that are responsible for data processing, it can either be executed on the front-end or the back-end. Both alternatives must be tested to find out which is the best alternative.

A client web-application with a dashboard builder must be implemented so that data visualization modules can be connected to data processing modules. A platform for crowdsourced developers to build and share their modules and a way for users to find these modules will be built using Amazon Web Services (AWS) as a cloud provider for the necessary infrastructure.

1.5 Scope Limitation

This thesis will not address the implementation details of the front-end application. The source code for the front-end application is open sourced and can be found at the following GitHub repository: <https://github.com/Pwntus/visualbox-frontend>.

1.6 Outline

Chapter 2 – Motivation and Vision explains the motivation behind the work of this thesis and describes the vision for the system.

Chapter 3 – Incremental Exploration describes paths taken while finding a solution for optimal data processing.

Chapter 4 – Vision Realized: Architectural Model outlines components of the interactive visualization dashboard builder model.

Chapter 5 – Crowdsourced Developer Model describes components of the crowdsourced module builder and the developer experience.

Chapter 6 – Implementation describes in detail the cloud architecture and the functionality behind the service.

Chapter 7 – Experiment and Evaluation outlines the experiments performed and obtained results while finding the fastest container startup time for different container orchestrating strategies and shows how third parties can utilize a minimum viable product of the VisualBox system.

Chapter 8 – Discussion and Future Work motivates choices taken and suggests future work and areas of improvements.

Chapter 9 – Conclusion concludes this thesis.

/2

Motivation and Vision

Common components involved when developing a complete IoT-like solution includes physical sensors powered by battery-driven microcontrollers, a data transfer layer and a back-end data storage service. In addition to these components a data presentation application to visualize processed information to end-users is also common (**Figure 2.1**).

During my work at Telenor Start IoT [4] we made various IoT device prototypes utilizing the LoRaWAN network and the Managed IoT Cloud (MIC) [5], developed by Telenor Connexion. Applications included soil sensors for golf courses and road monitoring sensors to alert weather conditions, all of which were deployed in and around the city of Tromsø in northern Norway. What we found was that for each new prototype came a new application, and a new "dashboard" had to be made to visualize the generated data.

It soon became a routine to setup everything necessary ranging from authentication, building a grid of graphs and connect them to the data source. Although each application had similarities in how data was fetched, they all varied in how the data was presented.

My fellow colleagues could also see the value in a tool for data visualizations that wouldn't care where the data is coming from or how it is stored, at least in a rapid prototyping phase of a project.

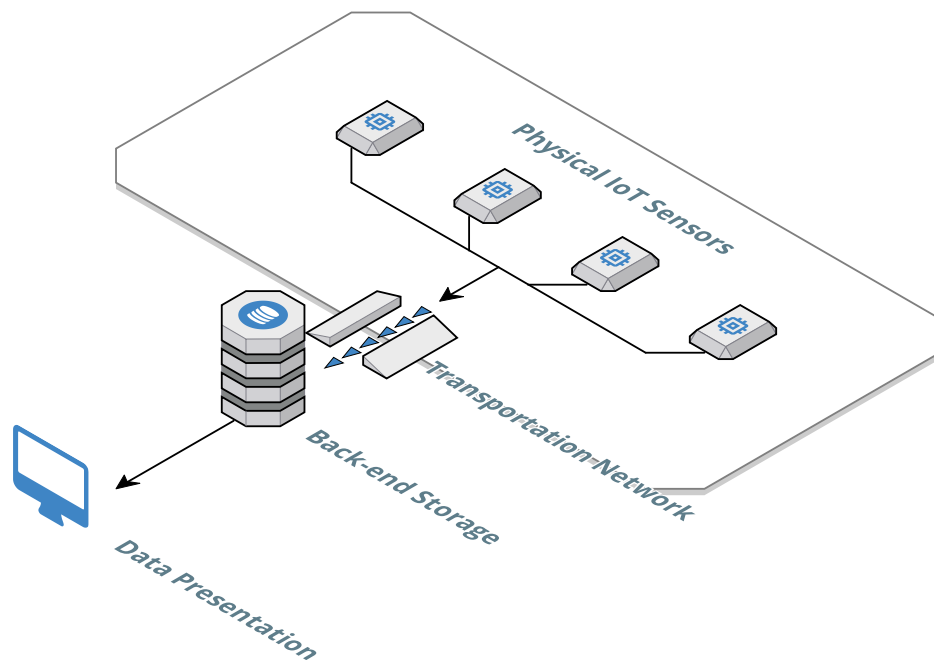


Figure 2.1: Abstract representation of components involved in a typical IoT solution.

A generic dashboard builder could solve all steps involved in an IoT solution while still allowing full control over how data is fetched and processed. By making the tool into a saas, installation and configurations steps would completely be removed and the client application could be run on a wide variety of devices and operating systems.

2.1 An Easy-to-use Dashboard Builder

If the goal is to visualize data, what if the task of putting together all components necessary to build the infrastructure for a complete IoT solution could be as easy as a drag-and-drop dashboard builder? And not just for an IoT solution, but with any kind of data stored in any kind of way?

A generic dashboard builder for setting up a grid of data visualizations with live updated data can be notoriously hard to make, and is often seen as a custom developed solution for a specific purpose. If this could be made into a saas where anybody, even without any technical skills, could start processing information from a data source and make beautiful data visualizations, it could bring great value to the user.

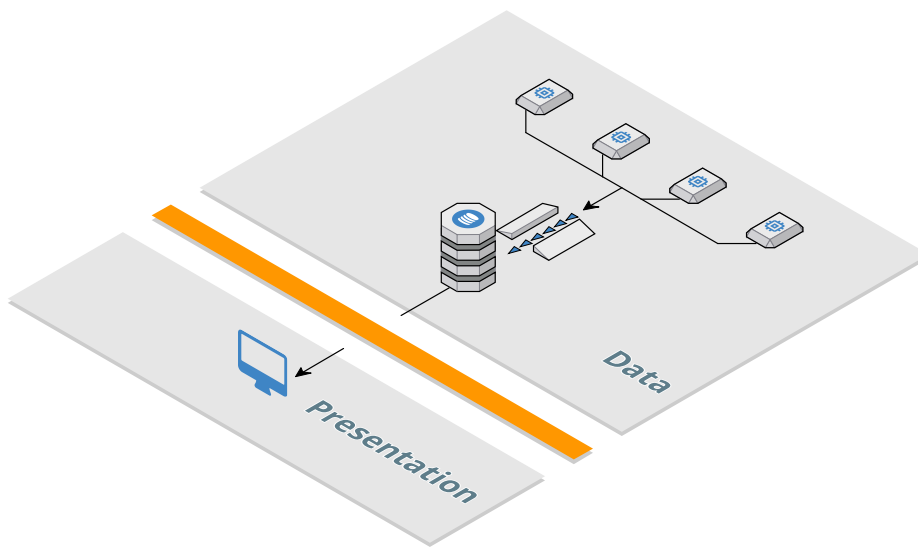


Figure 2.2: The complete IoT solution can be separated into two independently functional parts; the presentation and data part.

2.2 Crowdsourced Modules

What if we separate the abstract representation of components in a typical IoT solution into two parts; the *presentation part* and *data part* (**Figure 2.2**). These are essentially the two parts that can vary independently, and could give great flexibility if combined in different ways (different presentations for different data parts).

If we instead call the presentation part a *widget*, and the data part an *integration*, we can assign them the following roles:

- Integrations are responsible for fetching data, processing it and outputting a *data model* from a data source. They run the logic behind a dashboard and provide the system with data to be visualized.
- Widgets are responsible for actually visualizing data. They are connected to a *data model* (produced by an integration) and simply visualize it in a specific way.

We can draw similarities to the three-tier architecture, which is a software architecture pattern consisting of a presentation tier, a logic tier and a data tier. In this case the integration part would be a combination of the logic tier and the data tier.

Widgets and integrations can now be considered modules in an ecosystem. If someone made a highly generic widget, which is simple in its function but can be applied to visualize a wide variety of data, wouldn't it be useful if it could be shared? By crowdsourcing widget and integration modules, a large collection of pre-built modules ready to be used could greatly benefit other users of the system without the need to have any technical or programming skills.

Not only would crowdsourced modules prevent "the reinvention of the wheel", but could potentially open up for external contributions by third-party members that would improve or modify the look and feel of modules. Crowdsourcing integration and widget modules would be a powerful asset to a data visualization tool.

2.3 Motivation and Vision: Summary

It can be a tedious task to create a data presentation application in a typical IOT scenario. Especially if a project is in a prototyping phase, where quick data visualizations without excessive amounts of time spent on building an application is crucial.

A tool for connecting a variety of data sources that span different data storage methods, protocols and devices together, combined with presentational widgets to form data visualizations, will make it easier and faster to develop IOT applications.

The visualization and data processing parts can be separated into modules that can be combined in different ways. By crowdsourcing these modules, one user can benefit from the work of another user.

Based on this vision, a platform for module development and sharing will be built in addition to a client web-application with a "dashboard-builder". The dashboard builder will be used to combine visualization modules with data processing modules and can be used by users without any technical or programming skills. The first problem to solve then becomes; how and where should data processing modules run in such a system?

/ 3

Incremental Exploration: Data Processing

This chapter will incrementally explore the problem of how and where the data processing part of a dashboard builder could be solved. A trial-and-error approach is applied where different solutions are implemented to find the best option for its task.

The data processing parts of a dashboard will be called *an integration*. If integrations are to be crowdsourced they ought to be programmable by the user. This means that the service must protect other users that utilize crowdsourced integrations. The immediate danger from utilizing code that is not written by yourself is that the code may have harmful intent or unintended behavior (e.g. disastrous bugs with information leaks). This means that the code must be executed in isolation as to prevent one integration to interfere with other integrations or the host system where the integration is executed.

3.1 Code Isolation is Important

Integration code is arbitrary user-generated code and should be treated as being malicious. Great care must therefore be taken when code not written by a host program is to be evaluated/executed. A typical approach in such situations is to create a *sandbox*.

A sandbox environment encapsulates the running code in such a way that no matter what the sandboxed program does, it is contained within set limits of the host. This both protects the system in which the sandbox is hosted and prevents the executed code from using too much compute resources.

Situations where a sandbox environment is required depends upon the application. A sandbox gives more control to the user of the system to write arbitrary custom code that can be used to enhance the overall experience. However, if a severe bug is introduced (intended or unintended) it is up to the host of the sandbox to mitigate the effects of the bug from impacting other components of the host system.

VisualBox must allow integration developers to write their own data processing software, but at the same time protect the internal system. Multiple integrations may run on the same host, so sandbox isolation is important to protect multiple integrations from each other (**Figure 3.1**).

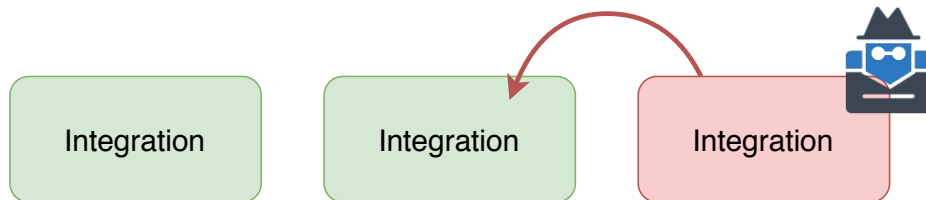


Figure 3.1: Running integrations must be isolated to prevent a harmful integration (colored red) to interfere with other integrations.

The resource utilization of an integration should also be controllable by the service so that a potential Denial of Service attack (DOS) can be mitigated. Ideally, an integration should be executed in a contained sandbox environment with restricted compute resources.

Two approaches are explored in finding the best way to run integration code. The first approach is based on running all integration code on the client-side in the context of web-workers. The second approach abandon the idea to run integration code on the client-side and moves it to the back-end.

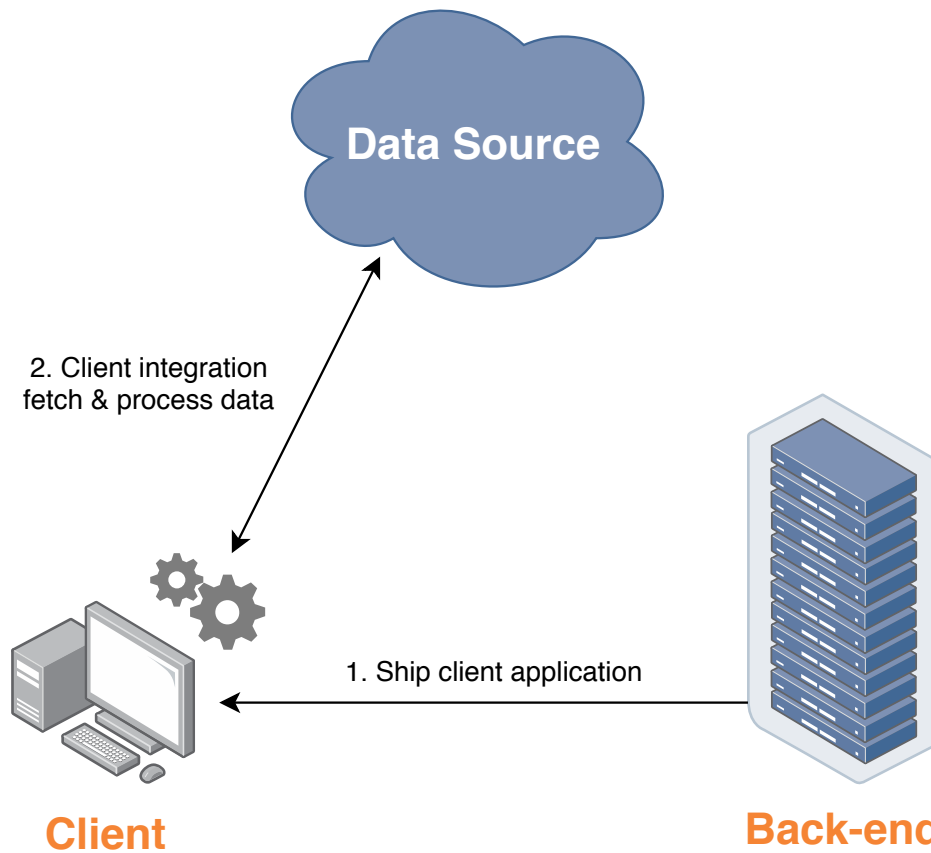


Figure 3.2: After the web application is sent to the client (1) it will fetch and process data on the client-side (2).

3.2 Client Side Data Processing

The design that was first implemented supported integrations written in JavaScript. The decision to only support JavaScript is natural because it's natively supported by all major web browsers. This also allows the user to completely run integration code in a web-worker and offloads the service of all computation involved in generating data models. All code would be executed directly on the client computer (**Figure 3.2**).

An effort to allow integrations utilize the Node Package Manager (NPM) [6] registry to load external JavaScript modules from a public registry was made. NPM is currently the largest package registry and with more than 800 000 packages (as of May 9, 2019) has far more packages than any other major package registry (**Figure 3.3**).

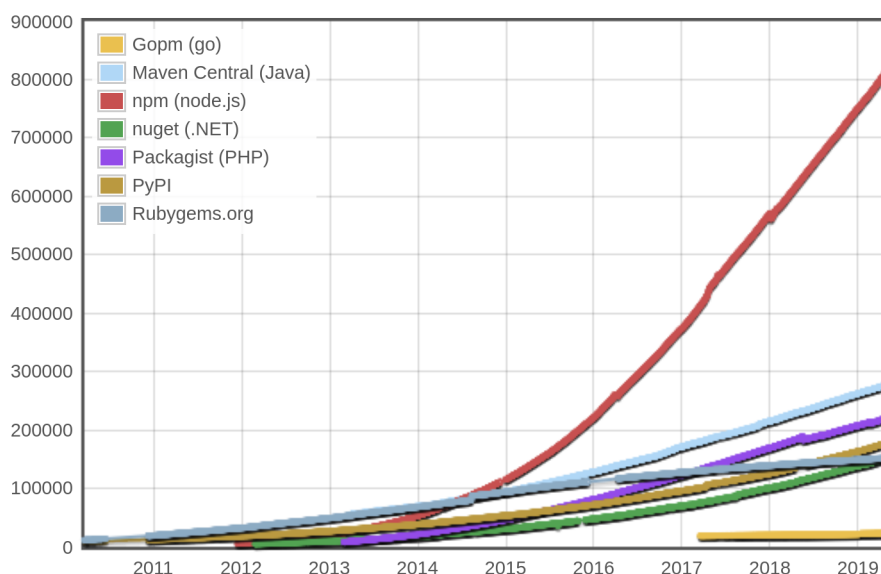


Figure 3.3: Total number of packages in each registry over time. Generated with <http://www.modulecounts.com/>.

However, most packages on the NPM registry is intended to work with the JavaScript runtime called Node.js [7], which is required to run on an operating system platform. External Node.js packages are also not guaranteed to be written in a JavaScript syntax specification that works in a web browser, and require a *transpilation* step before delivered to the web browser.

Running the Node.js runtime in a web browser environment would require mocking of native system calls such as file system operations. The overhead is also large due to the client application being forced to load the entire Node.js runtime code before running the JavaScript code. Including the Node.js runtime in the web browser was therefore not an alternative.

