BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

CHART BUILDER FOR ANSIBLE AUTOMATION ANALYTICS

VIZUALIZACE DAT PRO ANSIBLE AUTOMATION ANALYTICS

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **LEVENTE BERKY**
AUTOR PRÁCE

SUPERVISOR **doc. Mgr. ADAM ROGALEWICZ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2020/2021

Master's Thesis Specification

24178

Student:	**Berky Levente, Bc.**
Programme:	Information Technology
Field of study:	Bioinformatics and Biocomputing
Title:	**Chart Builder Ansible Automation Analytics**
Category:	Information Systems

Assignment:

1. Study the Open API definition and the Ansible Automation Analytics (AAA) tool with a special attention on client-server communication API.
2. Propose a way of processing a data from client-server communication of AAA tool. The intention is a data visualization (by various type of chars) with a subsequent automated processing of the visualized data.
3. Design a plugin into the client part of AAA, which allows one to visualize data from client-server communication for a user as well as for a machine processing. A user can customize the visualization using various filters and aggregations.
4. Implement the proposed plugin.
5. Validate the plugin from point 5 using unit tests and demonstrate that your solution can be used as an anomaly detection for a computation cluster managed by AAA.
6. Discuss possible improvements and future directions of your solution.

Recommended literature:

- Ansible Automation Analytics project: https://www.ansible.com/products/automation-analytics
- Open API documentation: https://swagger.io/specification/

Requirements for the semestral defence:

- Items 1 and 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Rogalewicz Adam, doc. Mgr., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2020
Submission deadline:	May 19, 2021
Approval date:	November 11, 2020

Abstract

This thesis focuses on creating a web component to render charts from a structured data format (schema for short), and creating an user facing interface for editing and creating the schema for the Ansible Automation Analytics. The thesis explores the current implementation of the Ansible Automation Analytics and the corresponding API, researches similar charting libraries and describes the fundamentals of used technologies. The practical part describes the requirements for the component and describes the development and the implementation of the plugin. Furthermore, the thesis describes the process of testing and the future development plans of the plugin.

Abstrakt

Tato práce se zaměřuje na vytvoření webové komponenty k vykreslení grafů ze strukturovanýho datovýho formátu (dále jen schéma) a vytvoření uživatelského rozhraní pro editaci schématu pro Ansible Automation Analytics. Práce zkoumá aktuální implementaci Ansible Automation Analytics a odpovídající API. Dále zkoumá vhodné knihovny pro vykreslování grafů a popisuje základy použitých technologií. Praktická část popisuje požadavky na komponentu a popisuje vývoj a implementaci pluginu. Dále práce popisuje proces testování a plány budoucího vývoje pluginu.

Keywords

web, web component, web application, data visualization, chart rendering, React framework, Ansible Automation Analytics

Klíčová slova

web, web komponenty, web aplikace, vizualizace dat, vykreslování grafu, React framework, Ansible Automation Analytics

Reference

BERKY, Levente. *Chart Builder for Ansible Automation Analytics.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Mgr. ADAM Rogalewicz, Ph.D.

Rozšířený abstrakt

Tato práce se zabývá tvorbou webové komponenty pro vytváření grafů ze strukturovaného datového formátu (dál jen schéma) a pro poskytnutí webového UX pro editaci schématu. Komponenta byla vytvořena pro project Ansilble Automation Analytics (dál jen AAA), který je vyvíjen společností RedHat.

Nejprve bylo zapotřebí stanovit technologie, které budou při vývoji využity. Nejznámější je framework React, který je používán i v AAA a podporuje vytváření komponent. Proto je nejjednodušší na integraci do AAA, a proto tato práce bude využívat tento framework. Dále využívá TypeScript pro statickou kontrolu typů, který zjednodušuje manuální vytvoření schématu. TypeScript je JavaScriptová nástavba kompilovaná do JavaScriptu. Výhodou je, že zavádí typy do JavaScriptu a upozorní na typovou nesprávnost při kompilaci, což zvyšuje kvalitu kódu a usnadňuje ladění.

Před vývojem byly prozkoumány AAA a AAA Application Programming Interface (AAA API). AAA API používá OpenAPI standard a poskytuje data pro AAA. Tato práce využívá data z tohoto API. Největší důraz byl kladen na formát dat, na které má vliv seskupování dat podle času nebo podle jiných parametrů. Dále byly přezkoumány parametry pro každý koncový bod API. API rovněž poskytuje koncové body pro validní nastavení datových koncových bodů.

Před vývojem byly dále prozkoumány další aplikace pro vizualizaci dat: Grafana a Chartlog a byly stanoveny požadované funkce pro stavbu grafů. Tyto aplikace pro vytváření grafů by měly být schopny vytvořit skupované, skládané, výsečový, čárové, sloupcové, bodové a plošné grafy, přičemž datový formát bude limitovat výběr grafů. Formulář pro editaci by neměl nechat uživatele vybrat nevalidní konfiguraci nebo konfigurace které nedávají smysl.

Pro vykreslení samotných grafů byly prozkoumány knihovny které vytvářejí grafy. Důraz byl kladen na jednoduchost použití, poskytnuté funkce a činnost vývojových týmů. Nakonec byl vybrán Victory Charts kvůli modularitě komponentů, pohotovosti týmu a velkému výběru funkčnosti.

Victory Charts poskytuje komponenty bez stylingu. Jelikož se práce integruje do AAA musel být vytvořena styl kompatibiliní s AAA. Dále byla využita knihovna Patternfly Charts, která je tenkou vrstvou nad Victory Charts a nastavuje style komponenty aniž by limitoval anebo změnil jejich funkcionalitu.

Před samotným vývojem bylo třeba nastavit vývojové prostředí. Pro vývoj byl použit verzovací nástroj git, který je hostován na GitHub platformě. Komponenta je distribuována jako knihovna pomocí NPM pod jménem 'react-data-explorer'. Scematic-release je používán pro aktualizaci dané verze a NPM knihovny po změnách v kódu. Dále práce obsahuje demo aplikaci, která je hostována lokálně pomocí nástroje webpack. Demo aplikace se využívá na vývoj a jako ukázkový kód.

Výsledná knihovna obsahuje tři hlavní komponenty. První a nejdůležitější komponenta je schopna ze schématu vytvořit grafy. Schéma je limitováno na základní datové typy, které jsou serializovatelné. Tato limitace je dána tím, že schéma bude ukládáno i v databázi. To znamená že když je potřeba ukládat funkci v schématu, pak se ukládají jen jména funkcí, a samotné funkce jsou předány jako další parametr komponenty. Schéma je plochá struktura, aby sa dalo zpracovat a editovat snadněji. Komponenta obdrží data z API a transformuje je tak aby se dala používat v grafech. Komponenta je schopna vytvořit požadované grafy: skupinový graf, skládaný graf, výsečové grafy nebo jednoduché grafy. Dále se ve schématu dá přizpůsobit každá část grafu. Komponenta podporuje popisky dvěma různými způsoby: buď jako popisek k jednotlivým datovým bodům, nebo jako

popisek ke všem bodům na svislé ose. Grafy podporují různé události, například kliknutí na určitou část grafu. Komponenta také poskytuje jednoduchý způsob pro přidání legend ke grafům včetně interaktivity, která umožňuje schování jednotlivých datových sérií.

Druhá komponenta nabízí formulář pro editaci schématu. Důležitou částí formuláře byl požadavek, aby se uživatel za žádných okolností nedostal do neplatné konfigurace. Formulář proto získává možnosti pro některé parametry z API a na jejích základě v každé konfiguraci zabezpečí správnost. Možnosti pro některé parametry z API ještě nejsou dostupné (například typy grafů). Uživatelské rozhraní editoru se skládá ze dvou částí: vytvořeného grafu a formuláře. Uživatel po nastavení hodnoty ve formuláři rovnou uvidí změny v grafu.

Třetí komponentou je panel grafů která používá dříve popsané dvě komponenty a přidává dvě tlačítka pro otevření a zavření editoru. Každá komponenta sa dá používat samostatně.

Unit testy byly provedeny pomocí knihoven Jest, fetch-mock a react-testing-library. Tyto nástroje umožnily modelovat odpovědi z API a testovat vyrendrované grafy pomocí snapshotů. Unit testy byly provedeny hlavně na komponentě, která vytváří grafy. Vývoj dalších dvou komponent dále pokračuje.

Tento další vývoj zahrnuje návrhy a implementace vylepšeného API pro grafy, UX návrh pro formulář pro panel grafů. Během vývoje se v knihovně Patternfly Charts našly dva problémy, a bylo navrhnuto její vylepšení. V budoucnosti se tato práce bude dále vyvíjet tak, aby se daly zpracovat i datové streamy pro živé data. Knihovna bude dále rozšířena o podporu 3D grafů a plně integrován do AAA.

Chart Builder for Ansible Automation Analytics

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Rogalewicz Adam, doc. Mgr., Ph.D. The supplementary information was provided by Ing. Ben Thomasson. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Levente Berky
May 12, 2021

Acknowledgements

I wish to express my thanks to my supervisor Rogalewicz Adam, doc. Mgr., Ph.D. and Ing. Ben Thomasson who helped to create a more professional work. I also want to express my gratitude to Kia Lam, the Ansible Automation Analytics team and Red Hat for their professional technical help and suggestions while programming the component.

Contents

Chapter 1

Introduction

In the last decade, the World Wide Web has become a remarkably huge system for storing information. People use it all around the world daily. The browser is the main gateway to the web and it has become more and more sophisticated over time. The web is used for creating documents, playing games, watching videos, or running different programs on far away and powerful computers. Thus, the data which is collected from the web increases exponentially.

As more and more data is collected, it is increasingly important to understand how this data is represented to the user on those web pages. Presenting a huge amount of data to the user via graphs and charts is one of the most efficient and used forms. We can find charts everywhere on the web, from flight prices to displaying which pages people tend to spend the most amount of time. Because of this, charts are a crucial part of the user experience.

This work focuses on creating a chart dashboard plugin for a Red Hat Insights for Red Hat Ansible Automation Platform system (referred to as Automation Analytics for simplicity) and a user friendly chart editor. The charting library will provide the charts for the dashboard. The dashboard will allow users to display a huge amount of data by visualising it in different types of charts. The plugin will allow the user to select, aggregate, and then display the data in various charts. After the initial setup, the dashboard will allow the users to have all of their important information displayed at glance in the form of charts.

It is important to be able to test the generated charts. Testing generated vector images is not a trivial task. Therefore, this work will develop a charting library which allows the developers to automate processing the generated charts with ease. This functionality is crucial for writing unit tests and for test driven development.

The thesis first describes the tools used and the background theory for web development. Next, Automation Analytics and the Automation Analytics API will be discussed. The following chapter introduces the chart, and describes the research done on the charting libraries and the chart dashboards.

The last three chapters describes the implementation of the plugin, including the unit tests and the future development.

Chapter 2

Utilized tools

This chapter aims to introduce the tools used and the technologies required to understand the program. The chapter will not describe the JavaScript programming language or any of its revision.

2.1 Theory Basis

In 1991, the Hypertext Transfer Protocol (HTTP) foundation of the World Wide Web (WWW) was introduced. Not long after, the Hypertext Markup Language (HTML) was introduced. HTML elements are the building blocks of a structured document. These elements and their nesting allowed the creation of the first static pages. With the introduction of Cascading Style Sheets (CSS), the ability to style the web pages has been extended. However, the pages were still static and served rather representing a document rather than offering any interactions with the user.

In 1995, Netscape, the dominating browser at the time, developed Mocha script. This language, after a few changes, is currently known as JavaScript. JavaScript is a prototype based, object oriented, interpreted scripting language. These three technologies together: HTML, CSS, and JavaScript are the core technologies of World Wide Web content engineering.

JavaScript was initially designed to write short scripts and sometimes behaved differently than the programmer would anticipate. Over time, it became the most used language; however, it also has the reputation of being the worst programming language in existence.

```javascript
let i = 1;
i + "1" = "11";
i - "1" = 0;
[] + [] = ""
[] + {} = "[object Object]"
```

Listing 2.1: Counter intuitive JavaScript example

JavaScript has no integer type. All numeric values are double precision floating points (represented as numbers). This can cause unexpected behaviors when comparing two numbers. Also, the type NaN (which is not a number) is represented as a number. The loose typing also can cause problems. Just to name a few: concatenating two empty arrays results in an empty string. Accessing an empty array element does not throw an error but returns „undefined“. The operator + and the operator - results in different types of values when used with integer and string listing 2.1. JavaScript as C or Java uses semicolons to divide lines. Not using a semicolon at the end of the line is not a problem, however, as it fills them in. But in doing so, this can cause subtle bugs in the code.

JavaScript is highly dependent on global variables. Implied global variables are the most problematic of them all. When JavaScript doesn't find a name for a variable assignment, it implicitly creates a new global variable. This makes it especially hard to detect typos in the code. The language does not makes debugging code any easier with silently failing.

The last problem, which is becoming more and more relevant, is the asynchronous programming. JavaScript's callback functions are messy and hard to track. The scopes of the functions are difficult to figure out. There are some new structures to solve this issue (such as the *Promise*), however, they are not perfect solutions.

As mentioned above, sometimes unexpected and counter intuitive behaviour allows the programmers to abuse the language. Therefore, libraries written in JavaScript often go against good practices. Programmers abuse the language to write workarounds for various problems. These workarounds are immensely complex and cumbersome to understand.

These problems were not so apparent before 2009. After introducing Node.js, a server side JavaScript framework, and multiple client side user interface building frameworks, it was used to manipulate smaller changes on the page and provided minimal interactions with the site. These days, it is not a surprise when the whole page is written in JavaScript. HTML and CSS have more and more become a part of the JavaScript program, rather than the JavaScript scripts being an extension to the HTML page.

When looking at maintainability, a good code structure is necessary. JavaScript is a typeless language, in that you don't have to specify what type of information will be stored in a variable ahead of time. Therefore, the developer has to understand the code very well. Most of the libraries are thoroughly documented but the code itself is poorly commented. The code should be well commented and structured in this project to be easily maintainable later on.

To help solve the maintainability problems, there are superset languages for JavaScript. The superset language means that the language compiles to JavaScript with additional features. One of the most popular of these languages is TypeScript [18] from Microsoft. TypeScript provides type checking, interfaces, and much more. On compile, it can check the code structure and create warnings if it is not the expected one. For example, the class member order can be defined or you could define warnings about logging and non-typed variables. The comments can be made as obligatory as needed. It is true that defining the comment section does not mean the code is properly commented, but at least it inspires developers to do it. As a result, anyone working with the code has to follow these rules, and they stay consistent and readable for the other developers.

2.2 Glossary

This section provides a glossary of terms and abbreviations to assist in the reading and understanding of this thesis topic.

1. HTML DOM - (referenced simply as DOM) is a standard object model and programming inter-face for HTML. It defines:
   (a) The HTML elements as objects
   (b) The properties of all HTML elements

   (c) The methods to access all HTML elements
   (d) The events for all HTML elements

2. JavaScript - is a client side scripting language which is able to manipulate the DOM.
3. JSON file - stores data in JavaScript object notation.
4. CSS - describes how HTML elements are to be displayed on screen, paper, or in other media. They consist of rules. These rules can be stored in separate CSS files.