The goal was now to allow purely JavaScript based integrations to write the most recent JavaScript language syntax specification without using the Node.js runtime. Integration code should also not be restricted to only one file. A multi-file project should be able to import files from different folders using the `require()`-syntax in Node.js. If integration code were to be executed inside a web-worker, all code would be in-memory and would be forced to implement a virtual file system to mock files and folders.

In order to allow integration code to utilize the NPM registry, and allow external files to be imported, a module resolution, transpilation and bundle step had to be involved.

3.2.1 JavaScript Transpilation

The JavaScript language specification is standardized by Ecma International [8], and is actually called ECMAScript (short ES). As new ECMAScript standards get finalized, web browsers start to implement features to adhere to the new standard specifications. The standard is drafted and finalized faster than most major web browsers can keep up with, which ends up with a finalized scripting specification that is not always supported in client web browsers.

To mitigate this effect, JavaScript code can undergo a transpilation step, where code written in a newer standard specification gets transformed to target a previous standard that is more widely supported by older web browsers. This process happens offline and often before a bundle step, where assets and files get built into a single bundle that can be shipped to a production server.

3.2.2 Module Resolution

Many online services provide a Node.js-like read–eval–print loop (REPL) environment. Some of them includes Codesandbox.io [9], StackBlitz [10] and Scrimba [11]. They all allow a user to write JavaScript in a Node.js environment where modules are resolved and downloaded, but do it in different ways.

- StackBlitz recognized that modules residing on the NPM registry has a lot of unused code. Code that is never used after the module has been transpiled. They developed a back-end module resolver, called Turbo [12], which claims to be 5x faster than NPM. By analyzing a requested package, and only returning necessary files they can both lower the package size and network transfer time.
- Codesandbox.io started with a back-end bundler, where the packages would be resolved and the project would be transpiled and bundled entirely by servers. Due to the solution not being efficient enough they later moved to a client-side module resolver, much like StackBlitz.¹
- Scrimba has tried to replicate the behavior of StackBlitz, and does a similar job of resolving modules on the client side.²

Since StackBlitz has open sourced the Turbo resolver, an effort was made to

1. Hackernoon article by Ives van Hoorne: "How we make npm packages work in the browser" <https://hackernoon.com/how-we-make-npm-packages-work-in-the-browser-announcing-the-new-packager-6ce16aa4cee6>
2. Scrimba lesson by Magnus Holm: "How we run NPM packages in the browser" <https://scrimba.com/c/c6azJtG>

implement the Turbo resolver in an AWS Lambda function [13]. AWS Lambda functions are used throughout the VisualBox system and will be described in more detail in **Section 6.2 – AWS Lambda Functions**.

When a VisualBox integration requires a module from the NPM registry, a request is made to the Turbo resolver Lambda function and the resolved module code is returned. The module code is then dynamically injected into the integration code.

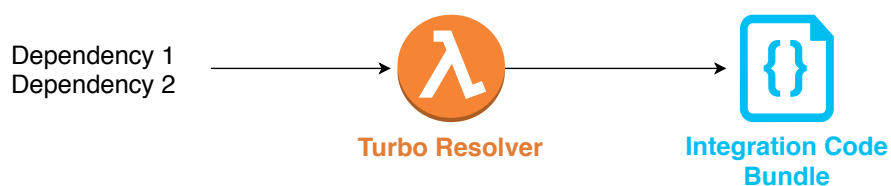


Figure 3.4: Required integration modules (dependencies) are sent to the Turbo resolver AWS Lambda function. The result is injected into the final bundle.

Modules can now be required and resolved but the project code still has to be bundled to work in a browser environment. Rollup [14] is a JavaScript bundler which is small enough to be included in the front-end client application. Rollup was therefore embedded into the client application and was used to bundle files (represented as text strings in memory) into a final integration bundle. The whole process was very primitive and it was apparent that this solution would not scale to support larger files.

When integration code started to function and successfully executed in web workers, a new issue arose. It is very important for integration code to request and download data from external servers. This is one of the core responsibilities of integrations; to fetch and process data. Web browsers implement a security mechanism when client JavaScript requests resources located on a remote server, as will be explained in the next section.

3.2.3 Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) [15] is a security mechanism in web browsers that uses HTTP headers to determine if an application on one domain (the origin) can request resources hosted on a different domain. Essentially, if the requested server in a cross-origin request doesn't return the *Access-Control-Allow-Origin* header granting the origin access, the HTTP request will be blocked (Figure 3.5).

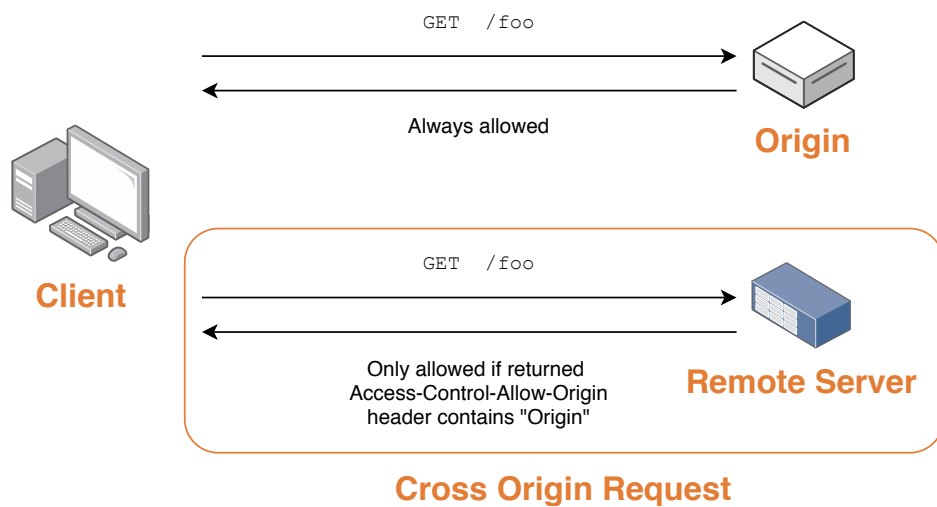


Figure 3.5: The same HTTP request done to the origin and a remote server.

This is a major problem for JavaScript code running in a web-worker. Integration JavaScript code must be able to make HTTP requests to servers that are not hosted on the same domain. Since CORS is not guaranteed to always allow this, a proxy had to be made to bypass this restriction.

The CORS proxy was designed to take an HTTP request as input and forward it, essentially making the HTTP request for the integration itself. This imposes many problems. The proxy becomes a major bottleneck in the system. It also is a single point of failure would it go down at some point. It was also discovered that automated bots by third-parties would use the proxy if the proxy endpoint was discovered.

To mitigate third-parties exploiting the proxy, an authentication mechanism would have to be implemented. All of these issues, and the fact that the integration code would still be forced to always use the proxy were it to make an HTTP call, contributed to the discontinuation of the execution of integration code on the client side.

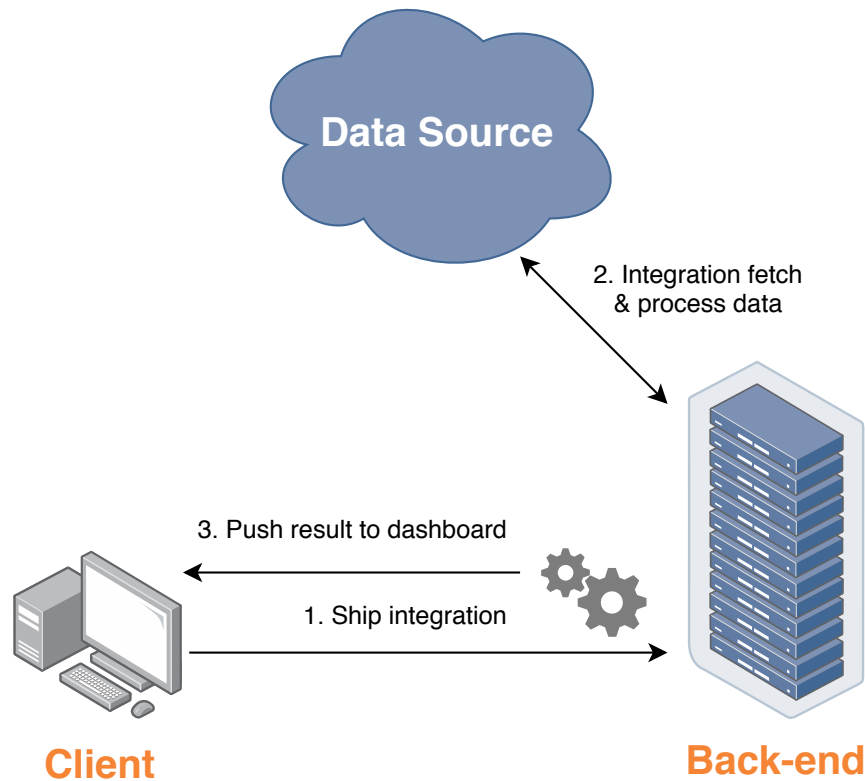


Figure 3.6: The client sends the integration code to the back-end (1) where it is executed (2). The result is then returned to the client (3).

3.3 Server Side Data Processing

Running bundled and transpiled code in web-workers on the client was still too limited. File system operations needed to be mocked and system calls are unavailable. E.g. if an integration needs to fetch a video file from a remote server and transcode it into another video file-format, the code needs to save it to a file-system both before and after the transcoding happens. Even though these functionalities could be mocked or mimicked in a web browser environment (such as `videoconverter.js` [16]), the developer would still be limited to only write JavaScript code.

The decision to move integration execution to the back-end was made (**Figure 3.6**). By running integration code in containers, the possibility to use any language becomes available, and true file system operations and system calls can be made. Containers also provide an isolated environment for integrations. However, there are many different ways to launch and orchestrate containers.

The following sub-sections outline the different approaches that was taken to find the best container orchestration strategy.

3.3.1 First Approach: AWS Lambda

The first logical approach was to see if AWS Lambda could be used to run integration code. Six different runtimes are available (Node.js, Python, Ruby, Java, Go and .NET) and code can be dynamically loaded and executed. However, there is a hard upper limit on the execution time of an AWS Lambda function set to 15 minutes.³

The limit of 15 minutes could not be accepted. A common pattern for data fetching and processing is to fetch the data periodically, pause and re-fetch again after a defined time. This pattern could potentially span days, weeks or months and essentially resembles a computer-server. Since integration code must be able to run for longer than 15 minutes, AWS Lambda was not an alternative.

3.3.2 Second Approach: Amazon ECS Fargate

AWS ECS [17] is a managed Docker container orchestration service for containerized applications running on AWS. By utilizing Elastic Compute Cloud (EC2) [18] worker nodes, a cluster can easily be created for containers to run on. The quantity of worker nodes, number of vCPU's and memory has to be manually managed.⁴

AWS Fargate [19] is a new way of launching containers where cluster management is completely removed from the developers' perspective. All that is required by the developer is to specify a *task definition* (which defines container resources and network interfaces), launch the container and AWS will take care of the rest. AWS will automatically scale the cluster in which containers are launched and load balance according to their internal infrastructure.

AWS Fargate is advantageous in the sense that no cluster management is involved. This can save valuable time and reduces complexity for customers that need to containerize their workloads.

3. "AWS Lambda enables functions that can run up to 15 minutes"
<https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/>

4. vCPU - virtual CPU is a portion of a physical CPU in a virtual machine.

The service characteristics of AWS ECS with Fargate is a good fit for launching containers that run integration code in VisualBox, where multiple containers needs to be launched at different times. However, the container startup time was not sufficient as described in **Chapter 7 – Experiment and Evaluation**.

3.3.3 Third Approach: Kubeless

The decision to move from a tailored and managed service such as AWS Lambda and AWS ECS to a bare container orchestrating service was now made. All major cloud providers offer a managed container cluster solution. Instead of relying on a cloud provider during this experimental phase of the project, where potentially unnecessary cost could get accumulated, the decision to run container orchestration on a local development server was now made. A *Lenovo ThinkCentre M900 Tiny* was acquired. This allowed for rapid prototyping and development without the fear of over-spending. If a good solution was found on this local server it could easily be transferred to a managed service in a cloud provider at a later stage.

Kubeless [20] is an open source Function as a Service (Faas) running on Kubernetes [21] that aims to replicate the functionality of services such as AWS Lambda. The idea was to tweak the weaknesses of AWS Lambda and to remove the code execution limit. However, the customizability of functions were not enough. Kubeless functions are based on a request-response model where a return value is expected to be the result of a function. VisualBox requires that an integration can be long running and return multiple values at different times.

3.3.4 Fourth Approach: Fission

Fission [22] is an open source Faas similar to Kubeless. Fission also runs on Kubernetes. Difficulty in customizing how functions run and return values (much like the previously tried solution, Kubeless) also lead to it not being used in the VisualBox implementation.

3.3.5 Final Approach: Kubernetes

Many container orchestration services has now been tried without finding an optimal way of launching containers for integration code to run in. The final approach was now made to run Kubernetes without any external software. By running Kubernetes without any third-party system gives maximum customizability for how containers are orchestrated. To fully customize the behavior

and management of containers in the Kubernetes cluster, a container bootstrapper was implemented. The bootstrapper handle container initialization and communication with the VisualBox system and will be described in more detail in **Section 6.4.3 – Container Bootstrapper**.

For testing purposes, the Lenovo ThinkCentre M900 Tiny served as a single node Kubernetes cluster where containers could be launched and run in isolation. Once again, this single-node cluster can easily be transferred to a managed Kubernetes service in a cloud provider if the cluster needs to be scaled up to support more containers. An AWS Lambda function can create the definition for a Kubernetes Job⁵ – authenticate with the cluster by using X509 client certificates and launch the job (with a Docker image definition) in the single-node cluster. The process of launching a container from the VisualBox service will be described in **Section 6.4 – Launching a Container**.

3.4 Incremental Exploration: Summary

The first approach of running arbitrary user generated code was to execute it on the client side in web-workers. There were multiple issues with allowing Node.js-style code to be transpiled and bundled in the browser (no true file-system, code transpilation and external Node.js-module resolution complexity, HTTP requests getting blocked by CORS) so the decision to move code execution to the back-end was made.

Different container orchestrating strategies were explored such as AWS Lambda, AWS ECS with Fargate, Kubeless and Fission. None of the tried solutions supported a containerized application to function as a traditional server application, so a plain Kubernetes implementation on a single-node local machine was used.

A container bootstrapper and controller program is included in every container so that it can be managed, limited and allowed to communicate with the VisualBox system.

5. Kubernetes API Reference Docs: "Job v1 batch"
<https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.10/#job-v1-batch>

/4

Vision Realized: Architectural Model

This chapter will describe the architectural model of the system and how the previously described vision has been realized. This chapter takes the perspective of how a user without any technical expertise would experience the VisualBox system.

The VisualBox system is designed to be a Software as a Service (SaaS) with a client running in the web browser. By providing a web client the user is completely relieved of software installation and system configuration. Data processing is offloaded to more powerful computers running on the back-end by the service in a Kubernetes cluster.

The client application is developed as a Single Page Application (SPA) using the Vue.js [23] JavaScript framework. A SPA is a JavaScript web application where the complete application is loaded at once in a single HTTP request. Pages and information is later dynamically loaded when the user interacts with the application. SPA's give a more "fluid" user experience in the sense that the web page never has to be reloaded once launched. A SPA is also easily extended to become a Progressive Web App (PWA), where the whole web application can be added to the home-screen of a smartphone, much like a native app.

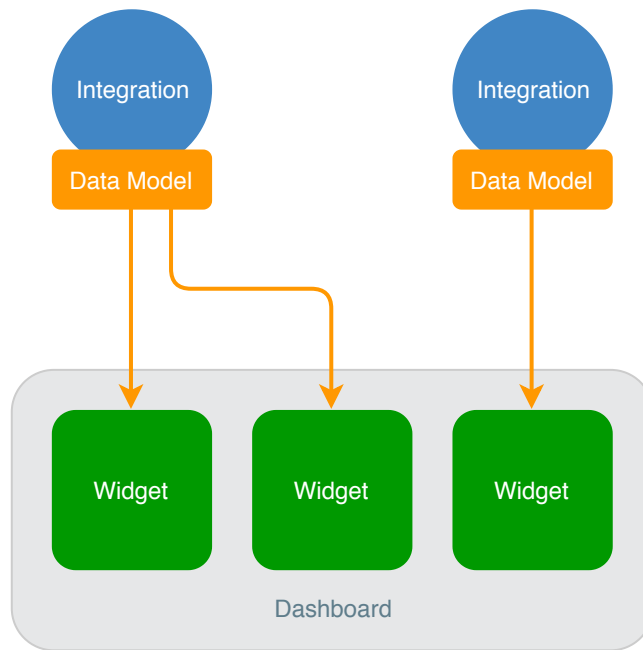


Figure 4.1: Integrations, data models and widgets together form a dashboard for data visualizations.

The architectural model is composed of two crucial components; *integrations* and *widgets*. Together they form a visualization dashboard.

- Integrations are responsible for fetching data, processing it and outputting a data model from a data source. They run the logic behind a dashboard and provide the system with data to be visualized. Integration code is written by the user and is run in a container sandbox environment as to keep the system and other users safe from harmful or badly written code.
- Widgets are responsible for actually visualizing data. They are connected to a data model (produced by an integration) and simply visualize it in a specific way. Users also write widget code in the form of HTML files. Widget code is run inside a sandbox iframe HTML element with limited capabilities as to what it can do. The same argument applies here for the safety of the system and users viewing a widget. E.g. a widget should not be able to redirect the web browser to an external website where potential harmful content may be hosted.

The data model generated by integrations can be connected to widgets (**Figure 4.1**). Data models can be combined with widgets to allow crowdsourced modules to be re-used, given that the widget can understand the data model.

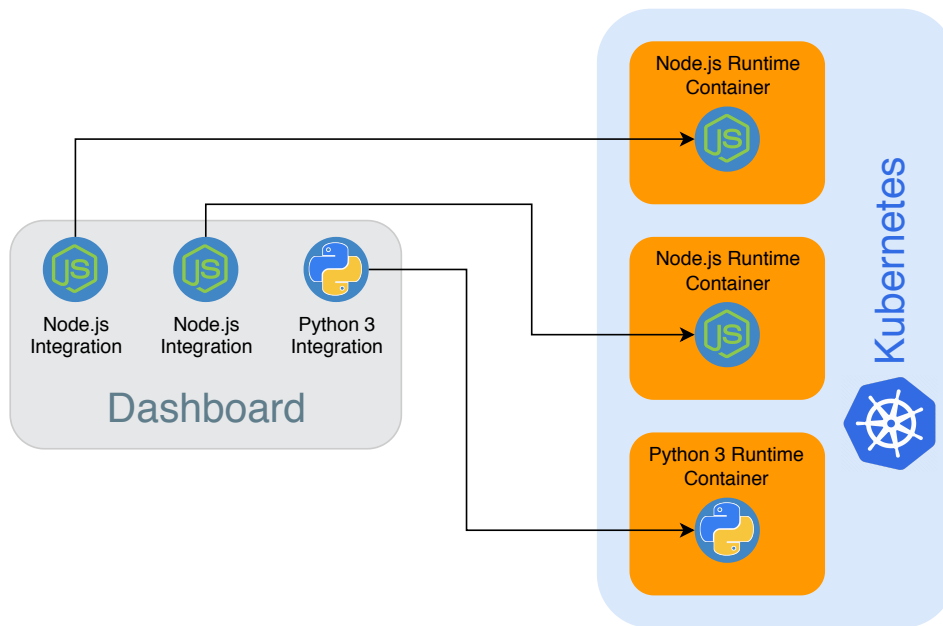


Figure 4.2: Integrations can be executed in different runtimes containers on the backend and are orchestrated in a Kubernetes cluster.

4.1 Integration Runtimes

A runtime environment is a Docker [24] image specialized for the chosen runtime language and is pre-built by the VisualBox system. A separate Docker image is built for each runtime environment. Each docker image includes libraries and binaries needed to run the integration source code for a specific runtime. The integration source code will be loaded and executed dynamically into the selected runtime image after the integration has been launched (**Figure 4.2**). This gives the integration developer choices on how an integration is written and run.

The VisualBox system only support three integration runtimes; Node.js, Python 3 and Go. However, this list can easily be extended by building a new Docker image with required libraries and binaries for a new runtime to function. The new integration runtime can then be made available to integration developers for use.

Runtime environments may require a build step if a compiled language is used such as Go. Interpreted languages such as Node.js or Python may also need to download required dependencies, so the entrypoint and start commands vary for each runtime (**Table 4.1**). These varying operations are defined in the Docker image and may not be changed by the integration developer.

Runtime	Prepare CMD	run CMD
Node.js	yarn install	node index.js
Python 3	pip3 install -r requirements.txt	python3 main.py
Go	glide install	go run *.go

Table 4.1: Prepare and run commands that are executed in respective runtime after launch.

By letting the runtime Docker image dictate how a project is prepared (compiled, dependencies downloaded) the VisualBox system can maintain a concise way of launching the source code. This limits the options for how a developer can structure their project, so an alternative approach is discussed in **Section 8.1 – Multi-stage Docker Image Builds**.

How to access the VisualBox API for sending data back to the system also varies for each runtime and is described in detail in the VisualBox integrations documentation [25] for each runtime environment. E.g. if the Node.js runtime environment is used, a global package called *visualbox* must be imported to allow the code to send data back to the VisualBox system:

```
// Import globally accessible 'visualbox' package
const visualbox = require("visualbox");

// Send some data back to VisualBox
visualbox.output("An output string");
```

Listing 4.1: Node.js integration code.

In the Go runtime environment a corresponding *visualbox* Go package must be imported:

```
// Import globally accessible 'visualbox' package
import "visualbox"

// Send some data back to VisualBox
visualbox.Output("An output string")
```

Listing 4.2: Go integration code.

4.2 Dashboard Builder

The dashboard builder is a tool in the front-end web application for combining widgets with data models that are generated by integrations. Widgets are arranged in a grid and visualize the data that they have been provided. The user must add integrations to the dashboard (so that data models are generated) and then add widgets. A widget needs to be connected to a data model before any data is visualized.

The following section will describe steps necessary to build a minimal functioning dashboard, where a widget is connected to a data model and visualize a piece of information in a specific way. The first step is to open the dashboard builder.

The dashboard builder is accessed by creating a new dashboard or by opening an existing dashboard (**Figure 4.3**).

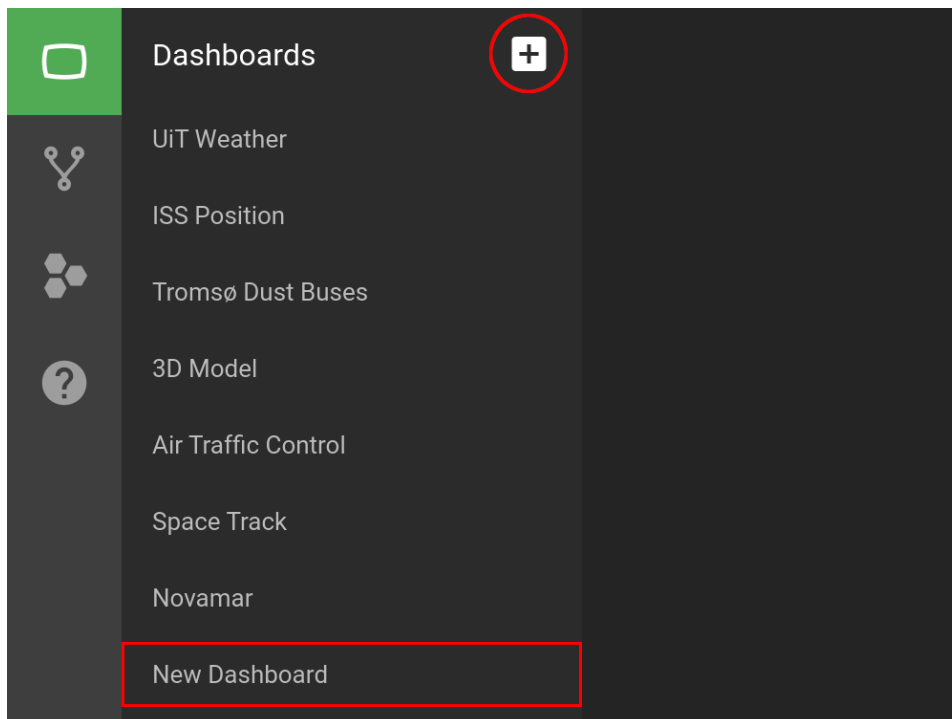


Figure 4.3: An existing dashboard can be opened by clicking on it in the dashboard list, or a new dashboard can be created by clicking the plus-button in the top right corner.

4.2.1 Main Panel

When a dashboard has been created and is opened, the main panel is displayed. If no widget has been added the dashboard will be empty. In the main panel there are five different buttons (**Figure 4.4**). The functions of the buttons are as follows:

1. Add integration button that will open the integration explorer.
2. Add widget button that will open the widget explorer.
3. Dashboard settings button that will open a new panel where the dashboard name and background color can be configured.
4. Fullscreen button.
5. Unlock/lock widgets so that they can be moved or resized.

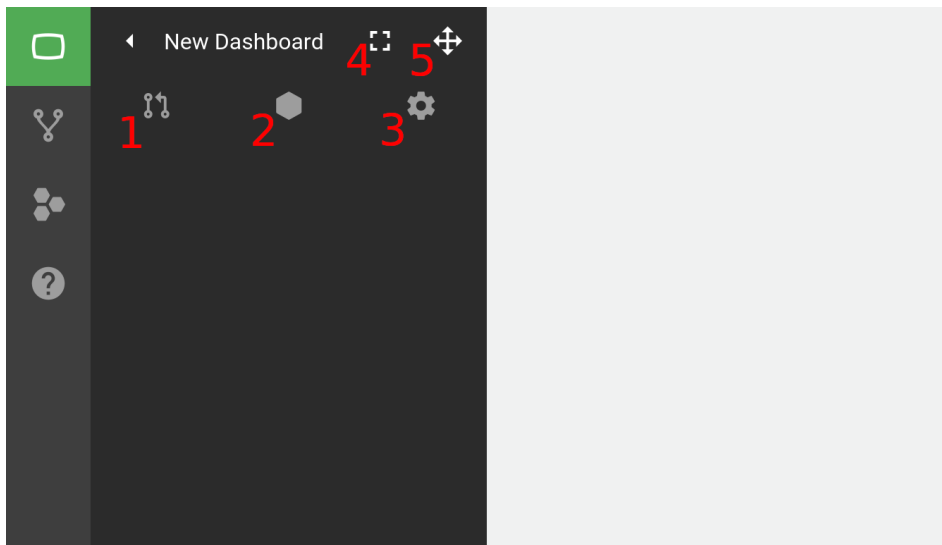


Figure 4.4: The main panel of a dashboard builder lets the user add/remove integrations and widgets.

These buttons will be used in the following sub-section when creating the minimum functioning dashboard.

4.2.2 Adding Integrations

If the "Add Integration" button is clicked (number 1 in **Figure 4.4**) the integration explorer is opened. Publicly available or local integrations (integrations that are created by the current user) can be explored and added to the dashboard. The explorer displays integrations that have been published to a registry and is part of the crowdsourcing developer model of the system. This will be discussed in more detail in **Chapter 5 – Crowdsourcing Developer Model**.

The user can choose an already existing integration and add it to the dashboard (**Figure 4.5**). Here we're adding an integration called "Managed IoT Cloud - Thing Shadow", which is an integration that connects to the Managed IoT Cloud API and fetches the latest data from an IoT device.

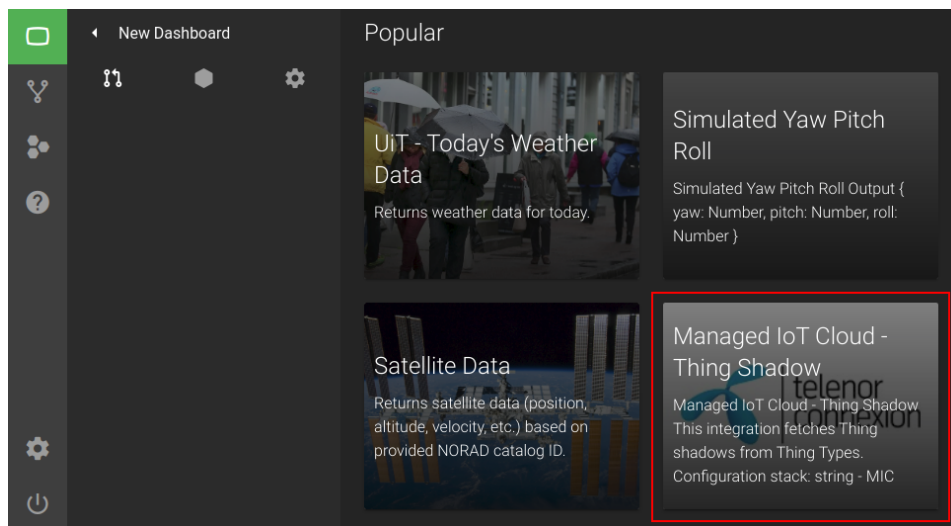


Figure 4.5: The integration explorer allows the user to add already existing integrations to their dashboard.

When we click the integration in the integration explorer we're shown the information page of the integration (**Figure 4.6**). The information page contains information about what the integration does, how it's configured and what the expected output will be.

If we decide to add the integration to the dashboard we can choose which version we want. The "latest" version is chosen by default, and will ensure that the integration always runs the latest code. If an older version is chosen the integration code will be "locked" and never change, even if the original author updates the code at a later stage. This behavior is described in **Section 5.1.1 – Modules are Versioned**.

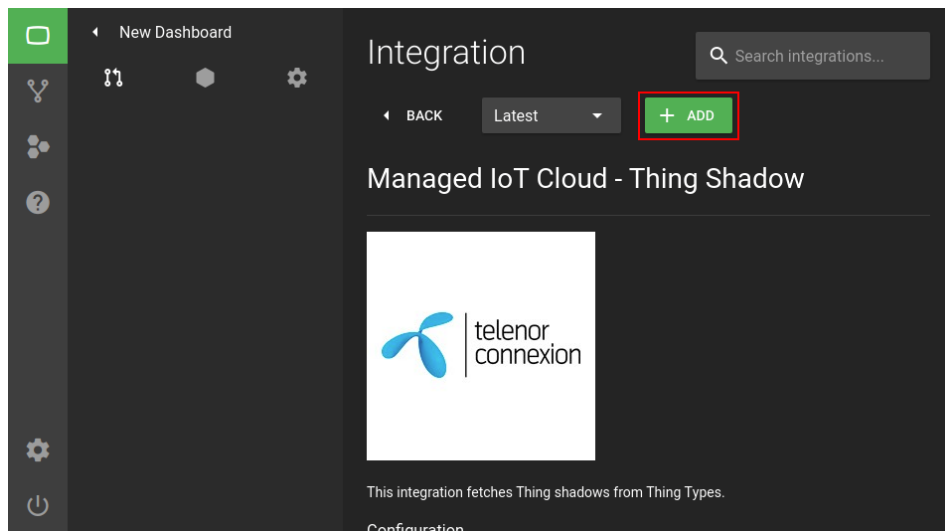


Figure 4.6: The information page is shown when an integration is opened from the integration explorer. The version can be specified before added to the dashboard.

After the integration has been added it will show up in a list in the main panel. The color of the newly added integration will initially be red, meaning that the integration code has not started yet. The color will eventually go from red to green after the integration has started (**Figure 4.7**). The integration startup time here is crucial and is part of the experiments in **Chapter 7 – Experiment and Evaluation**.

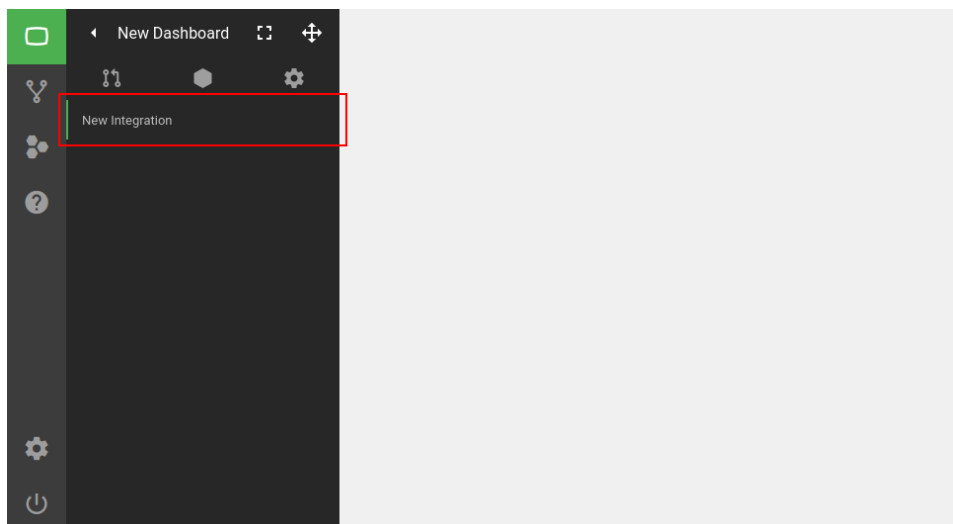


Figure 4.7: Added integrations to the dashboard are shown in a list in the main panel. Their color signals if the integration code is running (red or green).

After an integration has been added to the dashboard it will immediately start and eventually output a data model. If the integration is clicked, a configuration panel is shown where the user may specify input parameters for the integration in question. The input parameters vary and is completely defined by the integration itself. This is what's referred to as the *configuration data model* (see **Section 5.3 – Configuration Data Model**). E.g. if the integration needs to connect to an external API it may need credentials provided by the user. The integration will automatically restart when the input parameters are changed and saved (**Figure 4.8**).