2.3 React

React[12] is a JavaScript framework for building user interfaces and is developed by Facebook Inc. This framework is declarative and component based. React also has an effective re-rendering method.

2.3.1 State, props and Events

The state variable contains the components data. The difference between the props and the states is that the props are passed to the component and they should not change it. Unlike the state variable, which represents the state of the component, and the component can change its own state, depending on a user's input or other events. As data follows a waterfall model to synchronize the data between the components, the closest ancestor has to be found. This ancestor should store the data in question and it then becomes the „only source of truth".

Event handlers help to modify the data stored in the child components in a one way binding model and are passed to the child component. When the child component's data is changed, the passed event handler is called, which should modify the state which is passed to the child as prop in the parent component.

2.3.2 JSX

JSX is a syntax extension for JavaScript and looks like HTML but accepts JavaScript code. JSX elements are compiled to JavaScript expressions and objects. It is cheaper than creating DOM elements; therefore, in React, all modifications are made on JSX elements and React takes care of the synchronization with the DOM.

2.3.3 Rendering

It is necessary to specify at least one root element to render React content. The element rendering is done with a render function in which it is necessary to specify the JSX element (what to render) and the DOM element (where to render the defined element). When updating an element, React updates only what is necessary in the DOM. Conditional rendering is done with the help of JavaScript conditions, using if - then - else blocks. Making it easier to decide which elements to render and which to not.

2.3.4 Components

There are two types of components: functional and class components. In the latest versions of the React framework, functional components are pushed as the preferred style. Both functional and class components have the same capability. The difference is in the code style. While the class components use class methods and variables, the functional components use hooks.

Hooks are basically functions, returning variables and other functions to modifying them. The basic hooks supplied by the React framework are useState and useEffect. useState is for storing and modifying the component's state. useEffect is for hooking into the component's life cycle event and firing functions on changes.

The user has the ability to write their own hook functions, since it is a function. The user defined hooks can use other hooks as well, allowing them to bundle more complicated logic into one function. Which results in quite elegant solutions.

2.4 TypeScript

As mentioned in section 2.1, JavaScript, like other languages, has its own quirks and oddities. Since JavaScript was designed for quick uses and then grew into a full fledged programming language, there are many of these oddities.

As an example, we could show the '==' operator, which compares without type check, and throws some very interesting but mostly counter intuitive results. Most other languages would throw an error when trying to compare different types, whether at runtime or at compile time.

TypeScript[18] is trying to fix some of these problems with JavaScript by introducing types and static checking to JavaScript. At times, TypeScript can feel limiting for the programmer; however, it is a superset to JavaScript. Meaning that it can do anything that JavaScript can do. Typescript never changes the runtime behaviour of JavaScript code.

The next few sections will introduce a few crucial parts of the TypeScript language. They will describe the basic types, classes, and functions. They should also provide enough insight to understand the succeeding chapters.

2.4.1 Basic types

JavaScript[16], since ECMAScript 6, has had the ability to declare variables in four different ways. The variables can be declared without a keyword before the variable, or with one of the following three keywords: *var, let, const*. When a variable is declared without any keyword or with the *var* keyword, the variable becomes a global variable, no matter where it was declared. The *var* keyword allows users to declare a variable without an initial value.

The *let* and *const* keywords limit the scope of the variable to the current scope. They also allow the of shadowing variables from other scopes, instead of rewriting them. In addition, the *const* keyword requires the variable to be initialized with a value and prevents changes to this variable. However, it does not prevents the mutation of the variable.

The basic types in Typescript correspond to the basic types in JavaScript, with an addition of the *enumeration* type. The basic types are: boolean, number, string and array. They all work as expected. Boolean accepts true or false values. Number accepts any integer or float value, be it positive or negative. String accepts literals. For the array, the programmer has to decide what type of variables it will hold and it will only accept that type. The basic variable examples are shown in the listing 2.2.

```
let a: boolean = false;
let b: number = 42;
let c: string = "I am a string"
let d: Array<number> = [1, 2, 3];
let d2: number[] = [1, 2, 3];
```

Listing 2.2: Basic types in TypeScript

There is a more interesting type called *tuple*. It is basically a fixed sized array where each entry can be of a different type. Enumerations does not exist in plain JavaScript. The way TypeScript adds them is similarly to a map. Every name maps to a value (implicitly to a unique number in that enumeration type).

There are two additional keywords: *unknown* and *any*. The type *unknown* comes in handy when the type becomes dynamic context, like user input. The *any* allows any type to be assigned to the variable, which is the default in plain JavaScript; however, using it is not recommended.

The *void* keyword is the opposite of *any*. It signals that there is no type at all, and nothing should be returned. The types *null* and *undefined* are subtypes of all other types. Not really usable by themselves.

And last, but not least, there is the *object* keyword. Object represents non primitive types. Generally there is no need to use the *object* keyword, since the *interface* takes care most of the non primitive types.

2.4.2 Interfaces

TypeScript focuses on the shapes that the values have. This is called sometimes „structural subtyping“. This role is filled by *interfaces* in TypeScript. They are a tool for defining contracts inside the code as well as contracts external to the code.

```
interface PropType {
    requiredValue: string;
    optionalValue?: number;
    readonly value: number;
    [key: string]: any;
}
```

Listing 2.3: Interface example in TypeScript

Listing 2.3 shows a basic interface definition. It defines an object with one required key, one optional key, and allows users to add any other key. This interface can be used in the same ways as the types.

The interface also has a property labelled *readonly*. This keyword works similarly as the keyword *const* for variables. It allows the property to be initialized when the object is first created. Every other modification to this property would result in a compiler error.

There is a way to prevent arrays from mutating, too. It is achieved by creating the array with the help of *ReadonlyArray<type>*, where the type is a valid type or interface. Since this project does not use classes, this part will not cover the interfaces for classes.

Lastly is the *indexables*. It models a map where the keys can be numbers or strings. This allows for the mapping of values to these indexes. In one map it is impossible to mix keys with numbers and strings. The listing 2.4 shows an example of the indexable type.

```
interface IndexableType {
    [index: number]: string;
}
```

Listing 2.4: Indexable interface example in TypeScript

2.4.3 Functions

Functions are the fundamental building blocks in JavaScript. They build up abstractions and hide information. On top of that, React is pushing to replace all their classes with functions; therefore, the functions are used instead of classes and namespaces most of the time as well.

The TypeScript documentation focuses on the two approaches of the functions used in JavaScript. However, with ECMAScript 6[16], a modification of the second approach overtook most of the use cases. The examples and description provided here will focus mainly on the arrow functions.

```
// Without types
const funcName = (param) => {
    return param > 0;
}
// With types
const funcName = (param: number): boolean => {
    return param > 0;
}

//Shorthand with types
const funcName = (param: number): boolean => param > 0;
```

Listing 2.5: Arrow function example

Listing 2.5 demonstrates a trivial arrow function. In the typed examples, it is shown that every parameter has defined its type (one param is the type of number) and the return value of the function (boolean in this case).

```
const funcName = (param: number, optional?: string): string => {
    ...
}

// Interface for the previous function
interface FuncType {
    (param: number, optional?: string): string;
}
```

Listing 2.6: Arrow function with optional argument example

The listing 2.6 shows a function with an optional parameter. To declare the interface for that function, we use an identical declaration. After an optional parameter no other required parameters can follow. If the function can have more parameters passed to it, it can be captured with *...rest: type[]*, where the rest is the name of the array where we can access the parameters and the type is an existing type or interface.

In JavaScript, *this* is a variable that is set when a function is called. Because it is set automatically, the user always has to understand the context in which the current function is executed. In TypeScript, arrow functions allow us to bind the *this* variable to the place where it was declared – not to the context where it is used. Furthermore, the *this* type can be set in functions. This way the compiler can check if the variable has the correct type and warn about it. Note that setting the *this* variable to *void* in functions makes it unusable.

Overwriting the function is done with multiple declarations. The compiler selects the first matching functions, so declarations should go from specific to more generic.

2.4.4 Unions and Intersection Types

Intersection and union types are one of the ways in which a user is allowed to compose types. For example, a parameter can either be a number or a string. The union type can help by defining the type with a / character, as in: *string / number.* If two *interface* are used in a union, it is possible to access members of the interface which are common to all types in the union.

However, when using the union in a *switch* statement where the exact type can be derived, the other fields become accessible as well. This is particularly useful for state management and automation. It is called the discriminating unions.

To make the code more future proof, the compiler can be set to check for a strict null return. This makes sure we covered all cases of a union in a certain *switch* block. When there is no need to cover all the options, the *default* case should be used.

Intersection[18] types are closely related to union types, but they are used very differently. An intersection type combines multiple types into one. This allows the adding together of existing types to get a single type that has all the features. For example, Person & Serializable & Loggable is a type comprised of Person and Serializable and Loggable. That means an object of this type will have all members of all three types.

2.4.5 Classes

Typescript allows us to use an object oriented approach to classes where classes are inheriting functionally and objects are built up from classes. The inheritance is done with the *extends* keyword and calling the parent method is done with *super*, like with Java.

Classes have private, protected, and public instances. When nothing is explicitly defined, instances are public by default. Of course, the *readonly* option works here too. When using *readonly*, the variables have to be set in the constructor.

TypeScript supports getters and setters. The assigning and retrieving values schematics are the same with these functions, only there is a function where some advanced rules can be defined. These functions are created with the keywords *get* and *set*. When extending an abstract class, the child class must declare those abstract functions which do not have definitions.

2.4.6 Generics

Generics allow users to write a function with not one data type, but with a variety of data types. For this, there is a 'type variable' which works on types rather than variables. Once set, the type variable can be used as another data type. The type in the function argument can be set both explicitly and implicitly. Type variables can hold every kind of type which is available for normal variables.

The type variable works similarly to functions in the generic interface. When creating an interface, the type is passed. In classes, the type variable cannot be used to create static members, otherwise it works like an interface.

If there is a need to constraint a type variable, we can extend an interface or class. This ensures that properties defined in the interface will be available in the type and it will be checked on compilation. Further constraints can be placed between two type variables. As an example, the 'K' type variable has to be the 'key of T' type variable. This ensures that whatever the 'T' will be, the 'K' will contain a key from it. The only restrictions are that generics are not available for enumerations and namespaces.

2.4.7 Enumerations

Enumerations can be numeric, a string, or heterogeneous, but the last one is not recommended. The numeric value of an element is initialized with 0 when it comes first and it increments by one if it comes after an initialized value. They can be set to be computed members, but after a computed member a constant value has to be set. The string enumerations have to be explicitly initialized.

TypeScript provides reverse mapping to each enumeration, but if it is not desired then a constant enumeration can be used. Another property of constant enumeration is that it cannot have computed member. In ambient enumerations, a special case, the members are always considered as computed, but we do not have to define them at initialization, just later.

Chapter 3

Red Hat Ansible Automation Platform

This chapter introduces the Red Hat Ansible Automation Platform. The chapter also describes, in short, all the technologies which are important to understand the API which is used in the Automation Analytics client-server communication. At the end of the chapter, the Automation Analytics API is described in detail.

3.1 REST API and OpenAPI

A REST API [7] is a web service which uses the REST (Representational Transfer State) architecture to handle a request. Applications on the World Wide Web that use the REST API are called RESTful applications. REST APIs use HTTP (Hypertext Transfer Protocol) to process data requests. There are four types of request:

1. GET - to retrieve a resource from a database
2. POST - to create a resource
3. PUT - to update a resource
4. DELETE - to remove a resource

The OpenAPI Specification defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic [23].

3.1.1 OpenAPI Format and Structure

An OpenAPI document is itself a JSON (JavaScript Object Notation) object which can also be represented in a YAML („YAML Ain't Markup Language“) format. All field names are case sensitive, except when otherwise stated. The schema contains fixed fields which have a declared name and patterned fields which are declared with the help of regex (regular expression) patterns.

The OpenAPI document can be divided into multiple parts. When it is not a single file, the other parts of the document must be referenced in the *$ref*‘ object.

The OpenAPI Specification (OAS) supports all primitive data types, which are supported by the JSON Schema Specification [25]. Namely: boolean, object, array, number, and string. The OpenAPI Specification adds a *format* property to the primitive types. The format property may be any required format not specified in the OpenAPI Specification. As an example: if the field type is a „number“, but this number is a double precision floating number, the „format“ property can hold the keyword „double“ to make it explicit. Some of the defined formats in OpenAPI are byte, binary, date, date-time, and password.

A very basic example of the OpenAPI document is showed in listing 3.1.

```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Swagger Petstore
  license:
    name: MIT
servers:
  - url: http://petstore.swagger.io/v1
paths:
  /pets:
    get:
      summary: List all pets
      operationId: listPets
      parameters:
        - name: limit
          in: query
          description: How many items to return at one time (max 100)
          required: false
          schema:
            type: integer
            format: int32
      responses:
        200:
          description: An paged array of pets
          headers:
            x-next:
              description: A link to the next page of responses
              schema:
                type: string
          content:
            application/json:
              schema: $ref: "#/components/schemas/Pets"
        default:
          description: unexpected error
          content:
            application/json:
              schema: $ref: "#/components/schemas/Error"
```

Listing 3.1: OpenAPI example document

3.1.2 OpenAPI Object

The OpenAPI object is the root document object of the OpenAPI document. It can contain several fields. The required fields are:

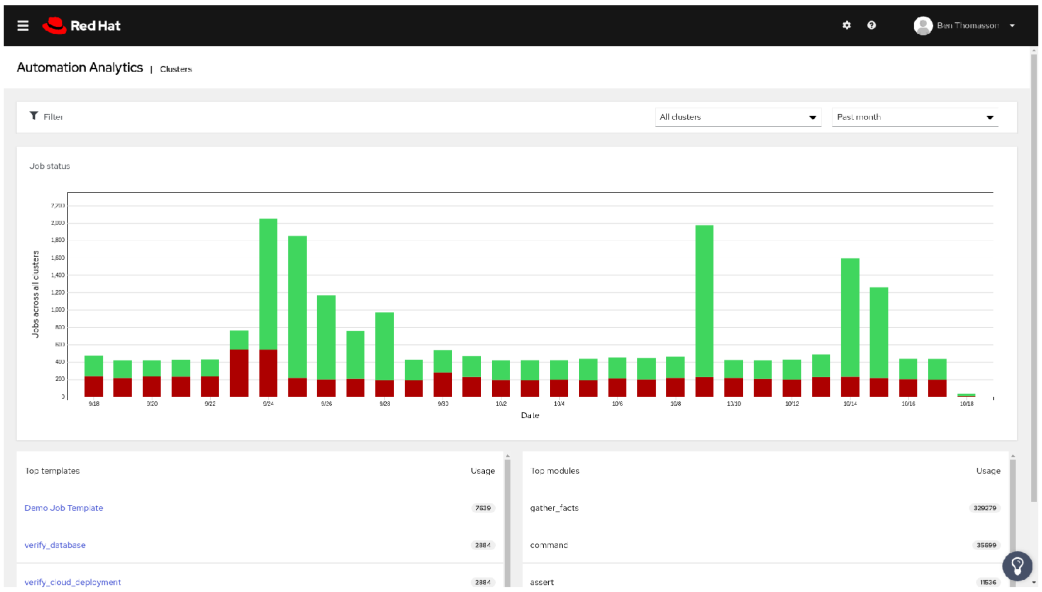
1. *openapi* - string representing the used OpenAPI semantic version number
2. *info* - an object providing the metadata of the document
3. *paths* - an object containing all the paths and operations for the API

There are other optional fields which are not listed here. Furthermore, the OpenAPI object can be extended with extensions that are described in the OpenAPI Specification.

3.2 Automation Analytics

Automation Analytics, also referred to as Red Hat Insights for Red Hat Ansible Automation Platform, is an online service provided within the Automation Platform. [20]. The Automation Platform allows for easily managing cloud infrastructures, including: systems, hosts, instances, virtual machines, containers, or devices. Automation Analytics can be used to analyze, aggregate, and report on data for the Automation Platform deployments running in the user's infrastructure.

Automation Analytics is a web application. Therefore, it runs in all four major browsers with JavaScript enabled. With the release of the version *1.1.0*, the Automation Analytics application can be divided into 4 bigger parts: visual dashboard for clusters and for organizations, health notifications, automation calculator, and the job explorer.



Top templates	Usage
Demo Job Template	7639
verify_database	2384
verify_cloud_deployment	2384

Top modules	Usage
gather_facts	329279
command	35699
assert	11536

Figure 3.1: Visual dashboard screen in AAA

Figure 3.1 shows the visual dashboard screen. The main parts of the dashboard are the filter and the chart, which changes when filtered. The bar chart shows the failed and successful job count by date through all clusters. If one specific cluster is selected in the

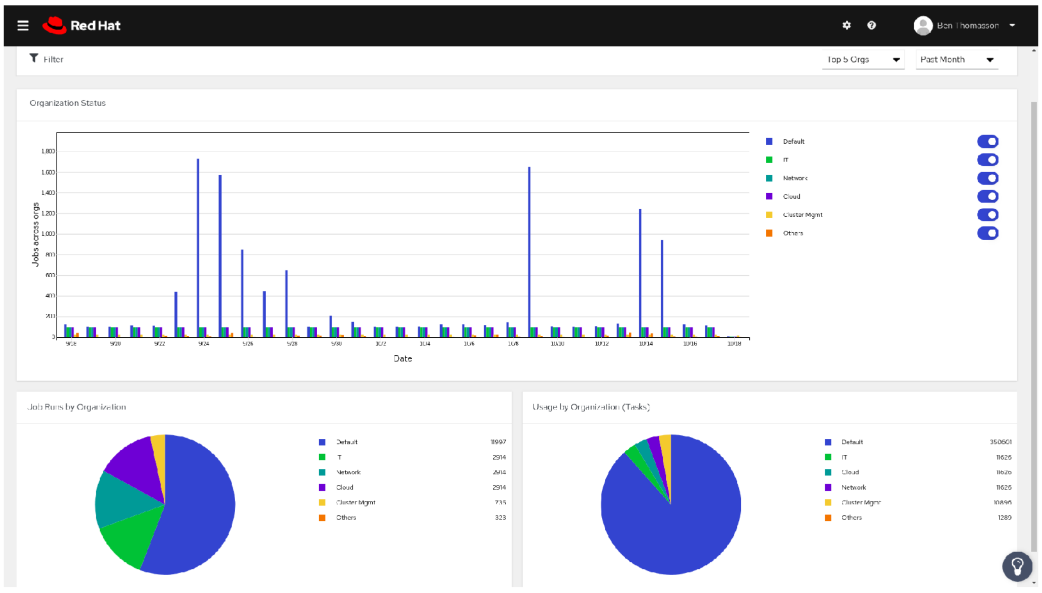
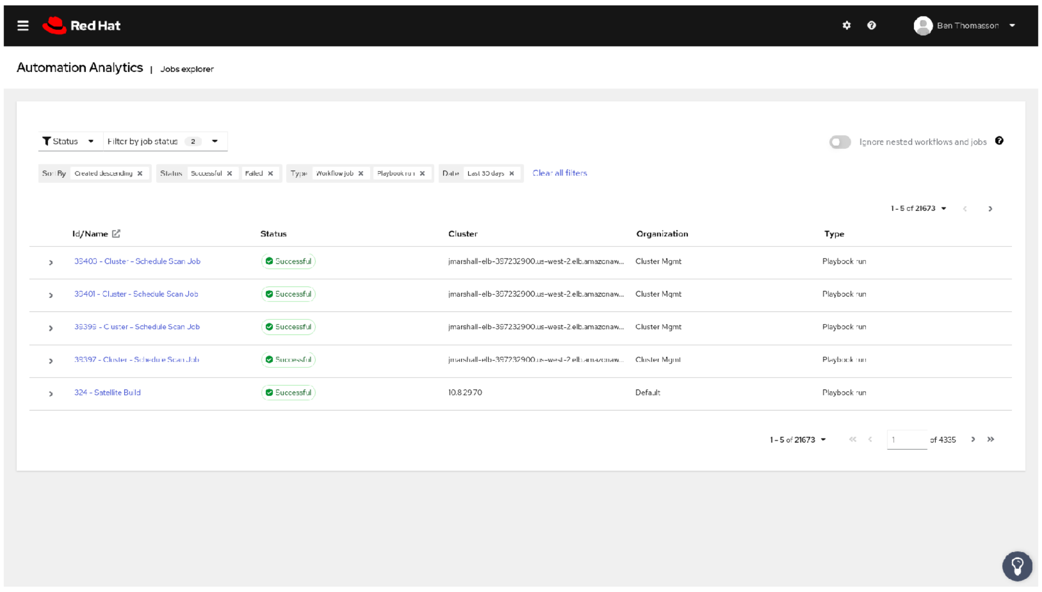


Figure 3.2: Visual dashboard for organisations screen in AAA

filter, the chart changes to a line chart with multiple lines. Each line represents successful or failed jobs.

Figure 3.2 shows the visual dashboard for organizations screen. This part shows the user different statistics across their organizations. The data is represented in pie charts and a special grouped bar chart.

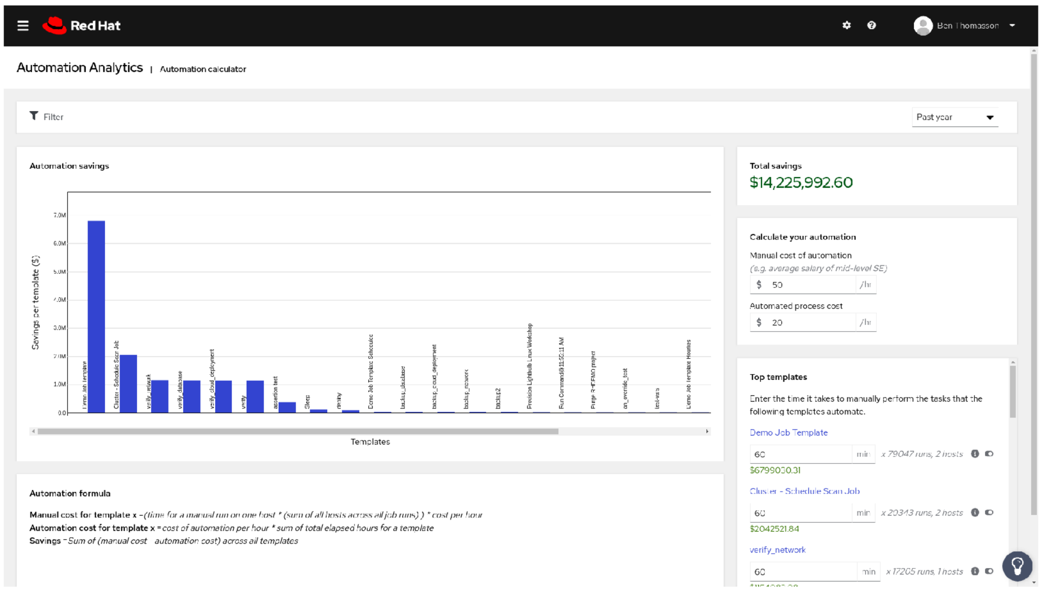

Red Hat
Ben Thomasson
Automation Analytics | Jobs explorer
Status
Filter by job status 2
Ignore nested workflows and jobs
Sort By Created descending
Status Successful Failed
Type Workflow job Playbook run
Date Last 30 days
Clear all filters
1 - 5 of 21673

	Id/Name	Status	Cluster	Organization	Type
›	33403 - Cluster - Schedule Scan Job	Successful	jmarshall-elb-397232900.us-west-2.elb.amazonaw...	Cluster Mgmt	Playbook run
›	33401 - Cluster - Schedule Scan Job	Successful	jmarshall-elb-397232900.us-west-2.elb.amazonaw...	Cluster Mgmt	Playbook run
›	33399 - Cluster - Schedule Scan Job	Successful	jmarshall-elb-397232900.us-west-2.elb.amazonaw...	Cluster Mgmt	Playbook run
›	33397 - Cluster - Schedule Scan Job	Successful	jmarshall-elb-397232900.us-west-2.elb.amazonaw...	Cluster Mgmt	Playbook run
›	324 - Satellite Build	Successful	10.8.29.70	Default	Playbook run

Figure 3.3: Job Explorer screen in AAA

Figure 3.3 shows the job explorer screen. This part of the application allows for the filtering of clusters in real-time, or through historical data, to see where specific jobs are failing or which job logs need further debugging. The queried data is displayed in a table.

The goal of this work is to allow the the data represented currently in the table to also be represented as a chart dashboard.



Total savings
$14,225,992.60

Calculate your automation
Manual cost of automation
(e.g. average salary of mid-level SE)
$ 50 /hr
Automated process cost
$ 20 /hr

Top templates
Enter the time it takes to manually perform the tasks that the following templates automate.
Demo Job Template
60 min *x 79047 runs, 2 hosts*
$6799030.31
Cluster - Schedule Scan Job
60 min *x 20343 runs, 2 hosts*
$2042521.84
verify_network
60 min *x 17205 runs, 1 hosts*

Automation formula
Manual cost for template x = *(time for a manual run on one host * (sum of all hosts across all job runs)) * cost per hour*
Automation cost for template x = *cost of automation per hour * sum of total elapsed hours for a template*
Savings = *Sum of (manual cost - automation cost) across all templates*

Figure 3.4: Automation Calculator screen in AAA

Figure 3.4 shows the automation calculator screen. This screen gives an understanding of what automation success looks like, how much time and money is saved by automating, and converts automation job costs into real-world currency.

The health notification screen displays different kinds of information about clusters. These notifications could be a warning about expiring clusters, a service outage, or too many pending jobs.

3.2.1 Ansible Automation Analytics API

The Automation Platform API specification is not publicly available yet. The API is built on the OpenAPI specification and currently offers thirteen endpoints. Some endpoints have both a POST and a GET request, bit they are the always the same. To simplify things, when both GET and POST requests are available, only the POST endpoints will be described here.

The first endpoint is the */authorized* endpoint. It takes no parameters and returns an error message if the client is not authorized to use the API. If the user is authorized, it returns an empty response and the status code 200 OK.

The other endpoints can be divided into two groups. The endpoints which return data and the endpoints which return the valid options for their respective endpoints. All the endpoints have the same query and request body parameters; however, the field values may be different. The option endpoints always provide the valid values for their parameters as well as for their corresponding endpoint for data. An example of one set of endpoints is *job_explorer*, which returns data, and *job_explorer_options*, its corresponding options endpoint.

The endpoints can have two query parameters: the *limit* and the *offset*. These two parameters allow the user to paginate in the data. The maximum limit is 25. The other parameters the endpoint accepts are passed in the request body. Such parameters are the:

org_id, inventory_id, template_id, quick_date_range, start_date, end_date, job_type, status, attributes, include_others, group_by, group_by_time and *granularity*. With the help of these parameters it is possible to filter the data, or change the grouping of the data in the endpoints for data, and they may change the valid options returned from the options endpoint.

Format of the returned data from the option endpoints

The options endpoint returns an object where the keys are the attributes, which can be passed to the request body. Each of these keys holds an array of objects, which are the options themselves. The option object consists of two attributes: the *key* and the *value*. The *key* holds the value, which the API accepts as a value in the request, while the *value* is the user friendly representation of the key. Such an example is shown in listing 3.2.

```
{
  "group_by": [
    {
      "key": "cluster",
      "value": "Cluster"
    },
    ...
  ],
  "quick_date_range": [
    {
      "key": "last_62_days",
      "value": "Past 62 days"
    },
    ...
    {
      "key": "custom",
      "value": "Custom"
    }
  ],
  "status": [
    {
      "key": "new",
      "value": "New"
    },
    {
      "key": "pending",
      "value": "Pending"
    },
    ...
  ],
  ...
}
```

Listing 3.2: Example of the returned data from the options endpoint

Format of the returned data from the endpoints for data

The object returned by the endpoints contains four attributes. The *response_type* attribute is a string describing what is returned. The *links* attribute contains pagination links to the first page, last page, next page, and previous page. The *meta* key contains an object with additional information to the data, such as the count of the returned elements, and the counts for each groups when the data is grouped.

The fourth attribute can be the *dates* attribute when the data is grouped by time or the *items* attribute when the data is not grouped. The *dates* contains an array of objects which has two attributes – *date*, which represents the value by which the data is grouped, and the *items* array.

The items array contains objects which are the entries in the data. Each of these objects always have an *id*, *name*, and the attributes which were specified in the request body under the *attributes* parameter. An example of the returned data is shown in listing 3.3.

```json
{
  "group_by": [
    {
      "key": "cluster",
      "value": "Cluster"
    },
    ...
  ],
  "quick_date_range": [
    {
      "key": "last_62_days",
      "value": "Past 62 days"
    },
    ...
    {
      "key": "custom",
      "value": "Custom"
    }
  ],
  "status": [
    {
      "key": "new",
      "value": "New"
    },
    {
      "key": "pending",
      "value": "Pending"
    },
    ...
  ],
  ...
}
```

Listing 3.3: Example of the returned data from an endpoint for data

Chapter 4

Charts

This chapter introduces the charts and charting dashboards. Furthermore, it discusses the existing solutions, their strengths, and their shortcomings. It also details the required features of both the charting library and the chart dashboard. At the end of the chapter, the reader should have a strong understanding of what the final solution should be capable of and what differences it has when compared to other charting dashboards.

4.1 Charting Library

The charting libraries main task is to turn data into charts. These charts usually display data in two dimensions. There are different types of charts. The most widely known and used charts are line charts, pie charts, bar charts, histograms, and their variations. There are of course a wide range of other chart types, like the radar chart, but they are not important for understanding the charting libraries [24].