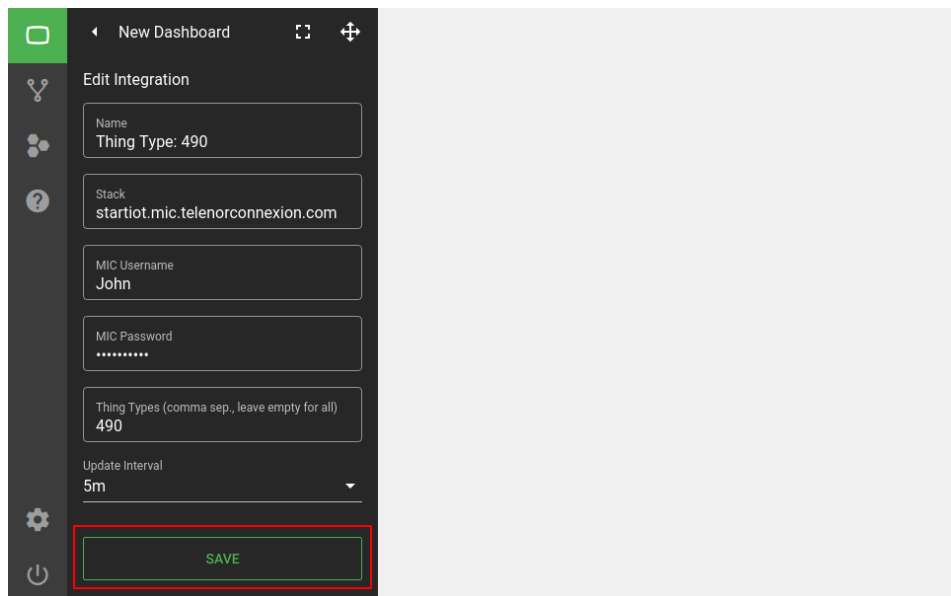


Figure 4.8: The configuration data model defines input parameters that the integration code can use.

4.2.3 Adding Widgets

Widgets are added to the dashboard by opening the widget explorer in the same way as when adding an integration. When a widget has been added it appears in the dashboard grid, where it can be moved and re-scaled.

By hovering over a widget in the dashboard grid, a menu of action buttons is shown. Here the widget can be edited, copied or removed. If the edit button is clicked the configuration data model for the widget is shown in the left side panel, much like when editing an integration (**Figure 4.9**).

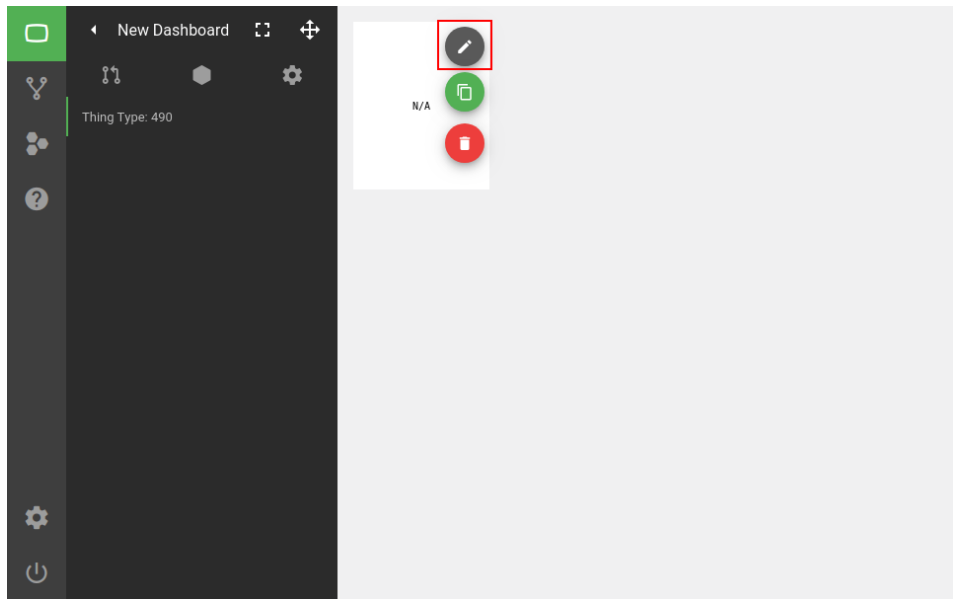


Figure 4.9: Action buttons are shown when a widget is hovered where it can be edited, copied or removed from the dashboard grid.

4.2.4 Connecting a Widget to an Integration

After a widget has been added to a dashboard it needs to be connected to a data model that has been produced by an integration. This is done by opening the configuration panel for a widget and opening the data model viewer (**Figure 4.10**).

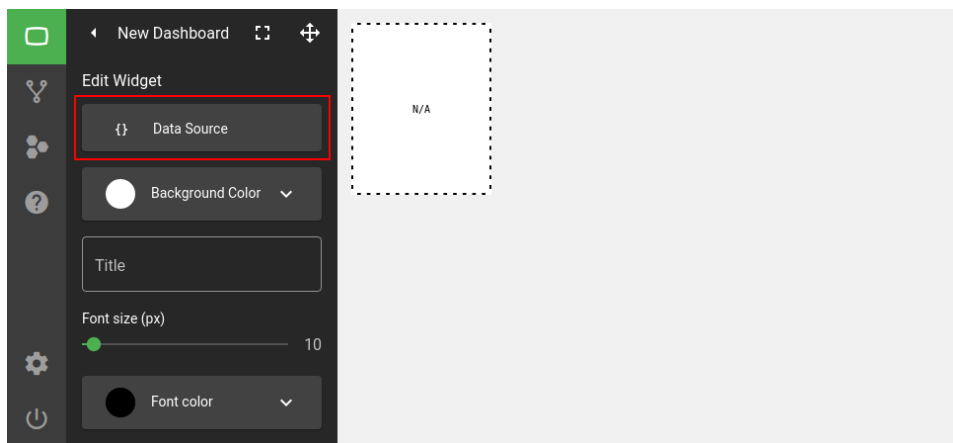


Figure 4.10: A widget can be connected to a data model (that has been generated by an integration) by clicking the "Data Source" button in the configuration panel.

If an integration has been added to the dashboard, and it has generated an output, the data model will be displayed for exploration (**Figure 4.11**).

The data model can be expanded to show more information about each property. Since the data model is represented as a tree, multiple values can be selected by selecting a node that is higher in the hierarchy. The selected data will then be available to the widget for use.

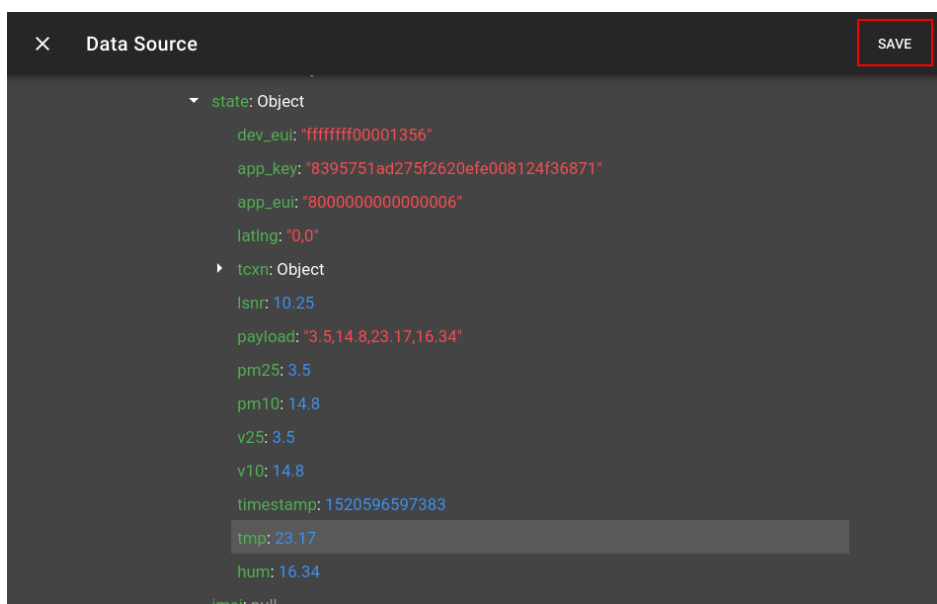


Figure 4.11: The data model viewer will display generated data models by integrations that has been added to the dashboard.

The data source of the widget is now pointed to a property in a data model that is generated by an integration, and will have access to the information whenever it is produced. This means that whenever a dashboard is re-opened, and widgets are loaded, they must wait until the integration that is responsible for producing the data source has started.

An example of a customized widget that is connected to a data source is shown in **Figure 4.12**. The widget that was used will simply display the value it as been provided, and can be assigned a title. The background color, font size and font color have been changed to give a more appealing look.

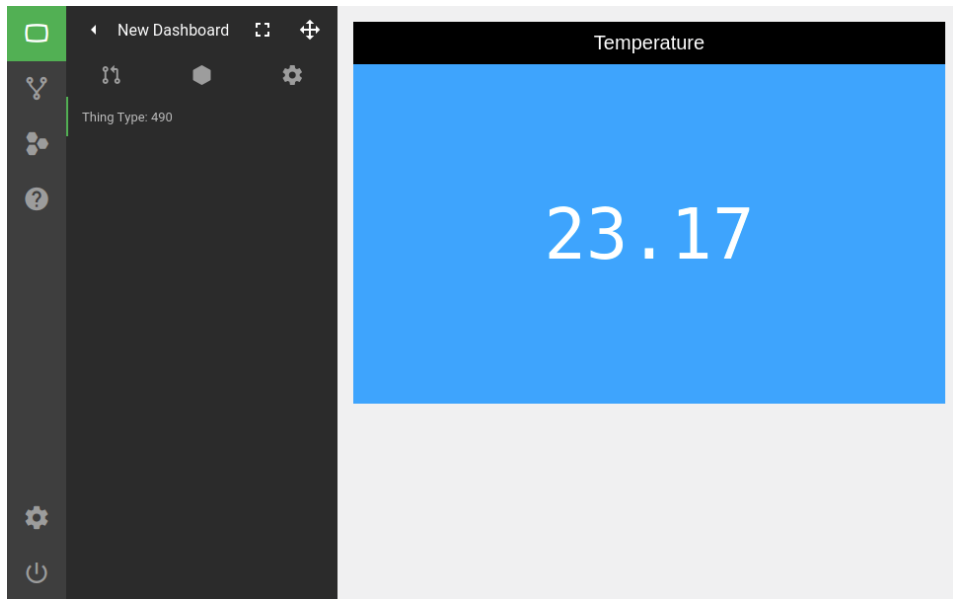


Figure 4.12: A minimal functioning dashboard with a single widget connected to a data model by an integration.

4.3 Architectural Model: Summary

This chapter has outlined how the architectural model is composed of integrations that produce data models and widgets that consume them. Integrations run in containerized Docker images that are pre-built, and act as runtime environments. Each runtime environment provide necessary libraries and binaries for the runtime to function.

A dashboard builder is used to add, configure and customize integrations and widgets. The data model explorer can be opened when configuring a widget and shows data models that has been produced by integrations. Widgets can then be connected to portions of data models that is then fed as input for the widget to visualize.

The following chapter will handle the crowdsource developer model of the VisualBox system; the perspective of a module developer.

/5

Crowdsource Developer Model

This chapter will handle the crowdsource model from a developers' perspective. A crowdsourced developer will write the code that integrations and widgets run. This perspective differs from that of a regular dashboard user, who will only use integrations and widgets to produce data visualizations. An integration or widget will be referred to as a module from now on in this chapter.

5.1 Shareable Modules

Widget and integration source code can be published to a public registry maintained by the VisualBox system. Published integrations and widgets will be indexed and made discoverable by other users. The source code becomes open source and can be re-used (forked), modified or extended. This opens up for widgets and integrations to be widely spread and shared, thus the notion of a crowdsourced ecosystem of modules.

As an example; if a highly generic widget module is developed which displays an array of data-points in a two-dimensional line graph, it can be used to visualize different data models from different integrations. There is no need to develop this type of widget multiple times, so by publishing this widget to

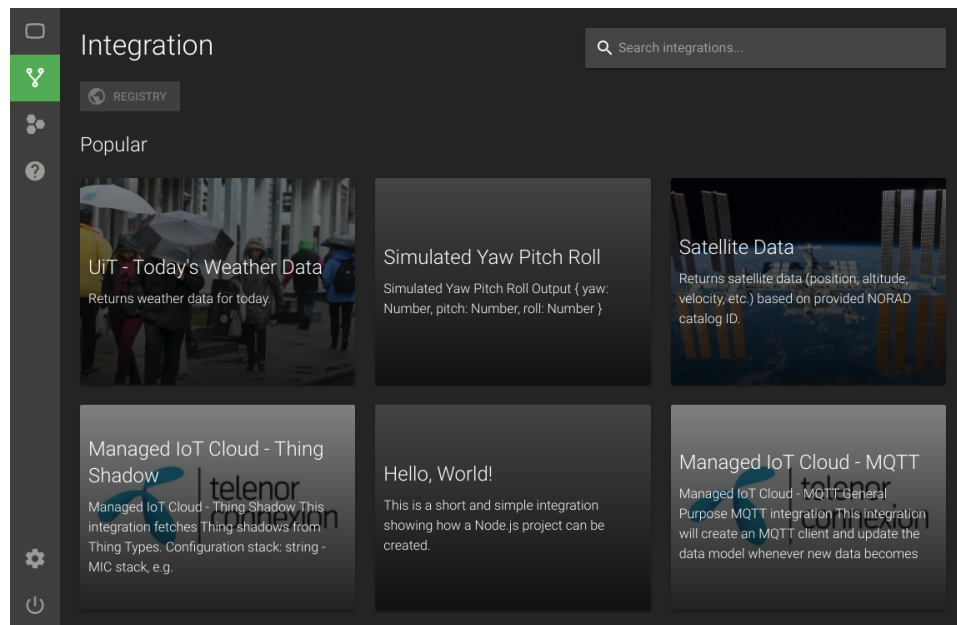


Figure 5.1: The explorer allows for the discovery of published integrations or widgets which can be forked or used.

the registry other users may use it to visualize their data models. This avoids "reinventing the wheel".

To fork a module source code it must first be found. The widget or integration *explorer* (**Figure 5.1**) is used to search for modules. When the desired module is found it can be forked by a single click of a button. The entire module is then copied and made as a new module in the account of the current developer. The forked copy can now be modified and/or re-published to the registry as a new module.

5.1.1 Modules are Versioned

Modules that are published to the registry are versioned. If you publish the same module more than once it must have a new version number and a snapshot is saved which can't be modified later. This ensures that widgets and integrations currently in use in other dashboards won't stop working if a breaking change is introduced to the source code by the module maintainer. Version numbers follow the Semantic Versioning Specification (semver) [26].

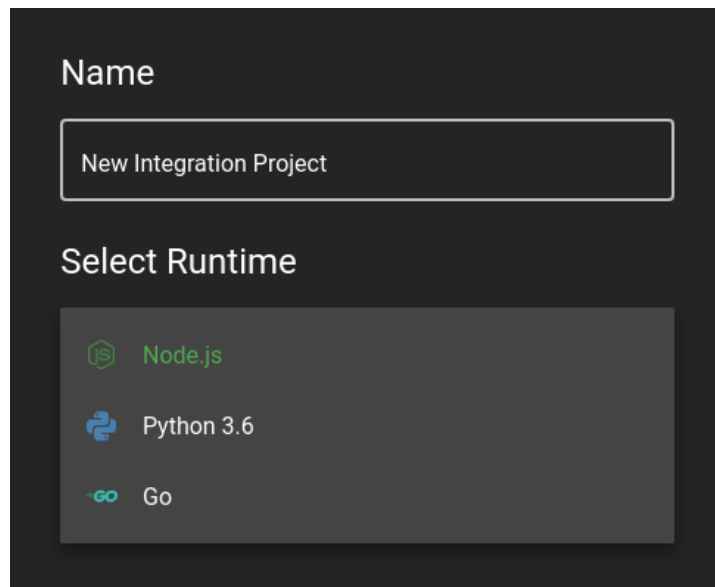


Figure 5.2: A runtime environment must be chosen when a new integration module is created.

5.2 Module Development

Integrations and widgets that are used in dashboards are written by the crowd-sourced developers of the system. This gives complete control on what and how code is run to generate and visualize a dashboard. To do so, a *project* is created. A project consists of multiple files and folders that are needed for the integration or widget to run and a selected *runtime environment*.

5.2.1 Select Runtime Environment

The VisualBox client is running as a web application, so the natural way to create visualizations is by using web technologies such as HTML, CSS and JavaScript. Since widgets in VisuaBox are responsible for visualizing data, the code is run inside of an iframe HTML-element. An iframe is an HTML element which allows one web-site to include another web-site into a framed portion. This frame is completely separated from the host web-site, but can communicate with the host using the *cross-document messaging* JavaScript API.¹ The environment must therefore always be HTML and JavaScript. Integrations on the other hand are not limited to run in the web browser (see **Section 3.3 – Server Side Data Processing**).

1. MDN web docs: "Window.postMessage()" <https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage>

When a new integration module is created the option to choose runtime environment is presented as shown in **Figure 5.2**. When the integration module is added to a dashboard it will be launched on the back-end in a container with the selected runtime environment.

5.2.2 Source Code Editor

The module developer may choose to use an in-browser code editor where the source code can be modified. The in-browser code editor is based on the Monaco Editor [27], which is the same editor that is used in VS Code [28], and supports common features found in an Integrated Development Environment (IDE) such as syntax highlighting, multiple tabs and text-search. The editor allows for rapid code modification without the need to setup a complete local development environment and is completely run inside a web browser.

5.2.3 Upload Module Code

The in-browser code editor provides a fast way for inspecting and make modifications to the module source code, but is in many cases not enough for the developer. The option to use a local development environment must be available.