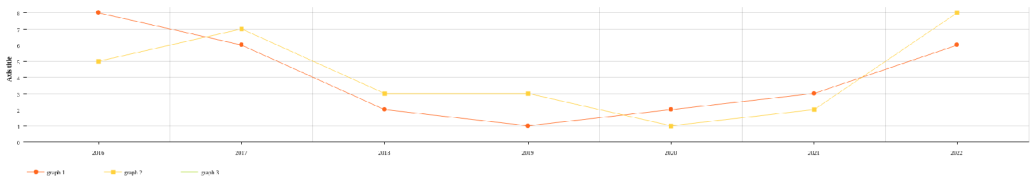


Figure 4.1: Line chart example

4.1.1 Type of charts

Line charts show continuous data over time. They consist of points which are connected, and of an x-axis and a y-axis. Usually, the x-axis represents the time variable. They are used to show trends and to help with predictions. Another usage of the line charts is to compare multiple variables over the same time period. Figure 4.1 shows an example of the line chart.

Bar charts display categorized data in the form of bars. The x-axis represents the category, while the y-axis represents the measured value. The width and height of the bars are proportional to the data they represent. They are ideal for comparing large data changes over time or comparing differences between multiple categories. Figure 4.2 shows an example of the bar chart.

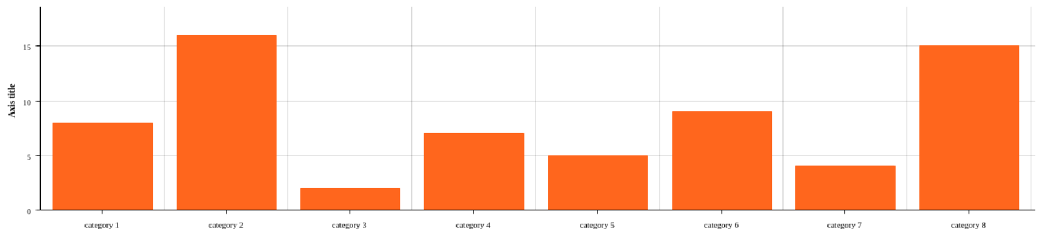


Figure 4.2: Bar chart example

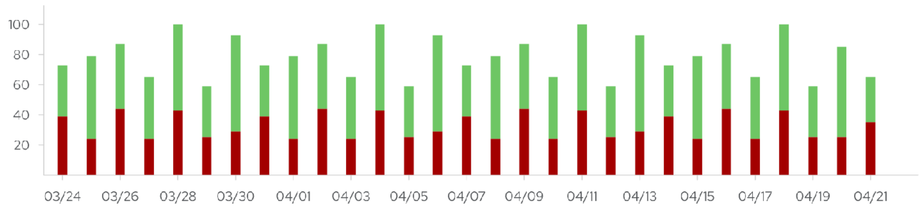


Figure 4.3: Stacked bar chart example

The stacked bar chart is a variant of the bar chart. It can split each bar into multiple smaller bars on top of one another. The stacked bar chart can display multiple measured values of a category. An example is shown in fig. 4.3.

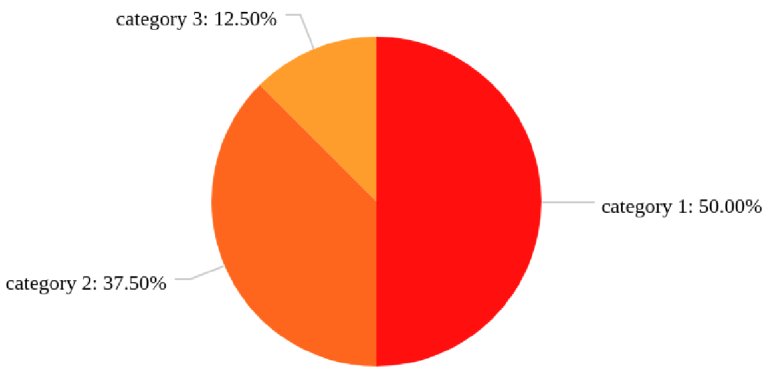


Figure 4.4: Pie chart example

Pie charts are great at displaying numerical proportions. Each slice's size is relative to the size of a particular category in a given group as a whole. Figure 4.1 shows an example of the pie chart.

The legend is one of the most significant parts of the chart. It shows which colour represents which category and gives additional information about the displayed data. This information can be an exact value of the measured data or any other calculated measurement. An interactive legend can allow the user to change the displayed data in the chart.

4.1.2 Required features

Automation Analytics uses the React framework and the TypeScript superset language. This fact is taken into consideration when searching for charting libraries to be used within Automation Analytics. The library should be easy to use within the React framework. The typing information is a bonus, as it would allow for static type checking of the used components from the library. When looking at a charting library, there are additional criteria to evaluate them by.

The first, most obvious criterion is the capability of the charting library. Automation Analytics requires that the charting library be capable of drawing at least a bar chart, a line chart, a stacked bar chart, a grouped bar chart, a multi-line chart, and a pie chart. The labels for each axis and the data should be customizable, and the scales of each axis should be modifiable.

The library should have an option to select colour options for the represented data set. It should also provide per data point colour options, which would allow colouring some parts of the chart differently from others. The next required functionality is to have an on-hover help toolbar option with an on-click handler for each data point represented.

For future development considerations, a nice feature would be having multiple charts under each other with the same values on the x-axis while representing different data. Then, in these charts, the on-hover option would fire the on-hover effect on both charts. Having some kind of „portal“ functionality is not a hard requirement, rather a welcomed addition.

Web pages in our current era are highly responsive and it is often a requirement that the page be usable on a range of displays from 4K monitors to smartphones. Therefore, it is important that each component of the page is responsive. Most of the charts are just generated images, which are, by definition, not responsive. The library which handles resizing charts internally would be favoured.

Obviously, the interface of the library will play a key role. It is important that the library's interface is easy to use and well documented. The interface should be modular while following good practices for a React application.

A well-known problem of JavaScript libraries is the dependency tree. The more dependencies a library has, the more active the maintaining team has to be. It can also introduce a problem when trying to use libraries with different versions of the same dependencies.

For cases where a library does not meet expectations, the library should be updated and extended. Because of this, the library should be open source, actively maintained, and have a responsible team behind it.

4.1.3 Existing charting Libraries

There are several charting libraries out there. This section analyzes the most popular charts for the React framework. The popularity behind them gives a basic level of assurance that the library is usable and maintained. The three most popular charting libraries are: Rechart, Victory, and Nivo

Recharts

Recharts[11] is a charting library built on top of D3 (Data Driven Documents). Recharts charts are built with the help of reusable React components. The charts are composed SVG elements. The customization is done with tweaking component props and passing in custom components. The listing 4.1 shows a basic example of the chart hierarchy.

```
<BarChart width={730} height={250} data={data}>
  <CartesianGrid strokeDasharray="3 3" />
  <XAxis dataKey="name" />
  <YAxis />
  <Tooltip />
  <Legend />
  <Bar dataKey="pv" fill="#8884d8" />
  <Bar dataKey="uv" fill="#82ca9d" />
</BarChart>
```

Listing 4.1: Recharts Bar chart example

The top level component is the responsive container which takes care of the resizing. This component is not required. Then, there are a variety of chart components: area chart, bar chart, line chart, pie chart, radar chart, scatter chart, funnel chart, and treemap. This chart component, most importantly, has the data passed to it. Also, this component can have some customizations like margins, width, and height.

All the other customization is done in the child components. The part of the chart, the axes, the legend, and toolbar are passed as a child component to the chart component. For example, to create a grouped bar chart, two bar components have to be passed to the corresponding chart component. The same applies to the other types of charts, too.

The axis component can have a nested label component for creating and customizing labels. There is an option for passing a label to the bar and line components too. This option allows labelling inside the chart.

The library also has the ability to draw different shapes in the chart. These components need the coordinates where they should be drawn. There is an option for passing a custom component to the chart; however, this option is not documented in more breath.

The library has 10 dependencies and two peer dependencies. On their GitHub page[14], there are lots of open issues and they are actively searching for contributors. This indicates that, even though the library is actively maintained, they do not have enough resources to keep up with new demands.

The documentation is readable and has multiple examples. One of its shortcomings is that there is no existing documentation on the topic of how to create custom wrapper components.

Victory

Victory[9] is a set of modular charting components for React and React Native. The library makes it easy to get started without sacrificing flexibility. Victory styles and behaviors are fully customizable. The components are highly modular and the library has no dependencies.

```
<VictoryChart
    domainPadding={20}
>
    <VictoryAxis
        tickValues={[1, 2, 3, 4]}
        tickFormat={["Quarter 1", "Quarter 2", "Quarter 3", "Quarter 4"]}
    />
    <VictoryAxis dependentAxis />
    <VictoryBar
        data={data}
        x="quarter"
        y="earnings"
    />
</VictoryChart>
```

Listing 4.2: Victory Bar chart example

The listing 4.2 shows the basic concept of the charts. First, the chart component has to be defined. It is responsible for the generated SVG image without the elements in it. It has different props like padding, margins, or dimensions. The other elements to the SVG are added within the child component.

The library supports line, bar, and pie charts, vornoi diagrams, and histograms. Additionally, it has components for stacking and grouping the charts. This allows the combination of multiple charts into one. Each of the components has its own props for customizing the look of the SVG element. The components also have options for events and the charts support animations.

The axis component is the same for both of the axes. The axis component allows the specification of the domain, tick values, formatting, and more. If not specified, the default value for the domain comes from the data passed to the charts.

The library legend component can be used to display the legend. To move the legend around, the positioning props x and y can be used. The legend functions as a standalone box, so it can be placed anywhere. The charts also can have labels. When the labels need to be customized, a label component can be passed to the chart component.

If the components from the library do not provide enough customization, there is always an option to write a custom component and use it. On one hand, writing custom components is not trivial; however, on the other hand, it has to render its own SVG elements. All in all, it is a great feature.

The GitHub[15] page confirms that it has no dependencies. The library has a broad user base, few maintainers, and lots of contributors. It has a larger issue backlog, but most of them are questions.

Their documentation provides good examples of usage. The examples are interactive and the props are explained well.

Nivo

Nivo[1] is built on top of D3 (Data Driven Documents) and Reactjs. Each Nivo chart is a separate component. These components are highly customizable with the help of a variety of props. Nivo charts do not use component hierarchy for customizing the charts. The listing 4.3 shows a basic bar chart example.

```
const keys = ['hot dogs', 'burgers', 'sandwich', 'kebab', 'fries', 'donut']
const commonProps = {
    width: 900,
    height: 500,
    margin: { top: 60, right: 80, bottom: 60, left: 80 },
    data: generateCountriesData(),
    indexBy: 'country',
    keys,
    padding: 0.2,
    labelTextColor: 'inherit:darker(1.4)',
    labelSkipWidth: 16,
    labelSkipHeight: 16,
}
<Bar {...commonProps} />
```

Listing 4.3: Nivo Bar chart example

Nivo has multiple packages. The 'core' package provides the core functionality. Then, another package has to be loaded with the exact type of the chart. Each chart has a large amount of props through which the given chart can be customized. There are common options like width, height, and argins. Each chart has a responsive version, too.

The on-hover can be turned on and off from the props. It can display one value or a range of values. The colour options of the chart support different colour schemas or exact colours. The props allow four different axis representations on the four sides of the chart. Each axis can have its own label, scale, and range.

Most of the items have an effect object, in which different event handlers can be set as on-click or on-hover. The legend has a variety of styling options, including dimensions, position, and padding. Labels can also be displayed inside the chart.

The library supports a range of animations for the charts and translations between data sets. What is different in this library is that the grouped bar chart and the stacked bar chart is the same bar chart, but that the group option has to be changed accordingly.

From looking at the Nivo repository[13], we can see that it has twelve dependencies and heavily relies on D3. The library seems to only have one active maintainer and a few contributors. While the library is actively maintained, there is a long backlog of issues waiting to be solved.

Their documentation is extensive and interactive. Sadly, the library does not allow custom components or more in depth customization.

4.2 Chart Dashboard

A chart dashboard is a page where multiple charts are displayed. These charts usually contain data which are in some sense connected. The chart dashboards are great at showing a huge amount of data to the user. It can be used to visualise the same data in different types of charts or, on the contrary, visualise different data in the same chart format.

The dashboards allow a broader understanding of the monitored data displayed in it. The user then may make assumptions or deduce useful information from the variety of charts. However, displaying information in a bunch of charts is not enough. This information should be curated, useful in the context, and has to be visualised in an appropriate type of chart. Otherwise, the information could be hard to understand or completely misleading.

In some cases, these chart dashboards are set up by a group of experts, especially if the data is presented to users who are less knowledgeable in the given field. However, in some cases, the user has enough knowledge of the given field and may want to customize or change the format and the displayed data in the dashboard. Therefore, the dashboard needs to be able to accept user input.

These inputs from the user can be a variety of displayed attributes: changes in the data, the option to change the chart's type, changing the data source, adding or removing charts, reorganizing the charts, and much more. These dynamic chart dashboards should expose these options in the most user-friendly way. Most of the use cases for these custom chart dashboards are utilized by non-programmers, so the user interface should adjust.

The rest of this section analyzes and takes inspiration from similar existing solutions. Thus establishing ideal requirements for the dashboard.

4.2.1 Existing Chart Dashboards

When searching for some kind of chart dashboard, there are two well known web applications which have chart dashboards – Chartlog and Grafana.

Chartlog

Chartlog[5] is an analytics and journaling platform for active day traders. The software is browser based. An account is needed to try out Chartlog; therefore, all the information provided further is gained from external review[22].

The software does not currently have fully customizable chart dashboards. It provides different types of charts with predefined data-sets. The user has some options to set the colours of the chart, and change views, which display different charts. Chartlog relies heavily on line charts and multi-line charts as the data they display is mostly time dependent and continuous. The software is also using a bar and a box chart. The charts show a small hint window when hovering on the data point in the chart. These hints show the exact value at that point.

Chartlog is a piece of software which relies heavily on graphs but it does not provide any means to the user to truly customize and build a chart dashboard.

Grafana

Grafana[17] is a multi-platform open source analytics and interactive visualization web application. It allows the user to connect external data sources from various web services. Grafana has a huge variety of charts and styles for each chart. The charts are highly customizable, as is the data they are representing.

The user can create multiple chart dashboards and save them. The charts can be reordered on the dashboard as well as resized. It can be done easily by dragging the window or the corner of the window around. The fig. 4.5 shows an example of the dashboard with five different charts on it.

Hovering on the charts creates a tooltip window next to the cursor, displaying the exact values of the data point. The line charts have a coloured area under them and the hover works even when the cursor is in this area. This makes it less painful to hover over the line chart.

The tooltips are also interactive. They display the colour and the name of the measured value, but clicking on them hides all of the other values from the chart. The top of the

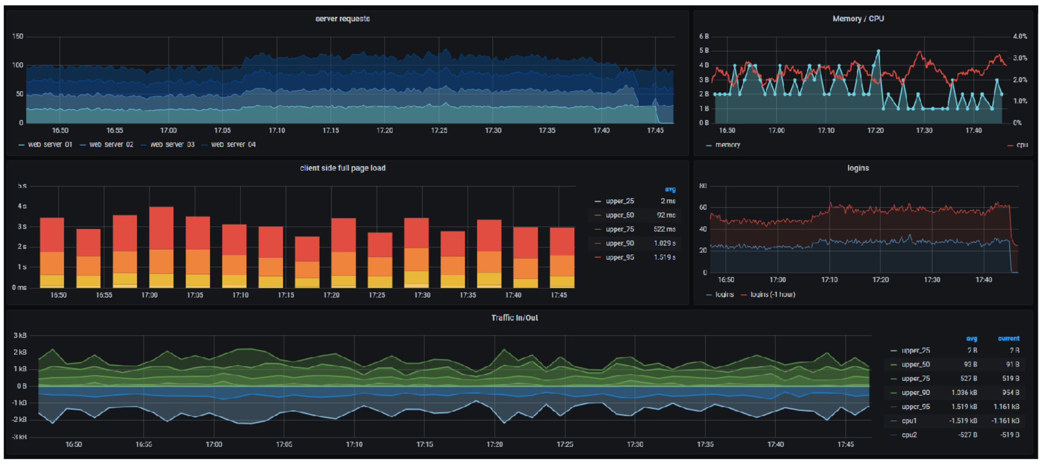
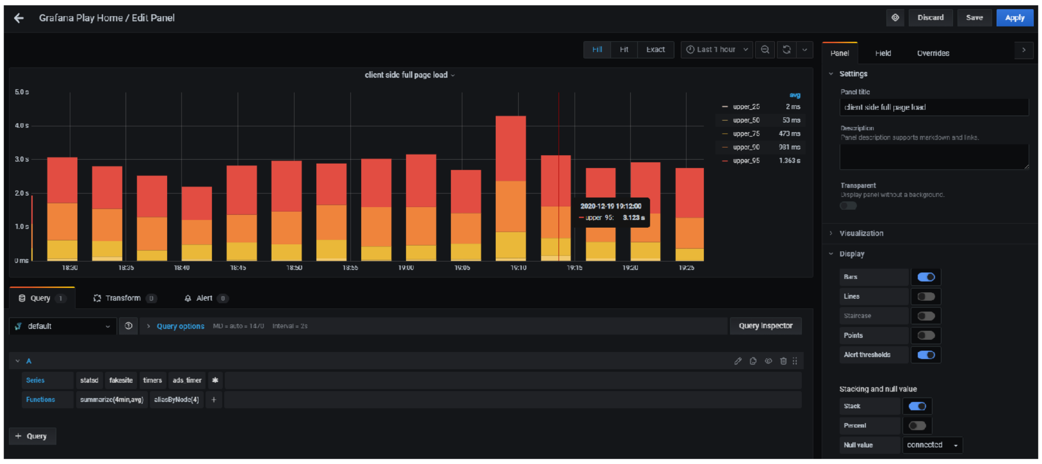


Figure 4.5: Grafana Dashboard Example

charts have a dropdown menu which allows additional views of the chart. The first one is a simple view, which opens the chart on full screen.

The second option is the edit. The edit option opens up a new window, which the chart is the part of, but which includes many panels and options from which to choose. Figure 4.6 shows these options. Right under the chart there are three main options: query the data endpoint, transform the dat,a and to set up alerts. The query allows users to select the endpoint from which the chart will retrieve the data. Additionally, there are various parameters which can edit the query, like maximum value, time frame, interval, and so forth.



2020-12-19 19:12:00
upper_95: 3.123 s

default
Query options MD = auto = 1470 Interval = 2s

A
Series: statsd fakesite timers ads_timer *
Functions: summarize(4min,avg) aliasByNode(4) +

Settings
Panel title: client side full page load
Description: Panel description supports markdown and links.
Transparent: Display panel without a background.
Visualization
Display
Bars: on
Lines: off
Staircase: off
Points: off
Alert thresholds: on
Stacking and null value
Stack: on
Percent: off
Null value: connected

Figure 4.6: Grafana Edit Chart Example

When setting up an endpoint, the query panel shows the options for the series. Here, the user can select the attributes to display in the chart, along with some predefined functions which can modify the data structure, so the chart can display it correctly.

The visual customization of the chart can be found in the right panel. The first chart customization item is the settings, where the title and the description of the chart can be set. The next menu item is the visualization. Listed under this item are all of the different types of charts. The display menu item has all the customization options for the body of the chart. The graph has the option to display bars and lines in the same chart. When lines are enabled, there are additional options to fill the area under the lines, to set the transparency, and the width of the lines. The tooltip options are limited to show one series in the tooltip or all of them.

Under the axes menu item are options to customize each of the axes. Users are able to set a unit, scale, minimum value, maximum value, and label for each of the axis separately. The x-axis can be set to display different data type: time, series, or a histogram.

The legend can be customized under the legend menu item. It offers a visual customization of the legend, along with the ability to choose the placement of the legend. The legend can be displayed under the chart or next to it. There are options to show additional data in the legend as the minimum, maximum, and average of the displayed series.

4.2.2 Required features

The chart dashboard should be customizable by the user. Therefore, part of its interface should allow the user to configure it. There are two configurable parts – the data source and the visualisation options.

The data source should be composed of two parts – the source and the parameters. For each source, there are different sets of allowed parameters and allowed values of these parameters. The user should not be able to select an invalid configuration. To achieve this, the user interface should not allow the user to create this configuration.

The visualisation options directly affect the shape of the displayed data. The first obvious option is to select which type of chart should be used for data visualisation. Then the user should be able to selects which attributes of the received data should be visualised and in which part of the chart.

Depending on the selected data type, the user can get a single series or a multi-series. The single series data can be viewed by time or by some other attribute. When the single series data is grouped by time, then it is expected that the data should be visualised by time, meaning that time will be showed on the x-axis. Then the data can be visualised by a simple line, bar, or area chart. However, if multiple attributes are needed to be visualised from the data set, the stacked chart can be used. When the single series is grouped by something else than time, a more appropriate chart to display the data in is the pie chart. Multi-series data is expected to be grouped by time and some other attribute. To visualise multi-series data, a grouped bar chart should be used.

Then the user should be able to select and assign different colours for different attributes. Another important step in the customization is setting the labels and the title of the chart. The title and the labels are crucial to understanding the data displayed in the chart.

The chart dashboard should allow the user to interactively compose his charts. This would mean that the edited chart is updated every time the user changes a value in the configuration. The user also should be able to add multiple charts and be able to remove or edit added charts.

Chapter 5

Design and Implementation

This chapter describes the three main components of this library as well as the development setup for the project. For each component, its features and restriction are described along with their interface. This chapter provides a good understanding of the usage of the components as well as the data structure they need, and the challenges which were solved during the development.

5.1 Utilized libraries

In the section 4.1.3 the existing charting libraries were analyzed in depth. Nivo unfortunatelly has only one mantainer and a long backlog of issue which can be limiting when changes are required from the charting library. This leaves open the two other options: Victory Charts and Recharts. They are similar to use and they have very similar functionalities.

The choice of using Victory Charts over Recharts is affected by two factors: Rechart is using D3 (Data Driven Documents) for chart rendering, which is a heavy library, not optimized for React, leaving the charts hard to test. The other reason is the existence of the Patternfly Charts, which is a thin layer over the Victory Charts. Patternfly Charts is reexporting the Victory Charts while giving a default styling to the charts. This design mostly aligns with the AAA design guide. This makes the resulting chart rendering library more lightweight as it does not have to deal with designing each of the components.

5.2 Data Driven Chart Rendering

One of the most complicated and most important parts of the project is the ability to render a chart from a schema. This component has its own input into the schema of the charts, along with an object containing functions, and generates the described charts.

The section discusses the features and the limitations of the library. It also dives into the interface of the library, describing the schema and the function object in more depth. The data fetching subsection discusses the issues with the current data format returned by the API and the parsing of this data. Toward the end, the structure of the components is described, along with the explanation behind each important trade-off and decision.

5.2.1 Features, limitations

The chart renderer is capable of rendering the following charts: line chart, bar chart, area chart, scatter chart, and pie chart. Additionally, with the exception of the pie chart, the chart renderer is capable of rendering grouped charts and stacked charts. Both the grouped chart and the stacked chart support grouping and respectively stacking different types of the chart. The stacked bar chart and stacked line chart examples are shown in fig. 5.1.

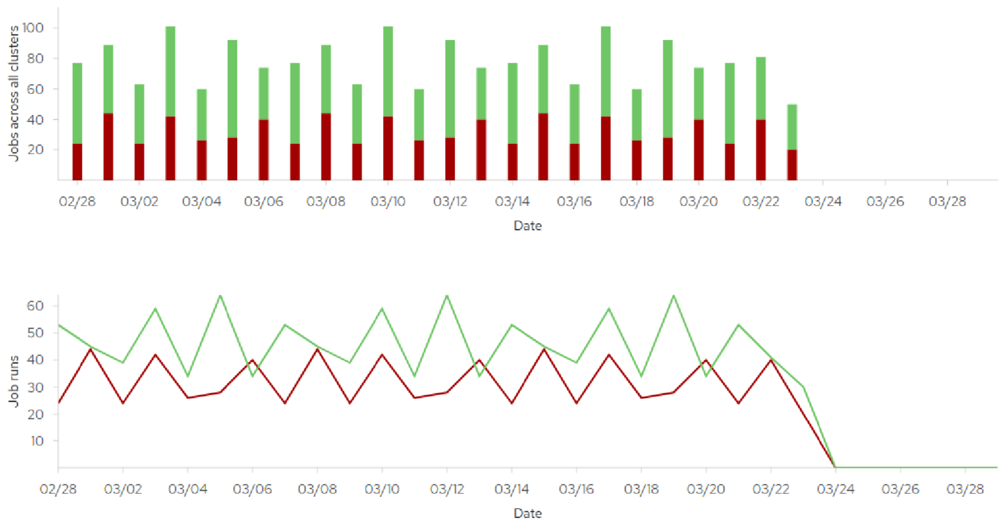


Figure 5.1: Stacked bar and stacked line chart

Stacked and grouped charts

Stacking charts where the two charts are using the same single series requires the schema to explicitly define the charts. This means that there is no option to render a variable amount of charts from a single series. The same limitation is applicable for the grouped charts. Dynamic chart generation is possible when the received data is a multi-series. In this case, a chart can be rendered for each series from a template. Then, each chart receives one series from the multi-series and the count of the series matches the count of the rendered charts in the group. This feature is useful when the user wants to set their custom limit for the API call. An example of a chart generated by the template is shown in fig. 5.2.

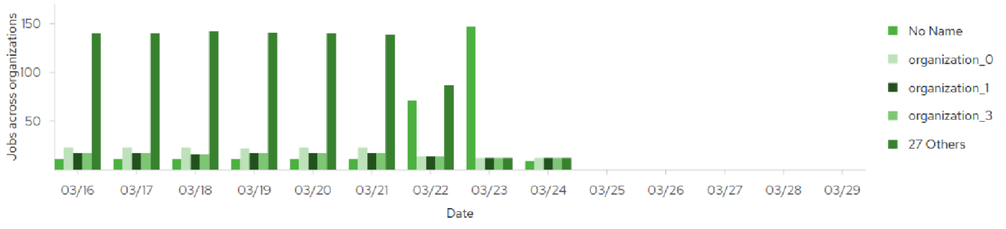


Figure 5.2: Dynamic grouped bar chart

The following feature is also connected to chart rendering. When rendering a grouped bar chart, each bar chart has to have its own width and the group itself has to have the offset for spacing the charts. The chart renderer is capable of calculating both the width and the offset for the bar charts. Since the dynamic rendering is not applicable for the stacked bar charts, the offset for the bar stacks has to be specified in the schema. If only one bar chart is being rendered, the chart renderer is capable of setting the offset again.

Legend

The most integral part of the charts is the legend. The chart renderer can create legends for the charts. From the limitations of the used charting library, the legends can be located below the chart or on the right side of the chart. In either position, it is possible to orient the legend horizontally or vertically. The legend then has a colour and a name, which aligns with the chart colors.

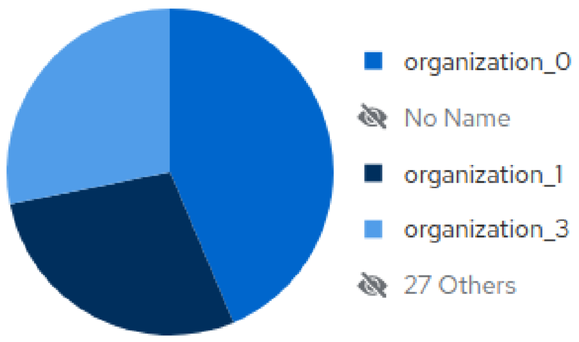


Figure 5.3: Pie chart with interactive legend

A really useful feature for the chart renderer is the capability to render an interactive legend. By setting one boolean value to true, the legend becomes interactive and, when clicked, the chart renderer hides or shows the associated series to the legend entry. This is illustrated in the fig. 5.3.

The implementation here is limiting. While having a mutli-series, the chart renderer toggles a whole series to hide or show a chart. In the pie chart, the legend toggles entries in the series to hide or show that value in the chart. However, when using a single series in multiple charts, like a stacked chart with a single series, there is no deterministic way to get which attributes should be hidden or shown; therefore, the interactive legend will not work.

By default, the PatternFly 4 Charts comes with helper functions to make creating the interactive legend easier. However, their implementation does not work in some cases, like with the pie chart. After some discussions, the PatternFly team will consider changing the event handling for their helper function. In the meantime, the interactive legend is custom implemented in the library using the basic Victory Chart events functionality.

Hover and click event

When hovering the mouse over a chart entry, the user may want to see additional information or more exact values tied to that point. The chart renderer supports these tooltips in two different forms.

The first format is to display the tooltip for a single value, when hovering over a specific point in the chart. This mode is useful when having lots of charts grouped together. In the schema, it can be generated automatically by specifying the attribute name it should display in the tooltip, else a custom function can be passed for advanced styling and processing. This function receives all the data associated with the concrete point in the chart. A renderer example of this tooltip is fig. 5.4.

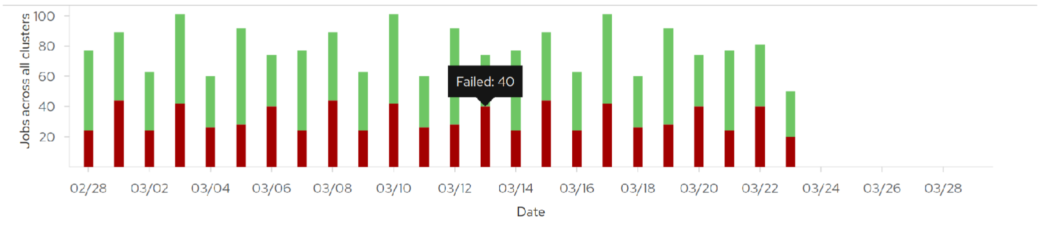


Figure 5.4: Stacked bar chart with a single value tooltip

The other option is to enable the tooltip for the chart wrapper. The tooltips are still defined for each chart, but, when hovering over the chart, the displayed tooltip contains all the chart tootlips over the x-axis. This tooltip is shown in the fig. 5.5.