A programmer may use their own local development environment and write the module source code on their own computer. After the module is ready to run, the source code can be compressed into a ZIP archive and uploaded to a VisualBox module. E.g. if a Node.js integration is to be created it can be developed locally using an IDE of the developers' choice, tested and run before compressed into a ZIP-archive and uploaded to VisualBox.

5.2.4 Sync with External Version Control Services

If a local development environment is used, or if multiple people are working on the same module, a version control system might have been used to e.g. keep features in separate branches and the possibility to revert changes if necessary. The open source project hosting site GitHub can be used to start, collaborate and develop integration code that will eventually run on the VisualBox service infrastructure.

By enabling the option to connect an external version control service such as GitHub, and synchronize the external repository with the internal module code, allows for wider collaboration and ease of use while transferring the

code into VisualBox. Continuous Deployment (CD) is an interesting method worth exploring where code that is pushed to a version control system can be compiled, transferred to and run in production immediately after the code was committed. This feature is not implemented but left for future work.

5.2.5 Preview Module

In the case of editing a widget, the rendered widget HTML can be previewed by opening a preview console. This allows for continuous feedback on what the final product will look like. A tab for displaying the widget itself and a tab for modifying the *configuration data model* is available (**Figure 5.3**). The configuration data model tab allows the developer to test the behavior of the widget code when the configuration model is changed, and is explained in the next section.

Similarly, when editing an integration it can be run in a development console. A live container is started and the standard output and error is printed for debugging purposes. Messages sent to the debug console are explained in **Section 6.4.6 – Socket Server**.

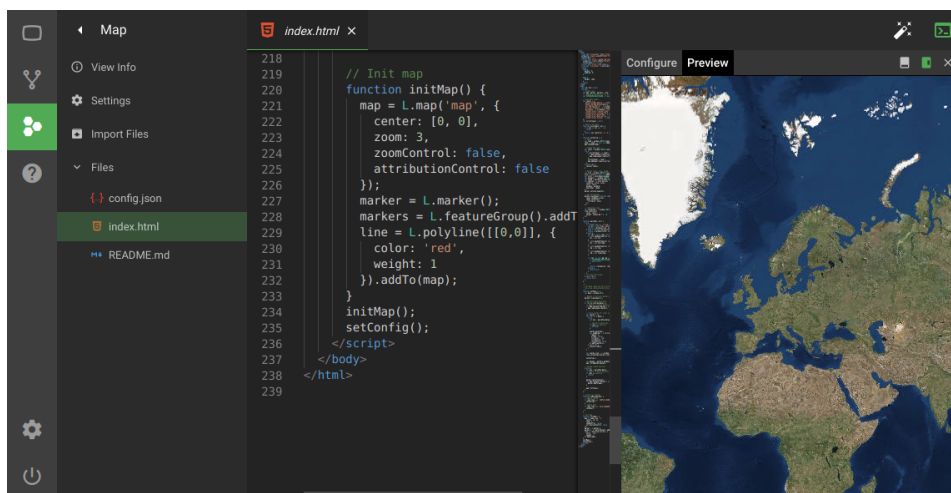


Figure 5.3: A widget can be previewed during development by opening the preview console (right). The configuration data model can be accessed by switching to the configure-tab.

5.3 Configuration Data Model

To allow dashboard users to alter the look and feel of widgets, a *configuration data model* can be defined. The configuration data model is a key-value map of variable names to user input values that are injected into the widget source code at runtime. E.g. a widget may need credentials to connect to a server. The configuration data model can then be configured to allow the user to input a username and a password, which are then transformed into variables that are injected and used in the widget source code.

The purpose of the configuration data model is to give the widget developer a controlled way of receiving user input and to separate variables from the source code. Integrations also has the same configuration data model where user input can be provided.

A configuration data model is by convention defined as an array of JSON objects in a *config.json* file placed in the root folder of the module project. A single object in the array takes the following form:

```
[
  {
    "type": "text",
    "name": "myVariable",
    "label": "Input Label",
    "default": "Default value"
  }
]
```

Listing 5.1: A sample config.json file. Different fields can be required depending on the type of the configuration object.

Type describes what input type to render. It can be text, password, color, switch, slider, date or select.

Name is the variable name to be injected into the widget/integration code and is used by the widget/integration developer.

Label is a user facing input label.

Default is the default value that the input should initially have.

The system will interpret the JSON-formatted file contents and render a HTML form with different input fields for a user to interact with. An example of what a rendered configuration data model input form may look like with text, slider, switch and color fields is displayed in **Figure 5.4**.

The image shows a configuration panel for a widget. It features a dark background with white text and green accents. At the top, there is a 'Text Color' dropdown menu with a white circle icon. Below this are several input fields: 'Title' with the value 'foo', 'Title Size' with a slider set to 51, 'Value Size' with a slider set to 187, 'Label' with the value 'data', 'Suffix' (empty), 'Min Value' with the value '-400', and 'Max Value' with the value '400'. Further down are 'Line Width' with a slider set to 40, a checked 'Automatic Line Width' toggle, an unchecked 'Show Legend' toggle, a checked 'Expand' toggle, and 'Transition Speed (ms)' with a slider set to 259. At the bottom, there are two more dropdown menus: 'Fill Color' with a blue circle icon and 'Fill Background' with a dark grey circle icon.

Figure 5.4: A config.json file has resulted in the following HTML form where the user can input values that are then injected as variables into the widget or integration code. This may alter the appearance and/or behavior of the widget/integration.

As an example, a gauge widget has been added to a dashboard as shown in **Figure 5.5**. At the top, the widget is in its initial, unconfigured state. If the widget is edited, the rendered configuration model HTML form is shown in the left-side pane. The bottom image shows the same widget with different configuration values.

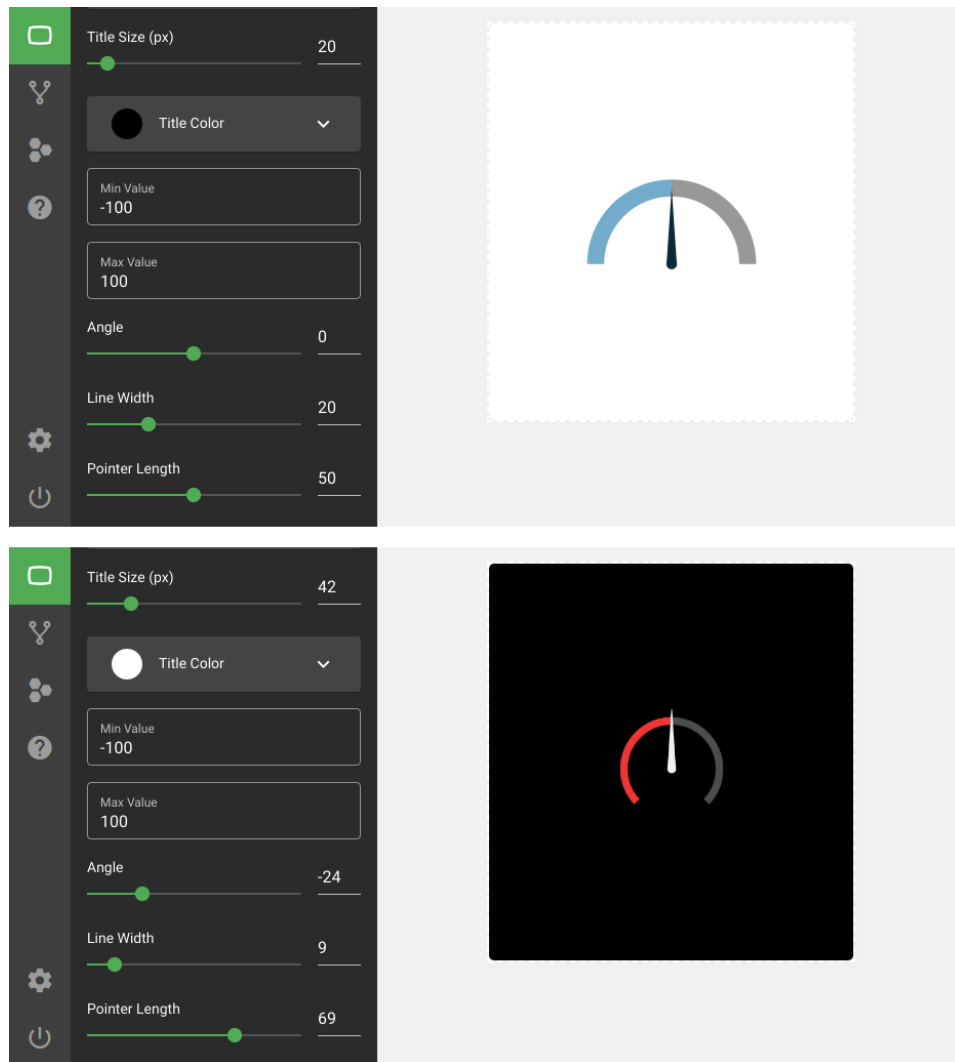


Figure 5.5: The same gauge widget with an unconfigured (top) and a configured (bottom) configuration model.

/6

Implementation

Regarding the infrastructure necessary to build and run a saas you have two choices; either you host your own servers and storage machines (and maintain the facilities, networking, electricity and server provisioning yourself) or you use a cloud provider such as AWS. During this thesis work, the architecture is built to run on AWS and utilizes many of the various services provided by Amazon Web Services. In this chapter we will therefore take a closer look at the AWS services used to create the cloud architecture that powers the VisualBox system and how they are tied together on a higher level.

6.1 Infrastructure as Code

Cloud computing is becoming more and more prevalent in the way computer systems and services are built. By moving infrastructure "into the cloud", companies can focus on building the application and don't have to worry about provisioning and maintaining machines.

Infrastructure as Code (IaC) [29] is an automation service pioneered by AWS which deals with problems surrounding infrastructure operations. Physical and virtual infrastructure residing at the cloud provider is kept as code and is used to provision, update and remove so called "artifacts". IaC is provided as a service by AWS and is called AWS CloudFormation [30].

Key points of using IAC as described by AWS:

- Lower cost because human capital can be removed that would else be used for provisioning and maintaining infrastructure.
- Higher consistency following configuration standards.
- Agility by the speed of which new versions of the service can be released.
- Attaining and maintaining compliance to corporate or industry standards.

By defining "cloud resources" as definitions in code, the developer can treat their physical deployment as software. When a set of cloud resources has been made, the developer can send IAC artifacts to the cloud provider and the whole architecture is then build as a coherent stack. The advantages of this approach is easier replicability, architectural changes can be version controlled (using e.g. Git), errors and other misconfigurations can be rolled back to a previously working state and transferability to different cloud providers is trivial.

Many components of the VisualBox system is built using AWS such as Lambda functions, DynamoDB tables [31][32] and S3 buckets [33]. IAC is used for deployment of cloud resources together with the Serverless Framework.

6.1.1 Serverless Framework

The Serverless Framework [34] is a tool for writing and deploying IAC to multiple cloud providers. The framework uses a normalized template-language so that cloud resources can be defined similarly over a set of different cloud providers.

Much of its popularity has come from the fact that it lets developers easily define *functions* that are then translated into its respective service in the cloud provider, as well as scaffolding necessary HTTP endpoints for it to be reachable over the internet. In the case of AWS this involves setting up an API Gateway [35] and connecting a Lambda function to it using the Proxy Integration.¹ This is what is commonly referred to as a *microservice*, or a *serverless function*.

1. Serverless Framework AWS documentation: "Lambda Proxy Integration"
<https://serverless.com/framework/docs/providers/aws/events/apigateway#lambda-proxy-integration>

The Serverless Framework also makes it easier to separate different *stages* of a stack of cloud resources. This is useful when testing new features by deploying a development stage in a complete replica of the production stage. When the new feature has been tested and is ready to be released it can be deployed and updated in the production stage.

VisualBox is utilizing the capabilities provided by the Serverless Framework to define, update, deploy and version control cloud resources in AWS. As described later in **Section 6.3 – VisualBox Cloud Architecture**, each *service* is its own Serverless Framework stack.

6.2 AWS Lambda Functions

AWS Lambda functions are stateless *microservices* that are invoked by an *event* and return a *value*. A developer can upload the source code of the AWS Lambda function that then becomes immediately available to be invoked across the AWS infrastructure. The main benefit of AWS Lambda functions is that the developer do not have to handle server provisioning or scaling. This is automatically handled by AWS and makes the speed and agility to develop applications composed of multiple microservices fast and trivial.

AWS Lambda functions are billed by *execution time*. This means that the customer only pays for what is actually used when an AWS Lambda function is invoked. Idle functions are not billed.

An AWS Lambda function may also invoke another action which makes it possible to chain multiple AWS Lambda functions. This is what is commonly referred to as *event based architecture*. E.g. when a file is uploaded to a Simple Storage Service (S3) bucket it can send an event to an AWS Lambda function to take the uploaded object and process it.

6.2.1 Creating an HTTP endpoint for an AWS Lambda

An AWS Lambda function is by default not exposed to the internet as an HTTP endpoint. To make this possible, an API Gateway is commonly used together with AWS Lambda functions to create and expose an HTTP endpoint for an external application to use. The API Gateway will expose an HTTP endpoint and when called, it will create an HTTP event for a connected AWS Lambda function. A single AWS Lambda function can also be connected to different API Gateway endpoints. E.g. the same AWS Lambda function can handle both GET and POST methods for the same HTTP endpoint.

The Serverless Framework automate the process of setting up an API Gateway and connect an HTTP endpoint to the AWS Lambda function. The API of the VisualBox system is composed of multiple AWS Lambda functions that are connected to a single API Gateway with different endpoints.

Creating a function in the Serverless Framework is done by defining a function name, the handler and a list of events.

```
getFunction:
  handler: get.main
  name: ${stage}-integration-get
  events:
    - http:
      path: integration
      method: get
```

Listing 6.1: Serverless Framework function definition with an HTTP event source.

If the cloud provider is set to be AWS, the Serverless Framework will take the definition in **Listing 6.1** and create a Lambda function with the name `${stage}-integration-get`, and replaced the first part with the stage name of the deployed stack. An API Gateway would also be created with a single HTTP GET endpoint at the URL path `/integration`. A client application can now call the endpoint and invoke the AWS Lambda function. The actual function code is pointed at by the handler and is written in a file called `get` and must contain the entry function `main()`.

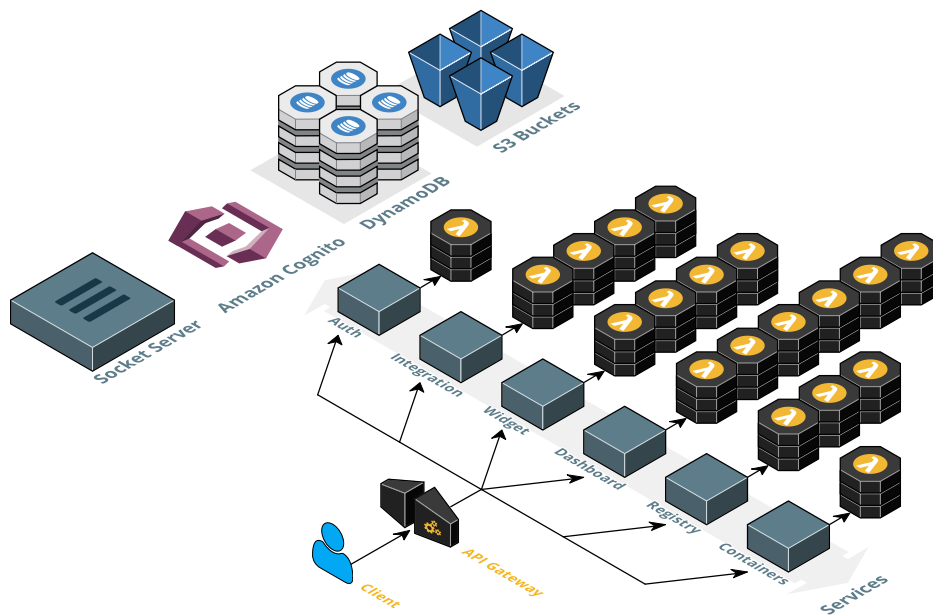


Figure 6.1: Architectural overview of the VisualBox back-end composed of multiple services and AWS cloud resources.

6.3 VisualBox Cloud Architecture

The VisualBox cloud architecture is grouped into six separate *services* (**Figure 6.1**). Each service is a separate Serverless Framework definition file and contains definitions for cloud resources that are related. These services mainly defines and implements a set of related AWS Lambda functions, but may also define AWS Identity and Access Management (IAM) policies and roles, which are omitted from the figure for brevity.