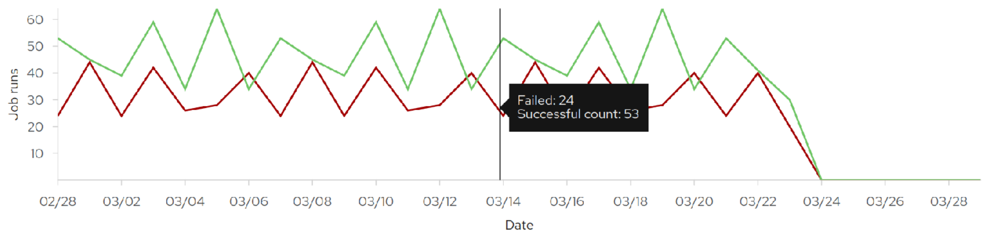


Figure 5.5: Stacked line chart with a tooltip

The most important event the chart can have is an on-click event. This is useful for drilling down into the data entry. Multiple on-click functions can be defined for the chart builder. Each chart, defined in the schema, and can have its own function for handling the on-click event. These functions receive all the data tied to the entry of the clicked point in the chart, as well as all the data tied to the on-click event in the browser. This allows the developers to have full control over what should happen when a click event is fired.

5.2.2 The interface

The interface consist of two objects. The first object is the schema of the chart and the second is an object containing functions. The need of the functions object is tied to the fact that the schema will be stored in a server-side database. This means that, in the schema, functions cannot be stored. However, there is a need for passing functions to the chart, which will be discussed later. One use case for these functions is on-click events or axis formatting. The chart rendering components take a functions object as it input and these functions are referenced in the schema by their name and are stored as a string.

This approach also increases the security of the code. Since the schema will be composed by the end user, the ability to inject a function into the code would extend the customization capabilities of the component. At the same time, it would also allow the user to inject and execute malicious code.

To avoid the security risk just described, the functions object is not exposed to the user. It is up to the developer to write their custom function and then give the user the ability to choose from these functions. This ensures that the only one who can write executable code is the developer, while keeping the customization on a high level. This also means that the developer does not depend on the library when new functions are required on their side.

Schema

Using TypeScript[18] has clear advantages when creating an interface. When creating the schema for the chart, the interface can be defined using the *type*, *interface*, and the *enum* features from TypeScript. This allows other developers to know exactly what type(s) and what structure is needed as an input for the chart renderer. This section only describes the types in the schema which are needed for the chart renderer to function. The structure of the required data from the API is described in section 5.2.3.

The schema is an array of objects. The top level objects describe chart elements. By default, there are two types of elements:

- Top level chart elements, like *chart wrapper* and *pie chart*. These elements have to fetch the data, handle resizing, and they have no parent.
- Child chart elements, like *grouped chart* or *simple charts*. These elements always have a parent.

There are three common fields for all top level objects in the schema – *id, kind, parent*. The *id* is an unique id of that element. This id is used for reference in the *parent* field to indicate the parent-children relationship. The *kind* property indicates what type of chart element the object is describing. The *kind* property can have currently four values:

- *wrapper* - indicates that the object is a top level chart element without a parent and also indicates the presence of a new chart.
- *group* and *stack* - these indicating „middle level“ elements. These types usually have more than one child and they are grouping or stacking them. Their parent is always a wrapper.
- *simple* - this indicates that the object is describing a simple chart element. This element can render different types of simple charts – one series visualised in a line, bar, area, or scatter chart. The *simple* object's parent can be both the *wrapper* or the *group/stack* elements.

All elements also have a *props* field, which is different for each element. This field is directly passed to the component which is rendered from this object as its props. This allows the developer to have full control over the component. The prop field is overwritten only in cases where a prop could break the chart renderer. A basic stacked bar chart example is shown in listing 5.1. In the example, the *ChartKind* and the *ChartType* are imported constants from the library.

```
[
    {
        id: 1000,
        kind: ChartKind.wrapper,
        type: ChartTopLevelType.chart,
        parent: null,
        props: {
            height: 300
        },
        xAxis: {
            label: 'Date'
        },
        yAxis: {
            label: 'Jobs across all clusters'
        },
        api: {...}
    },
    {
        id: 1100,
        kind: ChartKind.stack,
        parent: 1000,
        props: {}
    },
    {
        id: 1002,
        kind: ChartKind.simple,
        type: ChartType.bar,
        parent: 1100,
        props: {}
    },
    {
        id: 1001,
        kind: ChartKind.simple,
        type: ChartType.bar,
        parent: 1100
        props: {}
    },
]
```

Listing 5.1: Simple stacked bar chart schema

In the example listing 5.1, the *api* field is hidden. This object describes the API endpoint, which returns the data the chart renderer uses to populate the charts. The interface is straightforward, it has a *url* field, which points to the endpoint, and a *params* field, which is a key-value object passed directly to the API in the *POST* request body. The data which is received from the API is converted into a more robust structure, which is described in the section 5.2.3.

The axes of the chart are described on the wrapper object as is shown in the listing 5.1. The x and y axis are defined separately. Each of the axes can have a *label*, which shows up as text next to the axis. The axis fields are aligned with the Victory Chart's axis props, with one exception. The *tickFormat* field controls how the axis values should be formatted.

In the Victory Chart library, this prop is a function; in the schema, it is a function named as a string and should exist in the functions object passed to the chart renderer.

The tooltip can be defined on two levels. If the tooltip should be individualized for each chart, then it has to be defined in the *simple chart* element. The *tooltip* field accepts the type of the tooltip. Right now, only the default tooltip is supported, which creates a tooltip with 'Label: value' format. This component can be customized by passing options in the *props* field. These props are passed directly to the tooltip component. To further customize the tooltip, *labelName* can be set. This changes the label part of the tooltip. The *customFnc* can be used to generate a custom tooltip content from the data. When it is passed, it takes precedence over the default 'Label: value' format. This type of tooltip for a single chart is shown in listing 5.2.

```
...
{
    id: 2,
    kind: ChartKind.simple,
    type: ChartType.line,
    parent: 1,
    ...
    tooltip: {
        type: ChartTooltipType.default,
        props: {...},
        labelName: 'custom name',
        customFnc: null
    }
},
...
```

Listing 5.2: Schema for a single tooltip on a line chart

The other type of tooltip approach is to combine all the tooltips over a point on the x-axis and display all the tooltips at once. This tooltip can be achieved by defining the tooltip on the *wrapper* component. The chart components can still be customized in the same way as before, but they will be rendered under each other in one tooltip. The *tooltip* field on the wrapper can have a *customFnc*, which behaves the same as on the *single chart* tooltip. There is one customization option right now and that is the *cursor* field. If set to true, the chart renders a horizontal line in the chart, following the mouse, and displays the tooltip next to it. A basic example of the chart tooltip with the enabled cursor option is demonstrated on listing 5.3.

```
[
    {
        id: 1,
        kind: ChartKind.wrapper,
        type: ChartTopLevelType.chart,
        parent: null,
        tooltip: {
            cursor: true
        },
        ...
    },
    {
        id: 2,
        kind: ChartKind.group,
        parent: 1,
        props: {}
    },
    {
        id: 3,
        kind: ChartKind.simple,
        type: ChartType.line,
        parent: 2,
        tooltip: {
            labelName: 'Failed'
        },
        ...
    },
    {
        id: 4,
        kind: ChartKind.simple,
        type: ChartType.line,
        parent: 2,
        tooltip: {
            labelName: 'Success'
        },
        ...
    },
    ...
]
```

Listing 5.3: Schema for the wrapper tooltip

The legend has to be defined on the *wrapper* component. The two required fields for the legend are position and orientation. The position can be *bottom* or *right*, while the orientation can be *vertical* or *horizontal*. An optional field is *interactive*, which is set to false by default. When set to true, this makes the chart legend interactive and clicking on the legend entry toggles the series in the chart. A simple example is shown in listing 5.4.

```
...
{
    id: 1,
    kind: ChartKind.wrapper,
    type: ChartTopLevelType.chart,
    parent: null,
    ...
    legend: {
        interactive: true,
        orientation: ChartLegendOrientation.vertical,
        position: ChartLegendPosition.right
    }
},
...
```
Listing 5.4: Simple legend example

The on-click function is passed to the *simple chart* as a string, representing a function name. After that, a function from the passed functions object should have a function with the same name.

Functions

The functions object is much more straightforward than the schema object. The idea is to have a key for each type of functionality and that these keys hold another object, where the key is the name of the function and the value associated with the key is a JavaScript function. This format is readable for developers, easy to work with, and really easy to extend.

Currently, three types of functions are used in the chart renderer. The *onClick* key is for the functions which can handle on-click events. The *axisFormat* key contains functions for formatting the axes, like transforming dates, or converting high and low values to more readable formats. The *fetchFnc* key holds a single function. This function is used to fetch the data from the API. This function allows greater control over the data fetching. Especially useful when the application has to authenticate before the fetch can be made.

```
const functions = {
    onClick: {
        'doNothing': () => {},
        'myFunction': (data) => { console.log(data); }
    },
    axisFormat: {
        formatDateAsDays: (i) => (i && i.split('-')[2]);
    },
    fetchFnc: (url) => fetch(url);
}
```
Listing 5.5: Example of the functions object for the chart renderer

In the example listing 5.5 four functions are defined, two for the on-click events, one for formatting the axis, and one for the fetch function. The supplied functions are using the arrow function format, but any kind of function can be supplied. If the 'myFunction' is to be executed when a chart element is clicked, then it must be saved in the schema under the string 'myFunction' in the appropriate field. Then, the chart renderer will search for this function and call it with parameters for the on-click event. Since JavaScript has no type checks, the function can accept fewer parameters than is passed to it.

The chart renderer component ships with a few predefined functions. They can be imported to the code and then modified, or just passed to the chart renderer.

5.2.3 Data fetching from the API

Developing the API is not part of this scope of work and is handled by other people. This means that the chart builder needs to use the API that is made available to make the component usable for Automation Analytics and accommodate any changes made to the API. The current API which supplies the data to the chart renderer is still missing some features, which will be further discussed at the end of this section.

The basic schema of the API is described in Section 3.2.1. To be more readable, the returned objects from the API will be organized by charts.

Single series

The most basic structure is a single series, which can be visualised via line, bar, area, and scatter charts. The object contains an *items* array. The *items* array contain objects, such as where the *created_date* is usually on the x-axis, and other attributes that can be visualised on the y-axis of the chart. This also allows for the composition of a stacked or a grouped chart, where we display two attributes in two different charts, in the series with just one call to the API. An example of this type of object returned by the API can be found in listing 5.6.

```
{
    items: [
        {
            successful_count: 34,
            failed_count: 34,
            created_date: "2021-02-23"
        }, {
            successful_count: 60,
            failed_count: 24,
            created_date: "2021-02-24"
        },
        ...
    ]
}
```

Listing 5.6: The single series object returner by the AAA API

The pie chart uses also a single series, but no is date supplied, just the attributes. This means it should be handled differently when rendering; however, from the fetching and parsing perspective, it is still the same as the previous single series.

Multiple series

The other type of series that the API can return is the multiple series. This happens when the data is grouped by an attribute. This kind of data can be visualised perfectly in a grouped bar chart. There is an example in the listing 5.7. It is basically what it says, a single series nested inside another array. The extra information outside of the series is the attribute by which the data is grouped. In the example, the series are grouped by *date* (time).

```
{
    dates: [
        {
            date: "2021-03-11",
            items: [
                {
                    total_count: 12,
                    total_org_count: 1,
                    id: 1,
                    name: "organization_2"
                },
                ...
            ]
        },
        ...
    ]
}
```

Listing 5.7: The multi series object returner by the AAA API

Parsing

The charts require a different format of data than the API returns, therefore all the data received from the API has to be parsed. Additionally, there is some extra information that needs to be stored. This extra information is a *hidden* field, which is required for the interactive legend, and a *name* field, which contains a unique id for the chart that displays the data. This data structure is described in listing 5.8.

```
{
    data: [
        {
            serie: [
                {
                    attr1: value1,
                    attr2: value2,
                    ...
                },
                ...
            ],
            hidden: false,
            name: 'unique-string-identifier'
        },
        ...
    ]
}
```

Listing 5.8: The internal data structure for the charts

As it is shown in listing 5.8, the data is always an array of the series. This means that even if the API returns a single series, it is stored as an element of the *data* array. This step unifies the data structure for any data returned from the API and enables the data to be more uniform processed.

Legend

The legend is an integral part for most of the charts. However, the API does not returning the legend entries. This means, after parsing the data, the legend entries have to be derived from the data itself. This method of extracting the legend entries from the data is not bulletproof, but works with the current API.

The process is the following:

- If there is only one series:
  - Iterate over all the objects in the series and return the *name* attribute in an array.
- If there are multiple series:
  - Iterate over each series.
  - From each, series get the first entry and return its name in an array.

Future API work

The development behind the API is still ongoing and there will be discussions on how to deal with the format for the charts; however, here are some of the proposed features.

- Send the tooltip with the data as an attribute.
- Send the legend with the data in the *meta* field.
- Create a placeholder when the *name* attribute is empty in the data.
- Add a *chart* flag to the request to make the API return the data in a format which can be supplied to the charts directly.

5.2.4 Chart creation

The chart renderer uses the Victory Charts library to create charts. The Victory Charts library is discussed more depth in section 4.1. The chart renderer should follow the UX design for the Automation Analytics. This could have been done by styling the charts to fit into the UX explicitly; however, there is already an existing styling for the Victory Charts which fits the UX design the Ansible Automation Platform is following, called PatternFly 4 Charts [21].

PatternFly 4 Charts is a thin layer over the Victory Charts. The library reexports the same components which already exist in the Victory Charts, with default styling and, in some cases, with extra configuration, while keeping the Victory Chart components still fully configurable. Using the PatternFly 4 Charts instead of the Victory Charts eliminates the need for most of the custom styling of the components, but also makes the development more challenging, since now there are two libraries to work with.

There are two types of components provided in the PatternFly 4 Charts. The first type is the single component chart. An example of this chart type is the pie chart. The basic usage of the chart is shown in listing 5.9. It has a data prop, for passing the series, as well as props for sizing, labels, tooltips, and more.

```
<ChartPie
   data={[
      { x: 'Cats', y: 35 },
      { x: 'Dogs', y: 55 }
   ]}
   height={230}
   labels={({ datum }) => `${datum.x}: ${datum.y}`}
```

```
    legendData={[
        { name: 'Cats: 35' },
        { name: 'Dogs: 55' }
    ]}
    legendOrientation="vertical"
    legendPosition="right"
    padding={{ bottom: 20, left: 20, right: 140, top: 20 }}
    width={350}
/>
```
Listing 5.9: Pie chart component example

The other type of charts are the charts with x and y axes, like line, bar, area, or scatter charts and their stacked or grouped versions. From the basic example shown in listing 5.10, it can be seen that this format is more complex. The sizing and labels live in the top element chart, while it has axis components as child components as well as the charts it is displaying. In the example, the chart contains a group component which renders groups of charts. The groups in this case are bar charts. The bar charts have the data passed to them and the series they render.

```
<Chart
    containerComponent={<ChartVoronoiContainer
        labels={({ datum }) => `${datum.name}: ${datum.y}`}
    />}
    legendData={[
        { name: 'Cats' },
        { name: 'Dogs' }
    ]}
    legendOrientation="vertical"
    legendPosition="right"
    height={250}
    padding={{ bottom: 50, left: 50, right: 200, top: 50 }}
    width={600}
>
    <ChartAxis />
    <ChartAxis dependentAxis showGrid />
    <ChartGroup offset={11}>
        <ChartBar data={data1} />
        <ChartBar data={data2} />
    </ChartGroup>
</Chart>
```
Listing 5.10: Grouped bar chart component composition

Victory Charts does some background calculations from the data, even in the higher level components. This means that the top level *Chart* component has access to the data passed to the nested *ChartBar* child component. In React, this is only possible by passing down callback functions to the child component. This poses a problem when trying to wrap their components into a custom component. Their guidance on how to wrap the components is not working.

To overcome the challenge of not being able to wrap the component, but have separated logic for handling the render for each component, the chart renderer has basic functions which return already rendered React elements. In this way, there is no extra component layer on top of the Victory Charts, but the functions are able to pass custom props to the PatternFly components and do calculations on their own.

This approach has its own limitations. If there are only functions and no components, then the React hook cannot be used, making it impossible to watch for data changes or run asynchronous code. This also causes the whole chart to re-render, not just a sub-component of the chart when any of the input changes.

5.2.5 Top level elements

As described, there are multiple top level chart elements, like *Chart* and *ChartPie*. Their interfaces differ slightly, but there are common props and settings. These top level components also have to have extra logic for handling the resize event and handling the interactive legend event.

Responsiveness

To have a responsive charting library usable both on wide screens and on smartphones, it is necessary to handle the resize event and to adjust the chart to different screen widths. The chart itself is a picture composed of SVG elements. This means that Victory Chart's responsiveness is the same as a picture. This has multiple problems, but the most notable is the change of text size in the chart. When scaling the chart down, at some point the text will get so small that it becomes unreadable. When scaling up, the text becomes too big and unwieldy.

The solution to this scaling is to set the text size dynamically. Fortunately, the PatternFly 4 Charts can handle the resize. However, it comes at the cost of out of the box resizing. This means that every chart has to have a *width* and *height* defined as a number in pixels. This also means that if the chart is meant to be responsive, these numbers have to be variables, which then have to be recalculated when the resize event is fired.

In React, the solution is to create a wrapper *div* element around the charts and get its reference. Then, when the resize event is fired, the width of the wrapper div element is read and the width of the chart is set to this new value.

Fetching the data from the API

Each top element component must handle fetching the data from the API and passing down to the child components to render as a chart. The process is the same in every chart: set the loading variable to true, then start fetching the data. After the request is resolved, set loading to false and set the data to display the error.

All of the required information for fetching the data is passed down in the *api* field in each *top level schema element*. The *api* object consists of three fields: *params*, *url*, and *method*. This object is then passed to *fetchFnc*, which is the part of the *functions* passed to the component. By default, the *fetchFnc* function uses the 'POST' method, if it is not defined differently in the *method* field, and passes the *params* to the body of the request. If *method* is set to 'GET', then a GET request is fired without sending the parameters. This is great for local testing with mockup data. For more advanced fetching, the user should create their *fetchFnc* which works with the *api* object and returns the data.

Since the default responsiveness and the data fetching logic are the same for all the top level components, they can be offloaded into another component. This component is called *Responsive Container*. This component creates the required *div* wrappers for the chart. It hooks into the resize event and, on change, returns the new width to the parent component. The component also handles the fetching from the API. This means that it renders the child components passed to it (the *pie chart* and the *chart wrapper*) only when the loading is finished. When fetching data from the API, errors should also be handled. The component catches any errors returned from the call to the API and displays a simple error message instead of rendering the chart. This container component makes it easier to add new top level chart types, as well as modifying the existing logic.

5.2.6 Anomaly Detection

In data analysis, anomaly detection is the identification of rare items, events, or observations which raise suspicions by differing significantly from the majority of the data. One of the easiest methods to detect anomalies in huge data sets is with the help of charts. There are two options for spotting an anomaly in the charts. One is to display the correct data, and spot in the chart the deviant values and strange behaviours. The other one is to have some automated function which highlights the possible anomalies in the data.

The first use case does not require any special functionality on top of what the chart renderer is already capable of. The second use case requires the chart renderer to be able to highlight some values in a chart. This can be done by overlapping a line chart and a scatter chart, where the scatter chart has *null* data at the entries which are not anomalies and the anomaly value at points where the anomaly is detected.

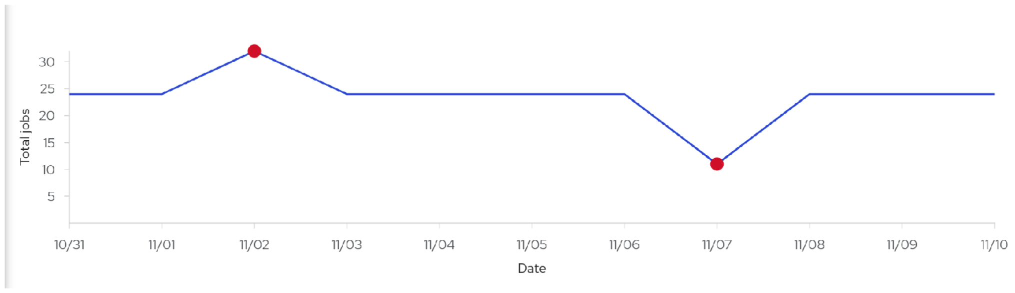


Figure 5.6: Anomaly detection example

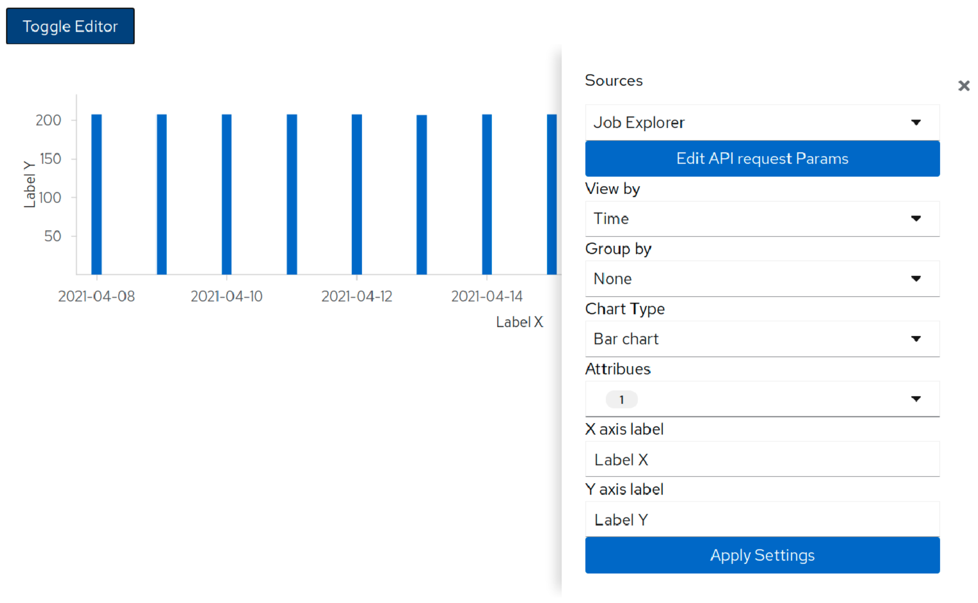
In the chart renderer, the anomaly can be appended to a single series. The Automation Analytics API does not currently support the anomaly detection, so the in the example, fig. 5.6, the chart is supplied with custom mockup data. This data contains the values for the line chart, and each data entry has an *anomaly* field, which is a *null* value if that entry is considered an anomaly. The styling in the example is minimal and custom, since there is no UX design available yet for it.

5.3 Chart Editor

The other main part of the project is the editor, which allows the user to create or edit charts by clicking through a form. The usage of this form is similar to the chart renderer. If the chart needs editing, then the existing chart schema should be passed. Then, the component needs the endpoints from which to get the option values and, finally, it has a callback function, which is called each time the schema is changed. First, this section discusses the features and limitations of the form. Then, the API format will be described, which is needed for the form to display the options correctly. In the end, the implementation details are described.

5.3.1 Features and limitations

Right now the editor is in its first iteration. A proper UX design is under way while the current version version is based on wireframes. The editor features a window with the currently edited chart and a draw in window, which can be toggled. This window, when visible, contains the form elements for editing the chart. The form in its current state has only few options to chose from, but they are the most important options. An example of adding a new chart can be seen in fig. 5.7.



Sources: Job Explorer
View by: Time
Group by: None
Chart Type: Bar chart
Attribues: 1
X axis label: Label X
Y axis label: Label Y

Figure 5.7: Chart Editor with open editor

The *sources* field allows the user to choose from different endpoints, from which the data for the chart will be fetched. In a future release, the button under it will open another field which will allow the user to send extra parameters to the endpoint. In the current version of the Automation Analytics API, the four endpoints have slightly different settings; therefore, this extra option may become available in the next version of the API. These two settings do not change the form of the chart, only the data in the chart.

The *View by* field allows the user to chose the attribute under which the data is grouped. This is a crucial option since it can change the chart type. If the data is grouped by *time*, the editor allows for the selection of another grouping option (the *Group by* field) and the chart type, a simple, stacked or grouped chart. On the x-axis of this charts, the *time* is displayed. However, if the *View by* option is set to something other than *time*, then the other grouping option is disabled, and the chart type can be only a *pie chart*.

The *Group by* option is only enabled if the *View by* option is set to *time*. Then the user can group the data by some other attribute (other than time). This causes the chart type to change to *grouped bar chart*, since this is the most reasonable option to display this kind of data set in a chart.

Currently, the *chart type* field is primarily controlled by the other options. The user can only select between different charts when the data is grouped by time. Then, the user can select between *line*, *area*, and *bar* charts.

When the chart type is *pie chart* or *grouped bar chart*, the attribute options allows one attribute to be selected. This attribute is represented from the data series. However, in the stacked charts, multiple attributes can be selected. Each attribute is displayed in its own chart, where the type is determined by the *chart type* field.

The last two options are text fields for customizing the labels. These options are not displayed when the chart type is a pie chart, since it has no labels. In the future, additional customization options may be added, which would allow users to completely change the look and feel of the charts.

To avoid invalid configurations and unnecessary re-renders in the end of the form, there is an *Apply* button which makes the displayed chart re-render with the new settings.

From the features, it is evident that the editor highly limits the possibilities of the chart creation. The chart renderer library can render a much higher variety of charts. The limitation, however, is caused by the fact that the end user may not have the knowledge to create a meaningful chart from all the options; therefore, the editor creates a more simplified view, where the user is not lost and all of the possible combinations of settings yield a meaningful chart.

5.3.2 Interface

The interface of the component consists of four props. The *schema* prop provides the default schema of the chart when editing a chart. However, the schema can contain multiple charts; therefore, the *id* prop specifies which top level element from the schema should be used as the starting point for editing the chart. If the *id* prop is not defined, then the editor creates a new chart.

The *apis* prop accepts an array of objects, where each object has the following keys defined: *url, options url, label.* The *apis* prop is used to render the options for the *sources* in the form. The *url* attribute defines the endpoint for the chart data, while the *options url* defines the endpoint from where the chard editor gets its options for the form. The *label* is the readable name displayed in the select box for the given *source.*

The last prop is the *onSchemaChange* callback function. This function is called each time the chart is changed with the new value. The new value is created by merging the new or edited chart with the rest of the passed charts in the *schema* prop.

5.3.3 Data format from the API

Some of the form options are hard coded, as is the case with *chart type.* The *source* comes from the props. All the other form options are populated from the Automation Analytics API. The accepted format is an object having a *key* attribute, which is the value of the option, and the *value* attribute, which is the displayed string for the given option. Then ,to populate the select field, an array of these option is required.

The Automation Analytics API returns these options in an another object, where the the keys in the top level objects are the names for the given attribute. When the same parameters are passed to the options endpoint and to the endpoint for the data, the options endpoint only returns valid options for the data endpoint. This makes the verification process for correct inputs obsolete. An example of the data returned by the option endpoint is illustrated in listing 5.11.

```
{
    granularity: [
        {key: "daily", value: "Daily"},
        {key: "monthly", value: "Monthly"},
        ...
    ],
    group_by: [
        {key: "cluster", value: "Cluster"},
        {key: "org", value: "Organization"},
        ...
    ],
    ...
}
```

Listing 5.11: Data returned by the option endpoint

5.3.4 Implementation

There are multiple challenges when working with schema creation and user input limitation. The logic behind it is not too complicated, but the implementation can get unreadable easily.

Chart IDs

The first challenge encountered was ID generation for new elements in the schema. The whole schema should be passed to the editor; therefore, for new charts, the highest ID is found and this number is incremented for each chart element. However, when editing the chart, the top element ID should not change. The editor is saving the top element ID when it is initialized. If a new chart is created, the editor saves the highest incremented ID in the chart. Whenever a new top element is created, this ID is assigned to this element. The editor cannot keep the IDs of the sub elements, because the number of elements depends on the chart. Therefore, all other chart elements, except the top level element, have different IDs after editing, even if the chart is returned to its previous state.

Limiting the form element values

When an input field's value changes, it can cause other input fields to change the type, value, or options. To make it more readable, each input field is rendered in its own function, where the function has to do three things:

- Generate the valid options for the input field.
- Set the value of the input field if the new options do not contain anymore than the original value.
- Render the correct type of input field with valid options.

To do this, all of the functions have access to all of the current values of the form. There is, however, one drawback to this approach. When an input field changes, a re-render is fired. This calls the other functions to render the other input fields; however, if one or more of the input fields does not have their selected option as a valid option, it sets the newly selected option. This action fires another re-render. If someone created a circular dependency, it would be able to cause infinite re-render cycles. However, these errors are easy to spot, since they break the entire application.

Rendering the correct chart

The chart renderings should reflect the settings from the form. Therefore, the single source of truth is from the form options. While the form is limited, some options should be filled automatically by default values. Then, the function for generating the form can have its own decision tree to select the correct elements to return, where each element can use different options from the form.

This approach is rather limiting when editing a chart. If the chart was not created with the chart editor, there could be loss of information when reading data in, causing, for example, styling to use the default value. However, the expectation is that only charts are eligible for edits which were created by the chart editor.

Summary

The single source of truth is the form options. When editing the chart the form options are set from the schema. The form and the chart are always changing, depending on the current values in the form. This approach allows the code to be split into multiple functions and the data to be edited independently from one another.

5.4 Data Driven Dashboard

The Data Drive Dashboard component is the most trivial of all. Also, this component does not have wireframes yet. Currently, its purpose is to call the chart renderer and pass the schema to it. It also features a button which creates a new chart with the chart editor and saves the created chart when the editor is closed. The component has four props, which are, excluding the schema, directly passed down to the chart renderer or the chart editor. The *apis* prop contains the sources for the chart editor. The *functions* prop is passed down to the chart renderer, too. The *schema* prop contains the the default schema, while the *onSchemaChange* contains a function which is called whenever a new chart is saved or a chart is edited.