By breaking down the whole architecture into these services only parts of the architecture can be updated without interrupting other parts of the system. The only component in the figure not deployed to AWS is the Socket Server. This is due to limited event-driven socket capabilities in AWS, but this is something that is expected to be generally supported.²

2. Blog post by Chris Munns: "Announcing WebSocket APIs in Amazon API Gateway"
<https://aws.amazon.com/blogs/compute/announcing-websocket-apis-in-amazon-api-gateway/>

The following list contains brief explanations on what is contained within each service and what responsibilities they have in the VisualBox system:

- **Auth Service** defines one single AWS Lambda function called *authMessage*. The AWS Lambda function takes as input an Amazon Cognito [36] event which allows the function to take action once a user interacts with the Amazon Cognito service. In this case the function is used to send an email message to the user after a sign-up event has been emitted, providing the user with a link that they must follow to verify their email-address.
- **Integration Service** defines four AWS Lambda functions that together implements the four basic functions, create, read, update and delete (CRUD), typically found in a RESTful API endpoint. Each function interacts with an Amazon DynamoDB table called *integrations* where metadata about data source integrations are stored.
- **Widget Service** is almost identical to the Integration Service with the exception that each CRUD function interacts with a different Amazon DynamoDB table that is exclusively storing widget metadata.
- **Dashboard Service** defines the most AWS Lambda functions. It also implements the four CRUD functions but the last two functions are designed to retrieve integration and widget metadata once a dashboard has been opened. These two functions are grouped in the Dashboard Service since the metadata is required only when the client application has opened a dashboard.
- **Registry Service** defines three AWS Lambda functions called *publish*, *delete* and *LFP*. LFP is described in **Section 6.4.4 – Lambda File Provider (LFP)** and *publish* in **Section 6.6 – Publishing a Widget or Integration** – and revolve around the registry used for crowdsourced widget and integration discovery.
- **Container Service** defines a single AWS Lambda function called *LTL*. The LTL is described in **Section 6.4.2 – Lambda Task Launcher (LTL)** and is used to launch containers and generating instance session tokens. The LTL also interacts with the last Amazon DynamoDB table which is called *container access* and is used for storing container access records as described in **Section 6.4.5 – Container Access Record (CAR)**.

6.4 Launching a Container

There are two AWS Lambda functions involved when an integration is started; the Lambda Task Launcher (LTL) and Lambda File Provider (LFP). They each play important roles in setting up the container with its initial parameters as well as providing the container with its source code without exposing it publicly.

6.4.1 Initial Parameters for a Container

As will be explained in the following sub-sections, a newly launched container will start with a set of initial parameters. A container that is starting doesn't have the notion of state and starts as a blank slate. The container must, with the help of these initial parameters, find the correct source code, configuration data model and establish a private socket connection with the client web application.

The following is a list of initial parameters that are given to a newly launched container:

Parameter	Description
TOKEN	An Instance Session Token (IST) unique to the current client-container session.
I	A unique number for the current container since multiple containers may be launched during the same session.
ID	The unique integration module ID.
VERSION	The integration module version to be used.
MODEL	The initial configuration data model and its values.
RUNTIME	The current runtime environment of the container.

Table 6.1: Initial parameters for a container.

6.4.2 Lambda Task Launcher (LTL)

The LTL is an AWS Lambda function used to start a project (or task) in a container (**Figure 6.2**). It is the first step for the client application to invoke this function when a dashboard is opened. The Lambda function takes as input a list of integrations it should launch (1). The LTL is stateless so each integration in the input list must include the integration ID, integration version and the current values for the configuration model.

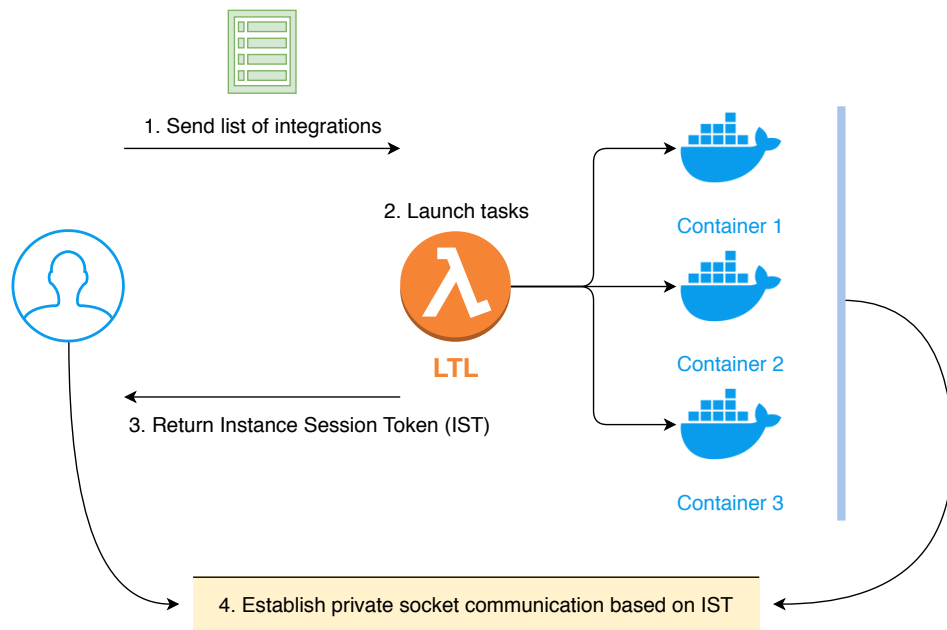


Figure 6.2: Steps involved for a client to launch one or more integrations in separate containers and establish a private socket communication channel.

The LTL will verify that each integration exists and that the user currently invoking the function has access to them. The correct Docker image is determined based on the runtime environment of an integration and a container is launched with the correct initial parameter configuration (2).

Before integrations in the list gets launched in separate containers, a randomly generated token is produced which represents the current Instance Session Token (IST). This IST is included as an initial parameter for the container bootstrap configuration. After every integration has been launched the IST is returned to the client application (3) and a socket channel is established with a socket server (4). The IST is used during the socket-handshake so that a private communication channel can be established between the client and all containers in the current *instance session*. This way the client can receive output and send commands to individual containers in the current dashboard.

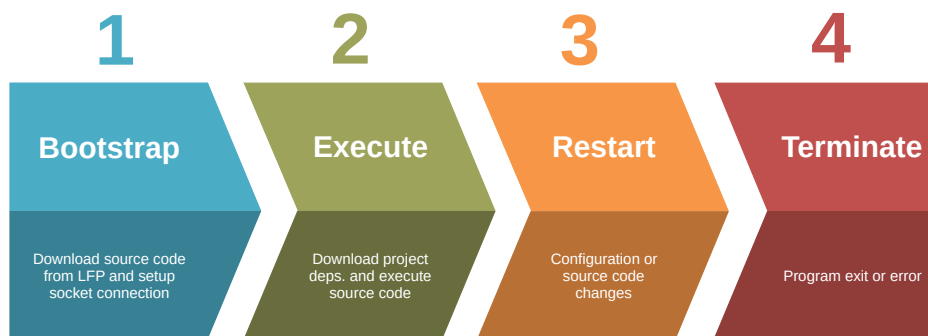


Figure 6.3: The container lifecycle is governed by the bootstrap binary which is running in each container.

6.4.3 Container Bootstrapper

Each container includes a bootstrap binary program which is responsible for loading the user source code, perform prepare and run commands and handle socket communication with the running user program and the VisualBox client. The bootstrapper is responsible for upholding the container lifecycle as depicted in **Figure 6.3**.

When a container is started it is given the initial parameters (such as integration ID, version and configuration model values) necessary for the bootstrapper to start the integration module. The first step is to fetch and download the correct integration source code. This is done by invoking the LFP.

6.4.4 Lambda File Provider (LFP)

The LFP is an AWS Lambda function responsible for retrieving integration source code based on its ID and version number. Source code is stored as a ZIP archive in an AWS S3 bucket. The LFP simply locate the ZIP archive and return it to the invoker. A running container will utilize this function during its bootstrap phase to retrieve its source code to be executed.

Since running containers are stateless the LFP has no way of knowing if the invoker has access to the requested integration source code. Integrations that are not published to the registry are considered private to the author of the source code, and should under no circumstances be accessible by anyone else. A method to limit what ZIP archives the LFP can access is required, so the Container Access Record (CAR) table is implemented.

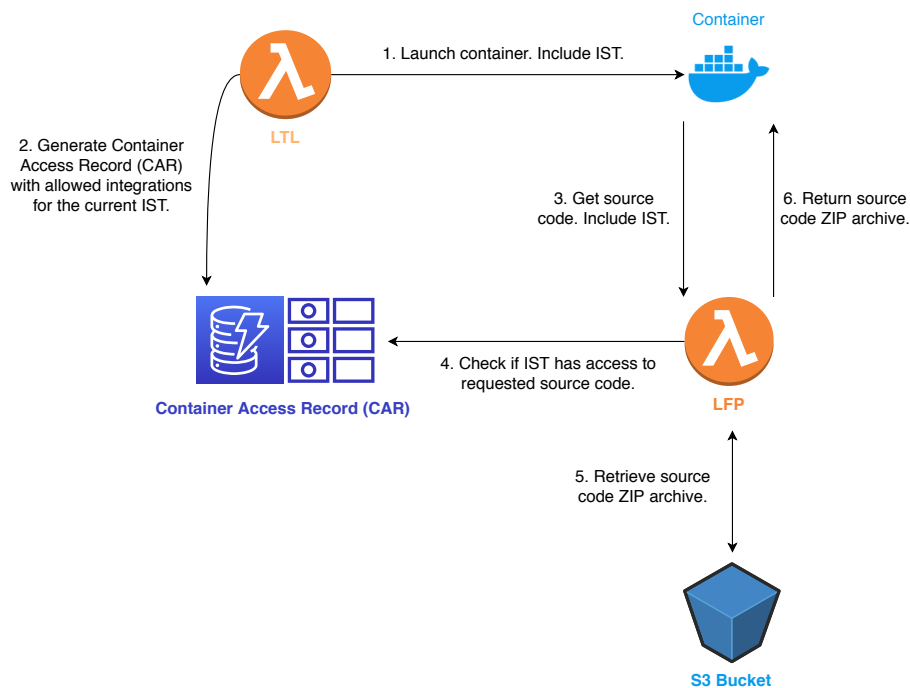


Figure 6.4: Steps involved for a container to get its source code from the Lambda File Provider (LFP).

6.4.5 Container Access Record (CAR)

A CAR is generated during a LTL invocation and contains the same list of integration ID's and their versions that was used during the function invocation, and is stored in an AWS DynamoDB table (**Figure 6.4**, no. 2). The IST generated by the LTL is included in the CAR so that the token is locked to the set of integrations originally launched. The IST is given to a launched container (1) which will in turn be used when the container invokes the LFP to retrieve the project source code (3). The LFP requires the IST and will use this to determine which integrations the invoker (or container) can access (4). This ensures that a container can never launch an integration not defined within the scope of its IST since its creation.

If everything checks out, the IST is valid and the requested project source code is within the scope of the IST, the LFP will retrieve the ZIP archive stored in an AWS S3 bucket (5) and return it to the container (6).

The IST becomes the key to which a container can identify and authenticate against the LFP to fetch the source code necessary. As described in **Section 6.4.2** the IST is also used by the bootstrapper to connect a container to a private socket channel so that a client can communicate.

6.4.6 Socket Server

The socket server is a Socket.IO [37] server responsible for setting up a private communication channel between a VisualBox front-end client and associated containers in a dashboard. Socket.IO has the concept of *rooms*, which groups connected socket clients and can be used to send messages between them.

The generated IST for a dashboard is used as the room name.

As soon as a client opens a dashboard and the IST is returned, the client will join the room by the IST-name and listen for messages of the type *INIT*. An *INIT* message is only sent once by a container that has just started, and signals that the container is ready. The client will then know that an integration has started and can inform the user by changing its color from red (not started) to green (started). The *INIT* message includes the number of the integration in the dashboard so that if multiple integrations are present it can know which integration in the current instance session that sent the *INIT* message.

Message Type	Payload	Description
INIT	Integration Number	First message sent by a container when it has started. The integration number is unique in the session instance for this integration so that the client knows which integration sent the message.
OUTPUT	Integration Number and Data	A message of this type is sent by a container when the integration code has generated data that can be used by widgets in a dashboard. The integration number is included so that the client can sort the data model by integrations in a dashboard.
STATUS	Status Type and Data	A message of this type is sent for all other messages not related to the data model by a container. Possible Status Types are defined in Table 6.3 .

Table 6.2: Socket Message Types sent between the client application and containers.

Status Type	Description
T_INFO	Standard output of the executed integration code.
T_WARNING	Standard error output during the prepare command of an integration bootstrap.
T_ERROR	Standard error output during the run command of an integration.
T_TICK	Signal from a client to a container that the connection is still alive.

Table 6.3: Possible Status Types for the STATUS message.

Once an integration has generated a data model it is sent as an *OUTPUT* message. The *OUTPUT* message includes the integration number and the data as payload. All other messages come in the form of *STATUS* types.

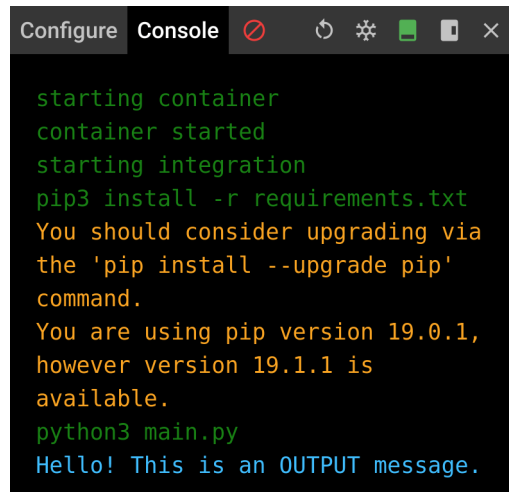
If any standard output is generated during the execution of integration code it is sent as a *STATUS* message with the type *T_INFO*. This *STATUS* message is only visible in the client if the user is utilizing the development console while developing an integration in the in-browser code editor.

A *STATUS* message with the *T_WARNING* type is sent if, during the prepare command in the bootstrap phase of an integration, a standard error is generated. This message is also only visible if using the development console.

The *T_ERROR* type is similar to *T_INFO*, which is sent if a standard error is generated during the execution of the integration code.

The *T_TICK* type is sent by the client to keep running container alive. The container bootstrapper will continuously decrement a counter for 1 minute. Each time the bootstrapper gets a *STATUS* message with this type it will start the counter from the beginning, keeping the container alive. This is a mechanism to gracefully shut down a container if the client is suddenly disconnected.

These messages allow the module developer to know what's going on in the container and send commands to restart it. An example of different STATUS message types and an OUTPUT message being sent to the client development console can be seen in **Figure 6.5**.

A screenshot of a development console window titled "Console". The window has a dark background with text in different colors: green for status messages, yellow for warnings, and blue for output. The text in the console is as follows:

```
starting container
container started
starting integration
pip3 install -r requirements.txt
You should consider upgrading via
the 'pip install --upgrade pip'
command.
You are using pip version 19.0.1,
however version 19.1.1 is
available.
python3 main.py
Hello! This is an OUTPUT message.
```

Figure 6.5: The development console with three different messages; T_INFO (green), T_WARNING (yellow) and OUTPUT (blue).

6.5 Widget and Integration Indexing

If a developer wants to open source a widget or an integration module it can be published to the registry maintained by the VisualBox service. To do so, the module needs to be versioned and indexed so that it can be discovered by other users using the widget or integration explorer. The published widget or integration can then be added to all dashboards, or copied and modified. This allows code to be re-used and improved. Reusability also prevents "reinventing the wheel" and keeps the time of creation for visualization prototypes to the minimum.

Module source code metadata such as author, module name, environment and thumbnail images are stored in DynamoDB tables. DynamoDB is a highly available, eventually consistent key-value store and is designed to handle large amounts of operations in a short time, but does not have strong text-based search capabilities. DynamoDB only support scanning index shards of entries or query a primary key.³ This is not enough for the widget or

3. As described in the Amazon DynamoDB Developer Guide:
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Scan.html>

integration explorer in VisualBox where text-based search is required, so a separate index has to be made.

Algoliasearch [38] is an index service with great free-text search capabilities and is chosen as the index for published widgets and integrations in VisualBox. An integration or widget is indexed in Algoliasearch when the module is published. By maintaining a unique ID of each index element, a one-to-one mapping can be made between the DynamoDB table containing the source metadata and the Algoliasearch index result whenever a search is made.

6.6 Publishing a Widget or Integration

A separate AWS Lambda function called *publish* is invoked once a module is going to be published to the registry. The Lambda function receives as input a module ID and its desired new version. The goal for the Lambda function is to create a new copy of the module and tag it with the new version number. AWS S3 support object versioning. This means that the same object can contain multiple "snapshots" with different versions. An AWS S3 object version is an MD5 hash which corresponds to a physical snapshot.

A published widget or integration cannot be modified. This is to prevent breaking dashboards that are using widgets or integrations on a specific version number. E.g. if a widget or integration is added to a dashboard with a specific version *X*, and version *X* is changed to include code that completely alters the behavior or appearance, then all dashboards using version *X* could potentially break. Published widgets and integrations are therefore immutable. Although, a module can be removed completely from the registry.

As depicted in **Figure 6.6**, the client will invoke the *publish* Lambda function by sending the module ID and its new desired version (1). The *publish* Lambda function will then create a new AWS S3 object for the ZIP archive corresponding to the module ID (2). AWS S3 will then create a new snapshot (3) and return the generated snapshot hash to the *publish* Lambda function (4). The *publish* Lambda function finally create or update the index record with the new metadata.

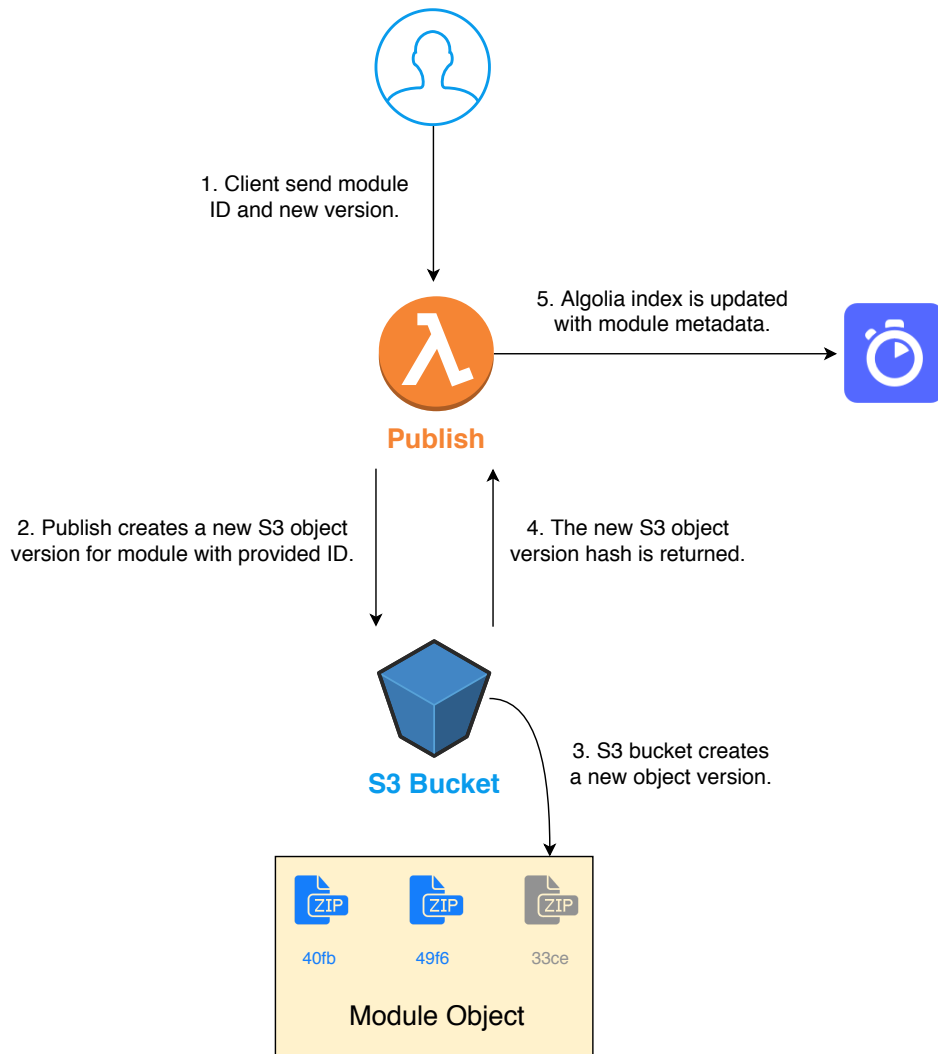


Figure 6.6: Workflow when a client wants to publish a new version of a module.

6.7 Implementation: Summary

In this chapter we've covered the cloud infrastructure in AWS that powers the VisualBox module platform and how the infrastructure is divided into separate services that each contain related cloud resources.

IaC is a automation service that is used together with the Serverless Framework to provision cloud resources that can be launched and updated in separate stacks. The VisualBox architecture is grouped into six services; Auth, Integration, Widget, Dashboard, Registry and Container. By making this separation, only portions of the architecture can be modified without the need to re-deploy other parts of the system.

Once a user opens a dashboard in the front-end web application, the LTL AWS Lambda function is invoked. This function is then responsible for producing a randomly generated token, the IST, that represents the instance session of the dashboard. Integrations are launched in separate Docker containers and include a set of initial parameters that identify in which instance session they belong. The IST is used to establish a private socket communication between the front-end web application and integration containers.

To protect integration source code from getting accessed by non-authorized containers, a Container Access Record (CAR) DynamoDB table is used. The CAR table is a list of integrations that are allowed for a specific IST. When a newly launched integration container requests the source code to be run from the LFP AWS Lambda function, a check is made against the CAR table. If the integration is included in the current CAR, the source code is retrieved and returned to the integration container. This ensures that a container won't have access to all integration source code in the registry, unless granted during the generation of the IST.

A bootstrap binary program is included in every integration container. The bootstrapper is responsible for upholding the container lifecycle such as:

- Downloading the integration source code from the LFP.
- Establish a socket communications channel with the front-end web application.
- Receive commands and send source code output from/to the front-end web application.
- Terminate container if dashboard is closed or code exit/error.



Experiment and Evaluation

In this chapter we experiment with the container startup time while using AWS ECS Fargate as the container orchestrating service. AWS ECS with the Fargate container launch type is the most suited for running containerized integration code in VisualBox, since all cluster management is completely hidden from the developer and makes it faster and easier to develop containerized applications. Therefore the performance must be thoroughly examined.

7.1 Container Startup Time

Each time an integration is used it is launched within a Docker container. This means that the container startup time must be as low as possible. Ideally it should not take longer than a couple of seconds, or the user may experience frustration while using the VisualBox service.

The container startup time is defined as the time it takes from the initial command to launch until the container image is running.

7.2 Experimental Setup

Five different Docker images with different sizes were produced to examine the potential impact image size would have on container startup times. Each image was built using various base images such as *Alpine Linux* and *Ubuntu* to get an evenly distributed size spectrum. The following image sizes were obtained to be used in the experiments:

4.21MB	29.6MB	56MB	69.9MB	95.2MB
--------	--------	------	--------	--------

Table 7.1: Image sizes used in the experiments.

They were then pushed to an AWS Elastic Container Registry (ECR), which is a hosted container registry. When using AWS ECS with Fargate, containers has to pull images from a registry in ECR.

When a container is to be launched in AWS ECS with Fargate, a *task definition* has to be made. The task definition specify which network interfaces and subnets to use but also how much memory and vCPU's the container will be assigned. The experiments were divided into three different task definitions with varying memory and vCPU count.

Experiment	Memory	vCPU
#1	0.5GB	0.25
#2	1GB	0.5
#3	2GB	1

Table 7.2: AWS ECS Fargate task definition configurations.

An AWS ECS cluster was then made and tasks were launched using the Fargate launch type. The recorded startup time was calculated by taking the difference in time of creation and time of termination from the information page of a task in the AWS ECS cluster console. Each data point in every experiment was run five times to produce a standard deviation.

7.3 Results

The first experiment was configured to use a task definition with *0.5GB memory* and *0.25 vCPU*. Each image was launched five times and the following results were obtained:

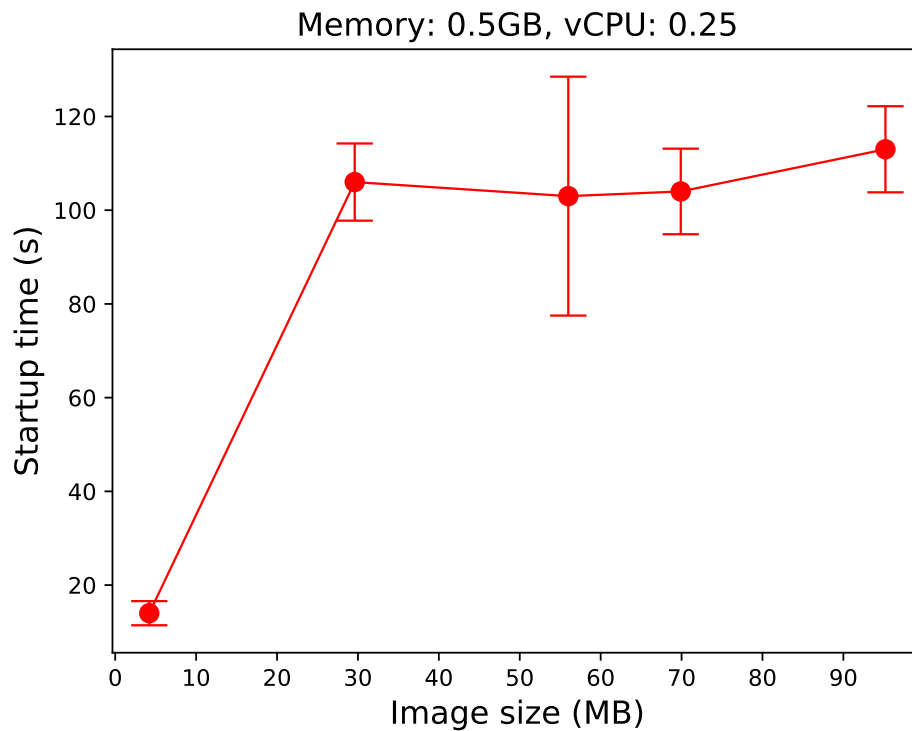


Figure 7.1: AWS ECS Fargate startup times for containers with different image sizes using a task definition with 0.5GB memory and 0.25 vCPU.

The smallest image size took little over 10 seconds to start. A large leap in startup time is then taken when the image size goes from 4.21MB to 29.6MB and goes from approximately 15 seconds to over 100 seconds. A plateau can be seen for each image with a size of over 30MB. The image size can be discarded after this plateau has been reached, and the startup time can in some cases be seen to decrease for larger image sizes.

The sudden jump in startup time clearly shows that the image size has an important role in how fast AWS ECS with Fargate can launch a container. It is however not possible to pinpoint exactly where the tipping point is from the obtained figure, so the conclusion is that the image size should not exceed 4MB, unless startup time is not critical.

The second experiment was configured to use a task definition with *1GB memory* and *0.5 vCPU* and the same five images. The following results were obtained:

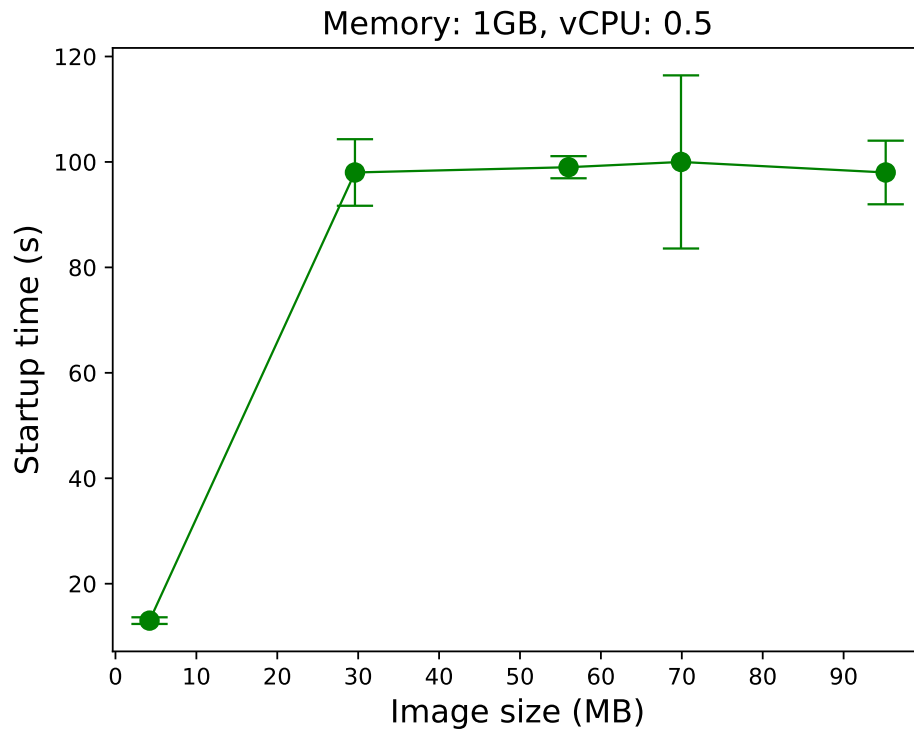


Figure 7.2: AWS ECS Fargate startup times for containers with different image sizes using a task definition with 1GB memory and 0.5 vCPU.

A very similar result was obtained compared to the first experiment. The first image took little over 10 seconds to start and the same plateau is reached when the image size exceeds 30MB. However, the increase in memory and vCPU count seems to have sped up the plateau and is closer to the 100 seconds mark than in the first experiment. This change can be considered negligible due to the long startup times while exceeding an image size of 30MB.

The image with size 69.9MB experienced a larger standard deviation. This happened because a single measurement took approximately 40% longer than the rest. This shows that the startup time with AWS ECS Fargate can be inconsistent, so an achieved startup time should not be relied upon.

The final experiment was configured to use a task definition with *2GB memory* and *1 vCPU* and the same five images. The following results were obtained:

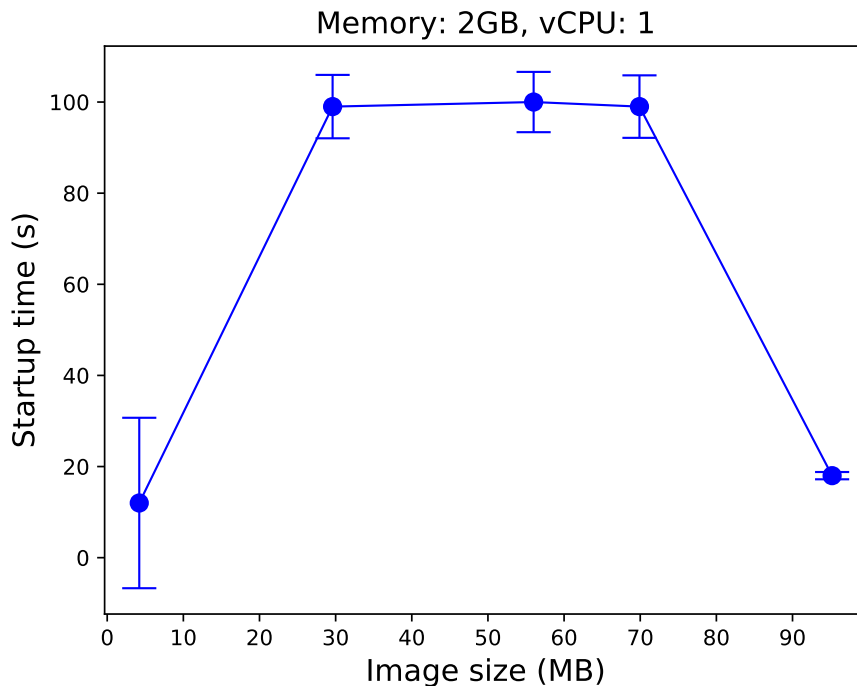


Figure 7.3: AWS ECS Fargate startup times for containers with different image sizes using a task definition with 2GB memory and 1 vCPU.

Much like the two previous experiments we can see that the smallest image has a startup time below 20 seconds, followed by a plateau. The increase in memory and vCPU count seems to make the startup time more stable at this plateau.

An exception happened during the final experiment where the largest image with a size of 95.2MB was almost as fast as the first, smallest image. It was believed that a configuration error was made, so the data point was re-calculated. However, the same result was always obtained. A possible explanation to this behavior is that AWS ECS stores the image in cache so that the container can load and start faster. AWS ECS can be configured to cache large images.¹ – this was however not configured in the experiment so a probable cause is that it has been cached automatically.

1. "Amazon ECS Adds Options to Speed Up Container Launch Times"
<https://aws.amazon.com/about-aws/whats-new/2018/05/amazon-ecs-adds-options-to-speed-up-container-launch-times/>

7.4 Experiment: Summary

The experiments has shown the startup time for containers using the AWS ECS Fargate orchestrating service. The goal was to find which task definition results in the fastest container startup time for varying image sizes, and if the service could be incorporated in the VisualBox service for use with running integration modules. The obtained results shows that the image size plays a significant role in how fast AWS ECS Fargate will launch a container. There seems to be a tipping point where the container startup time tenfolds, and stabilizes on a plateau of approximately 100 seconds.

It was also shown that the startup time can be drastically decreased if images are cached. VisualBox can however not rely on caching if a miss could result in a 100 second startup time.

Due to the nature of AWS ECS Fargate, the inner workings of cluster management and worker node provisioning is hidden. The purpose of the Fargate launch type is to relieve the developer from these tasks. This apparently comes with a cost of longer startup times and leaves room for improvement by AWS.

The same images were for contrast tested on a local computer running a single-node Kubernetes cluster, where startup times were constantly in the sub-five seconds and almost solely relied on the time it took to download an image from an image repository.

The shortcomings with relatively long startup times for AWS ECS Fargate influenced the choice to move integration and data processing to a "plain" Kubernetes solution. The Kubernetes cluster setup becomes more complex but container startup times are at an acceptable level of sub-five seconds.

7.5 Distributed Arctic Observatory (DAO)

Since VisualBox is designed as a SAAS, part of the evaluation is to create a Minimum Viable Product (MVP) and test it in practice. The Distributed Arctic Observatory (DAO) research group at UiT was working on IOT devices to be stationed in arctic conditions. It was desired to constantly be able to monitor the internal sensors of the device as to be notified if a device suddenly becomes defect or needs a battery change. Some of the metrics to be displayed was the internal temperature of the devices and their power usage over time. A GPS position was also reported.

These devices sent the data to a server which then eventually ended up as text files on a data dump server. The information was structured in folders separated by devices, sensors (such as temperature sensor, GPS module etc.), and text-files for each new day. These files were then accessible over a simple HTTP-based file server.

A VisualBox integration was made to take as input credentials necessary to authenticate against the HTTP-based file server. Once the integration started it would figure out the current date (and if specified, a date-range) and request the files necessary to put together all information. After each text-file was fetched, the result was merged and an output model was generated. This whole process could be configured to repeat itself at set intervals to constantly feed the dashboard with fresh data.

Since visualization widgets such as gauges, line-graphs and maps already existed in the crowdsourced registry, they could be used to plug into the generated data model, and a dashboard with device metrics was quickly built as seen in **Figure 7.4**.



Figure 7.4: VisualBox in use by the Distributed Arctic Observatory (DAO) research group. The middle screen is a VisualBox dashboard visualizing metrics reported by two separate IOT devices.

/ 8

Discussion and Future Work

8.1 Multi-stage Docker Image Builds

Integrations that are used to fetch and process data in a dashboard are written by users that may choose which runtime the code is going to run in. To get this to work, a separate Docker image is built for each runtime with required binaries and packages. E.g. for the Python 3 runtime to work, Python 3 and pip3 has to be installed once the container starts.¹ This also means that if a compiled language runtime is used it has to compile the integration code each time a new container starts. This can greatly reduce container startup time and the disk space needed for the integration to run.

With the introduction of multi-stage Docker builds, a pre-build step can be used to produce Docker images that can then be stored in an image-registry before a container is launched. A multi-stage Docker build allows the usage of multiple, separate base images at each build stage that can be used to introduce additional custom functionality from other base images. This is something that is currently not possible in the implemented VisualBox version.

1. pip3 - the Python 3 package installer.

Controlling the build process would only require a Dockerfile to be included in the project source code. More disk space is required by building a separate Docker image for each unique integration, but a pre-built Docker image can be configured to compile, install and prepare all necessary resources for an integration before it is started and would reduce the container startup time.

If an integration has to be pre-built before it is used, the code would also be bundled inside the produced Docker image. This would eliminate most tasks performed by the LFP, as there is no longer a need for the bootstrapper to pull down the source code of the integration during the bootstrapping phase. This could also reduce the size and complexity of the bootstrap binary, which is included in each integration runtime to allow for communication with the VisualBox system.

AWS CodeBuild could be used to automatically build the integration source code into a Docker image that is then updated in an image-registry. The leap is then not far to make the AWS CodeBuild pipeline trigger on external events, such as GitHub pull-requests, and would act as a CD pipeline, completely bypassing the VisualBox client application. This would make it easier to develop integrations locally before deploying them to the VisualBox pre-build service.

The question then becomes; when should an integration be built? If an integration project is uploaded to the VisualBox system without being pre-built it would not work when added to a dashboard. The user must explicitly command the project to be built by e.g. a "build" button in the client application. Extra care must be taken by the integration explorer as to not show integrations that are not pre-built. The same goes for an integration being published to the registry. An integration must be pre-built before it can be published and shared with other users, as to avoid adding it to dashboards when an image is non-existing.

8.2 Warm Containers

Another way of achieving faster container startup times is to keep a pool of already running containers, or "warm" containers. When a user needs a container for data processing, a check can be made to see if a warm container already exists, and assign it to the task at hand.

However, if the multi-stage docker image build strategy is to be implemented as described in the previous section, it would mean that almost no container

images are generic. This then makes it hard to keep a pool of warm container since few containers are similar, and the probability of a user requiring the same container is slim.

One could argue that if a specific crowdsourced and public integration got extremely popular, and a supporting analytical system could follow the use of certain container images, a ranking could be made for the most used container images to be put into a pool of warm containers. This could speed up container startup times for the top most used integrations, even if they are uniquely built.

8.3 gVisor

Some security experts have raised their voices on the practice of running arbitrary, untrusted code in containers and simply believing that the host system is completely isolated from attacks [39]. The argument is that since a container is able to directly communicate with the host kernel and major Linux kernel subsystems are not namespaced, an attack can be made and the host system can be owned. The broad surface area of the kernel that is exposed to a container is a great attack vector for a container with malicious intent.

In response to these claims, Google introduced gVisor [40], which is a container sandbox that offers greater isolation between the host system and containers while being more lightweight than a Virtual Machine (VM).

gVisor runs as an unprivileged process that intercepts system calls made by the application. It is compatible and integrates with Docker and Kubernetes, and is definitely a security measure that must be implemented in VisualBox to further isolate integration code.

8.4 Securing Sensitive User Data

An issue with utilizing the configuration model (see **Section 5.3 – Configuration Data Model**) is the use of the *password*-type. Each value that is provided by a user that is interacting with the rendered HTML form must be stored on the back-end. It is never a good idea to store passwords in clear-text format, ever. Ideally, the VisualBox system should never permanently store sensitive configuration data.

One alternative is to never store the input of a *password*-type in a database. Instead, the client application could notice when a password-field is being used, and choose to store the password temporarily in local storage on the client side. This would in turn mean that the values for password-fields are never guaranteed to be present whenever a dashboard is opened, but the user would instead have to re-type the password whenever it disappears, and is definitely a better alternative than storing the password in clear text in a database.

Another alternative is to utilize encryption services provided by e.g. AWS, such as AWS Secrets Manager [41]. AWS Secrets Manager encrypts secrets (such as password) and can decrypt them upon an authenticated request from an application. Automatic secret rotation will add additional protection at the cost of more complexity.

8.4.1 Initial Configuration Data Model

The initial parameters sent to a container that is launched in Kubernetes includes the configuration data model, the unique ID for the instance session, integration ID and the integration version. The initial configuration data model is included as environment variables in the container when it is launched where they are read by the bootstrapper binary. By including the initial configuration data model during the bootstrap phase of a container, it can pull the source code and start it at once without having to communicate with the client application and ask for the configuration data model. However, when initially sending the configuration data model as environment variables they can potentially be later discovered in Kubernetes-specific log files of the node that is running the container.

It is not desired to store the configuration data model as environment variables at any given time. This means that the bootstrapper would have to ask for the client application to send over the initial configuration data model once the communication channel has been established. This could potentially increase the total container startup time, but would keep sensitive configuration data models (such as passwords) from getting stored on the back-end servers in the form of log-files.

8.5 Sharing Dashboards

An important aspect of the dashboard builder is to be able to share data visualizations with external users. An easy solution would be to export an

image of the current state of the dashboard. However, if a dashboard is to be displayed at e.g. a public display screen, or to be opened by someone with another account, the integration configuration should be protected. This is because integration configurations may contain sensitive information such as credentials and passwords.

Fortunately, this problem is almost already solved due to the nature of how containerized integrations communicate with the client web application. The only "link" between a dashboard and its integrations is through the socket server. If a dashboard could be launched without specifying which or how integrations run, the user viewing the dashboard is never exposed to sensitive configuration data (**Figure 8.1**).

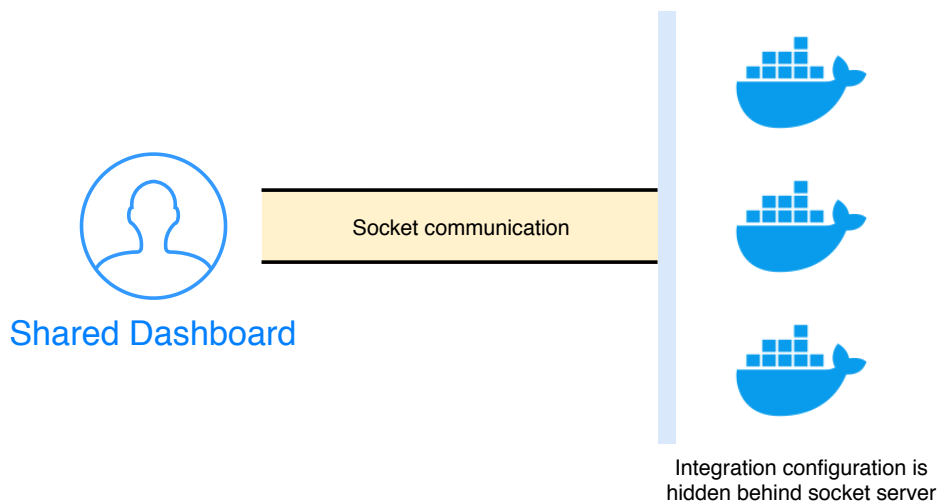


Figure 8.1: A socket communication is the only link between the client web application and the VisualBox back-end Kubernetes cluster, and hides away any sensitive information that may have been used while generating data.

When a dashboard is to be shared, a unique URL link can be generated with an randomized ID that identifies which integrations to include in the shared dashboard and how they are configured. When the URL link is visited, the correct containers are launched and a socket communication is established, all without exposing any information on how or where the data is obtained in the dashboard.

/9

Conclusion

Safely executing arbitrary user generated code in containerized environments can be done in many different ways; web workers in a web browser environment on the client-side; in a Function as a Service (FaaS) such as Kubeless, Fission or AWS Lambda; AWS ECS Fargate or in Docker containers orchestrated by a Kubernetes cluster. All of these alternatives has been explored while finding the most suited to be used in a system of crowdsourced data processing modules. The most flexible solution is to use a Kubernetes cluster combined with a control program that bootstraps a container into its working state.

We've seen how the architectural model of the VisualBox system utilizes integration and widget modules that can be combined in different ways to create visualization dashboards for different data models, and how they can be configured by taking user input via a configuration data model.

By offering a platform for crowdsourced modules to be written, tested and shared directly from the web browser, the creation of data visualizations can be made even without any technical experience.

The Serverless Framework provides a normalized language for specifying cloud resources in different cloud providers, and makes it easy to deploy and update cloud architectures. A collection of cloud resources can be deployed as a coherent stack and by dividing related functionality into separate stacks only parts of a larger cloud infrastructure can be updated.

AWS ECS with the Fargate launch type can be a good alternative for some applications to run containerized workloads without the need to worry about cluster management. The container startup times are however too slow for the VisualBox application. An interesting behavior was discovered with AWS ECS Fargate, where the startup time would plateau after a certain image size was exceeded.

A successful MVP of the VisualBox system was launched and evaluated in the field for monitoring IOT sensor metrics in cooperation with the DAO research group at UiT where internal temperature, battery drainage and geographical position was visualized in a dashboard and displayed on a large screen.

As we've seen there are many components involved when building a complete SAAS, but as long as the number of cellular IOT connections keep rising, and data is becoming ever so more available, VisualBox may just be a small catalyst in solving a much greater challenge at hand.

Bibliography

- [1] Ericsson Mobility Report, p. 16, June 2018. [Online]. Available: <https://www.ericsson.com/assets/local/mobility-report/documents/2018/ericsson-mobility-report-june-2018.pdf>
- [2] O. Liberg, M. Sundberg, Y.-P. E. Wang, J. Bergman, and J. Sachs, “Cellular Internet of Things - Technologies, Standards and Performance.” Academic Press, 2008, ch. 8, pp. 316–318.
- [3] Grafana Labs, “The open platform for analytics and monitoring,” 2019 (accessed May 10, 2019). [Online]. Available: <https://grafana.com/>
- [4] Telenor Start IoT. (2019). [Online]. Available: <https://startiot.telenor.com/>
- [5] Telenor Connexion, “Managed IoT Cloud,” 2019 (accessed May 9, 2019). [Online]. Available: <https://www.telenorconnexion.com/managed-iot-cloud/>
- [6] NPM Inc., “the heart of the modern development community,” 2019 (accessed May 9, 2019). [Online]. Available: <https://www.npmjs.com/>
- [7] Node.js®, “JavaScript runtime built on Chrome’s V8 JavaScript engine,” 2019 (accessed May 14, 2019). [Online]. Available: <https://nodejs.org/>
- [8] “European Computer Manufacturers Association (ECMA),” 2019 (accessed May 10, 2019). [Online]. Available: <http://www.ecma-international.org/>
- [9] I. van Hoorne, “CodeSandbox: Online Code Editor Tailored for Web Application Development,” 2019 (accessed May 6, 2019). [Online]. Available: <https://codesandbox.io/>
- [10] StackBlitz, “The online code editor for web apps. Powered by Visual Studio Code.” 2019 (accessed May 6, 2019). [Online]. Available:

<https://stackblitz.com/>

- [11] S. Aarsaether and P. H. Borgen, “Scrimba,” 2019 (accessed May 6, 2019). [Online]. Available: <https://scrimba.com/>
- [12] E. Simons, “Introducing Turbo: 5x faster than Yarn & NPM, and runs natively in-browser ,” 2017 (accessed May 7, 2019). [Online]. Available: <https://medium.com/stackblitz-blog/introducing-turbo-5x-faster-than-yarn-npm-and-runs-natively-in-browser-cc2c39715403>
- [13] Amazon Web Services (AWS) Inc., “AWS Lambda – Serverless Compute,” 2019 (accessed May 7, 2019). [Online]. Available: <https://aws.amazon.com/lambda/>
- [14] Rollup.js, 2019 (accessed May 10, 2019). [Online]. Available: <https://rollupjs.org/>
- [15] MDN Web Docs, “Cross-Origin Resource Sharing (CORS),” 2019 (accessed May 9, 2019). [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>
- [16] B. Grinstead and A. Marasco, “videoconverter.js,” 2019 (accessed May 24, 2019). [Online]. Available: <https://bgrins.github.io/videoconverter.js/>
- [17] Amazon Web Services (AWS) Inc., “Amazon Elastic Container Service,” 2019 (accessed May 10, 2019). [Online]. Available: <https://aws.amazon.com/ecs/>
- [18] —, “Amazon EC2,” 2019 (accessed May 10, 2019). [Online]. Available: <https://aws.amazon.com/ec2/>
- [19] —, “AWS Fargate,” 2019 (accessed May 10, 2019). [Online]. Available: <https://aws.amazon.com/fargate/>
- [20] Kubeless, “The Kubernetes Native Serverless Framework,” 2019 (accessed May 10, 2019). [Online]. Available: <https://kubeless.io/>
- [21] Kubernetes, “Production-Grade Container Orchestration,” 2019 (accessed May 10, 2019). [Online]. Available: <https://kubernetes.io/>
- [22] Fission, “Serverless Functions for Kubernetes,” 2019 (accessed May 10, 2019). [Online]. Available: <https://fission.io/>
- [23] E. You, “Vue.js - The Progressive JavaScript Framework,” 2019 (accessed

- May 3, 2019). [Online]. Available: <https://vuejs.org/>
- [24] Docker Inc. (2019) Docker: Enterprise Application Container Platform. [Online]. Available: <https://www.docker.com/>
- [25] VisualBox, “Integration Environments,” 2019 (accessed April 30, 2019). [Online]. Available: <https://docs.visualbox.io/integrations/#environments>
- [26] T. Preston-Werner. (2019) Semantic Versioning 2.0.0. [Online]. Available: <https://semver.org/spec/v2.0.0.html>
- [27] Microsoft. (2019) Monaco Editor. [Online]. Available: <https://microsoft.github.io/monaco-editor/index.html>
- [28] ——. (2019) Visual Studio Code. [Online]. Available: <https://code.visualstudio.com/>
- [29] Amazon Web Services (AWS) Inc., “Infrastructure as Code,” 2017 (accessed April 25, 2019). [Online]. Available: <https://d1.awsstatic.com/whitepapers/DevOps/infrastructure-as-code.pdf>
- [30] —, “AWS CloudFormation,” 2019 (accessed May 9, 2019). [Online]. Available: <https://aws.amazon.com/cloudformation/>
- [31] —, “Amazon DynamoDB,” 2019 (accessed May 7, 2019). [Online]. Available: <https://aws.amazon.com/dynamodb/>
- [32] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: amazon’s highly available key-value store,” *SOSP*, pp. 205–220, 2007.
- [33] Amazon Web Services (AWS) Inc., “Amazon Simple Storage Service (Amazon S3),” 2019 (accessed May 7, 2019). [Online]. Available: <https://aws.amazon.com/s3/>
- [34] Serverless Inc., “The Serverless Application Framework powered by AWS Lambda, API Gateway, and more,” 2019 (accessed May 7, 2019). [Online]. Available: <https://serverless.com/>
- [35] Amazon Web Services (AWS) Inc., “Amazon API Gateway,” 2019 (accessed May 7, 2019). [Online]. Available: <https://aws.amazon.com/api-gateway/>

- [36] —, “Simple and Secure User Sign-Up, Sign-In, and Access Control,” 2019 (accessed May 9, 2019). [Online]. Available: <https://aws.amazon.com/cognito/>
- [37] Socket.IO, “Realtime application framework (Node.JS server),” 2019 (accessed May 10, 2019). [Online]. Available: <https://socket.io/>
- [38] Algolia, “Fast, reliable and modern search and discovery,” 2019 (accessed May 7, 2019). [Online]. Available: <https://www.algolia.com/>
- [39] D. J. Walsh, “Are Docker containers really secure?” 2019 (accessed May 29, 2019). [Online]. Available: <https://opensource.com/business/14/7/docker-security-selinux>
- [40] Google, “gVisor - A container sandbox runtime focused on security, efficiency, and ease of use.” 2019 (accessed May 29, 2019). [Online]. Available: <https://gvisor.dev/>
- [41] Amazon Web Services (AWS) Inc., “Amazon Secrets Manager,” 2019 (accessed May 13, 2019). [Online]. Available: <https://aws.amazon.com/secrets-manager/>