In the future, this component will be responsible for custom buttons for reorganizing, hiding, enabling, adding, and removing the charts, as well as giving context or extra information to the user about the other two components.

5.5 Development Setup

This section describes the setup of the project. Information on the code and the package versioning, as well as the plugin which automates the process of publishing the package, will be provided here. Furthermore, this section describes the testing setup for the library and the demo application setup, which showcases some of the library's features.

Workflow and versioning

Before setting up the programming environment, a git repository is needed. Using git helps to manage changes in the project over time. This project's git repository is hosted on GitHub[10]. The source code of the project can be found at:
https://github.com/brumik/React-data-explorer

Hosting the git repository on GitHub can also leverage the *Issues* feature which allows other developers to report issues and makes the code more visible for later contributions.

The project is using the NPM package manager[19] for publishing the library. The package is available at https://www.npmjs.com/package/React-data-explorer. To publish the package to the NPM registry, the source code needs to be compiled from source and the version number should be set in the *package.json* file.

To make releases more consistent, more automatic, and emotion-free, the project has set up the semantic-release[3] plugin. This plugin fires when there is a new commit in the *main* branch (formerly master branch). From the commit messages, it determines which version number should be incremented depending on whether its a bugfix, feature, or breaking change. It automatically compiles the source, runs the tests, and increments the version number. After all of that is done, with a help of token, it publishes the new version to the NPM registry. Using a semantic release approach requires discipline from the developers when creating commit messages to work properly.

Testing

The project is set up such that, before publishing a new version of the library, all test have to pass. For unit testing React components, the most widespread tool is Jest[8]. However, Jest itself cannot test the DOM (Document Object Model), but rather works with the React DOM with React components. To extend the functionality and be able to test the DOM which is rendered in the browser, the React Testing Library[6] was added to the project.

There are few other libraries used for setting up complete testing environments to deal with *CSS (Cascading Style Sheet)*, icons, and other static assets. In addition, since the project uses TypeScript, the tests need to be able to run on TypeScript. This requires additional plugins like *ts-jest* and transformation settings to be able to run the tests.

The last important piece of testing was the ability to test asynchronous fetch requests. The fetch-mock[2] library makes it easy to catch requests to different URLs and return mocked up data, making the tests deterministic and giving more control over the data. This library also returns various kinds of errors in the response, allowing us to test for API failures.

A more in depth explanation of the testing tools is provided in chapter 6.

Demo application

The project also has a demo application to showcase some of the features of the library and to allow other developers to see the library in use. For compiling the library, as well as the demo application, and hosting the application on localhost, the Webpack[4] library is used.

However, the demo application uses real data from the Automation Analytics API. Since the demo app is not the part of the Automation Platform, the authentication is not available in the demo app for the API. Therefore serving the API has to be local too. Unfortunately, as it currently stands, the Automation Analytics back-end repository is not public and access is restricted to Red Hat associates.

Chapter 6

Testing

This chapter is dedicated to unit testing. During the development process, testing is one of the most important parts, because properly tested code means less maintenance time. Automated tests discover a lot of bugs when the code is modified.

6.1 Libraries

This section introduces the most important libraries used for testing. It provides how they work independently and describes their capabilities which were used to test the library.

Jest

The Jest testing library [8] was used for our testing purposes. This library was developed to easily and effectively test JavaScript code. A unit test can be created by the *test* or *it* functions. These two functions are equivalent to each other. The *it* keyword fits better, not only because it is shorter, but because the test should be defined as the *it („should do something“, function())*. This makes the tests more readable. The *expect()* function should appear at least once in the function to match values. If all of the *expect()* functions are passed, then the whole test passes, otherwise the test ends with a failure.

The need of asynchronous testing arises when the lazy loading function must be tested. Jest provides it in multiple ways. In the end, the 'asynch' and 'await' keywords were chosen to wait while the promise became fulfilled, and then tested if the tree was corresponding to the correct form.

The Jest testing library also provides snapshot comparisons. This can be done with the help of *expect(containerWithTheRenderedElements).toMatchSnapshot()*. On the first run, when the snapshot does not yet exist, it gets created in a separate file for each test suite. Then, all the subsequent test run snapshots are compared to the saved one. If they match, the test passes. If there is a difference, the test fails. When the snapshot changes because the implementation is changed, there is an option to update the saved snapshot.

An additional tool of the library is the ability to mock functions. By mocking functions, it is possible to return controlled or dummy values. A mocked function can additionally be tested in different ways as stored information about the number of calls and the about call arguments.

react-testing-library

The snapshots require rendered DOM and the react-testing-library[6] is built to supply this. It provides a *render* function, which renders a react component into a variable in the *jsdom* format. The *jsdom* format is the simpler representation of the true *DOM*, which means it is missing some advanced functions which are not required for testing.

Additionally, the react-testing-library provides addition functions for search elements in the DOM – by text, role, type, and many others. This helps when the tests need to verify if an element is represented in the rendered component or not.

For interacting with the rendered React component, the react-testing-library provides a handy *fireEvent* function. It needs the element as it's parameter on which the event should be fired and the event type to use. This allows users to click on elements or use keyboard inputs for input fields.

When rendering asynchronous React components or components which fetch asynchronous data, it is necessary that the element has finished the request before the tests are run. For this use case, the *waitFor* asynchronous function can be used. It waits until the supplied function returns true or fails with a timeout.

fetch-mock

The fetch-mock library provides an easy interface to mock-up *fetch* calls to different URLs. Mocking up API requests allows testing without an actual API and, furthermore, it gives full control over the data. This makes easier writing tests, since the exact data is defined and it is then supplied to the components. The fetch-mock library is capable of returning data or different error types with custom messages.

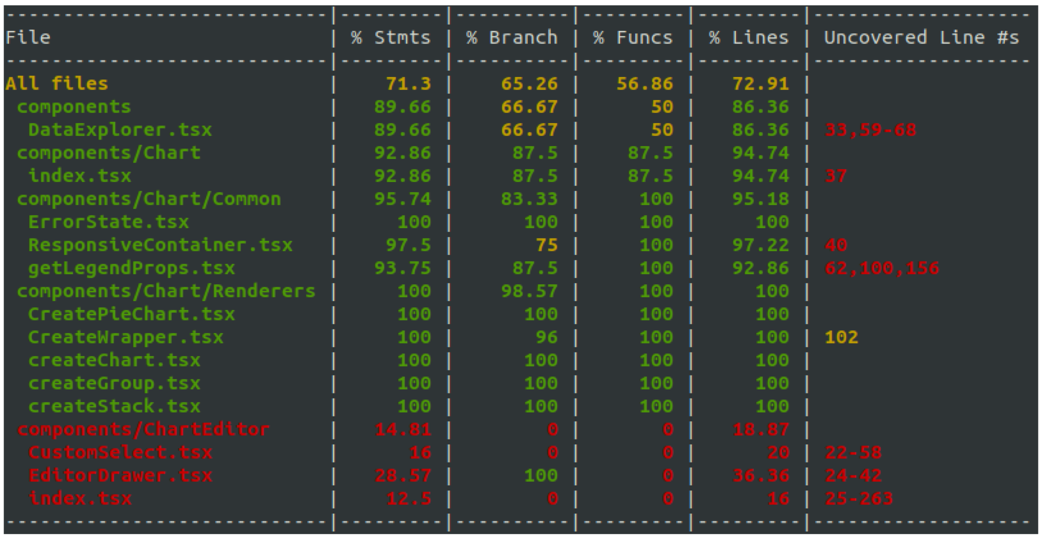
It uses a similar interface to the Jest function mock-ups. It has methods for checking the number of calls made and call parameters. This capability is extremely useful when the tests need to verify if the component is passing the right parameters when fetching from the API.

6.2 Testing the library

The goal of the testing is always to reach the highest coverage possible. However, the project is made up from two main parts: the chart renderer and the chart editor. The chart renderer can be considered to be stable, but the rendered elements are not due to big changes. The chart editor is still missing a proper UX design and will eventually undergo a lot of changes. Testing the UI for the chart editor would become obsolete in no time at all. Therefore, the testing focuses mainly on the chart renderer. The final coverage of the chart renderer is around 92% as shown in fig. 6.1. Some functionalities, like resizing the browser window, were not testable by Jest unit tests.

Some functions can be tested as standalone pieces, while other functions are tied to the underlying charting library. These functions are tested through the components using the charting library.

The easiest parts to test are the functions without asynchronous calls. These functions are followed by the functions which make asynchronous calls to the API, like the default *fetch* function. All of the functions are tested independently – the asynchronous ones with the fetch-mock library.



```
----------------------------|---------|----------|---------|---------|-------------------
File                        | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------------------|---------|----------|---------|---------|-------------------
All files                   |    71.3 |    65.26 |   56.86 |   72.91 |
 components                 |   89.66 |    66.67 |      50 |   86.36 |
  DataExplorer.tsx          |   89.66 |    66.67 |      50 |   86.36 | 33,59-68
 components/Chart           |   92.86 |     87.5 |    87.5 |   94.74 |
  index.tsx                 |   92.86 |     87.5 |    87.5 |   94.74 | 37
 components/Chart/Common    |   95.74 |    83.33 |     100 |   95.18 |
  ErrorState.tsx            |     100 |      100 |     100 |     100 |
  ResponsiveContainer.tsx   |    97.5 |       75 |     100 |   97.22 | 40
  getLegendProps.tsx        |   93.75 |     87.5 |     100 |   92.86 | 62,100,156
 components/Chart/Renderers |     100 |    98.57 |     100 |     100 |
  CreatePieChart.tsx        |     100 |      100 |     100 |     100 |
  CreateWrapper.tsx         |     100 |       96 |     100 |     100 | 102
  createChart.tsx           |     100 |      100 |     100 |     100 |
  createGroup.tsx           |     100 |      100 |     100 |     100 |
  createStack.tsx           |     100 |      100 |     100 |     100 |
 components/ChartEditor     |   14.81 |        0 |       0 |   18.87 |
  CustomSelect.tsx          |      16 |        0 |       0 |      20 | 22-58
  EditorDrawer.tsx          |   28.57 |      100 |       0 |   36.36 | 24-42
  index.tsx                 |    12.5 |        0 |       0 |      16 | 25-263
----------------------------|---------|----------|---------|---------|-------------------
```

Figure 6.1: Unit test coverage

A bigger challenge was was the testing of the React components. These are tested mostly with snapshots. The *jsdom* format does not support selecting *svg* image elements and inspecting them. Therefore, all the data is mocked up with the fetch-mock function, allowing different charts to be rendered. The rendered charts are checked manually and, if they look as they should, a snapshot is created. The chart on-hover functions are currently impossible to test.

The interactive legend for the chart is tested by interacting with the rendered component. It is done with the *fireEvent* function. Then, the tests wait for to be re-rendered. When the re-rendering is done, a snapshot is created in the same fashion as the other snapshots.

In the future, tests will be updated and extended as the charting library continues to evolve with user feedback, UX design, and unexpected bug fixes.

Chapter 7

Future development

The library developed within this thesis is a first version, which is planned to be developed in the future. The library is currently being migrated into Asnible Automation Analytics. Since it is a bigger system, it takes more time. Future development will be an iterative process involving other developers and, eventually, the end users too. There is a possibility that the library will be used platform-wide on the Red Hat Ansible Automation Platform.

The library currently lacks a proper UX design. The UI mocks will be created with the UX team and the Automation Analytics developers. It is foreseeable that the library will be able to organize the charts into a grid like pattern and store this information in the schema.

The chart renderer will eventually have to work with continuous data streams, too. This is especially important when the user should request to see live data about his/her virtual machines or servers. This feature will require internal optimization and probably external optimizations in the underlying library, too.

Currently, there are two new open discussions with the PatternFly 4 Charts team. One of the discussions is around trying to make the interactive legend more consistent to use across different chart types. The other one is a possible bug where the PatternFly 4 pie chart is passing an invalid option to the Victory chart, which creates warnings when running the tests. Neither of them is a blocking issue for the created library, but they can make the code base more simple if solved.

The exact API is still under discussion for the charts. Soon, the API will provide the legend for each chart in the *meta* field. This will make parsing the legend obsolete and will ensure the correctness of the legend for any chart. Furthermore, the API will provide an endpoint where the charts schema can be uploaded and saved in permanent storage. There are also ongoing discussions on how to make the returned data format more convenient for the charts. In the future, the API may also allow more options and multi grouping, which could require 3D chart support from the charting library.

Chapter 8

Conclusion

The goal of this thesis was to design and implement a charting library in React capable of rendering charts from a schema, and provide a user interface for creating and editing the schema for Automation Analytics. The library was implemented with the help of TypeScript and the React framework, and built upon the Victory charts charting library.

When developing the library, the Automation Analytics API was taken into consideration as well as the feedback for the format of the schema from other developers. When problems were encountered with the underlying libraries, the respective developer teams were contacted and fixes or features were proposed.

The resulting library is shipped as an NPM package and it is available through the NPM registry for public use. The package contains three React components, which can be used together or as standalone components. The primary feature of these components is the fully customizable and editable chart rendering from schema. The schema only contains serializable objects, which makes it possible to store the schema in a database. The user interface for the chart editor allows for the creation and addition of the following chart types: simple line, bar, area or scatter chart, stacked charts, grouped bar charts, and pie charts. The data for the charts and for the options in the chart editor are provided by the API. The chart renderer also makes it possible to visualise and detect anomalies in the data.

The charts are unit tested with the Jest testing library and additional plugins, like react-testing-library and fetch-mock. The unit tests over the more stable chart renderer reached 92%+ coverage.

As the component is required by Automation Analytics, the development will continue after this thesis. The library will receive a UX design and, with the help of user and developer feedback, it will be iterated over. The API providing the data for the charts will be improved upon, too, which will mean new features for the chart renderer, like continuous data stream processing.

Bibliography

[1] *Nivo* [online]. 2020 [cit. 2020-12-16]. Available at: https://nivo.rocks/.

[2] *Fetch-mock* [online]. 2021 [cit. 2021-04-20]. Available at: http://www.wheresrhys.co.uk/fetch-mock/.

[3] *Semantic Release* [online]. 2021 [cit. 2021-04-20]. Available at: https://semantic-release.gitbook.io/semantic-release/.

[4] *Webpack* [online]. 2021 [cit. 2021-04-20]. Available at: https://webpack.js.org/.

[5] Chartlog, I. *Chartlog* [online]. 2020 [cit. 2020-12-17]. Available at: https://www.chartlog.com/product/dashboard/.

[6] Dodds, K. C. *React Testing Library* [online]. 2021 [cit. 2021-04-20]. Available at: https://testing-library.com/.

[7] Education, I. C. *REST APIs* [online]. 2020 [cit. 2020-10-17]. Available at: https://www.ibm.com/cloud/learn/rest-apis.

[8] Facebook, I. *Jest* [online]. 2021 [cit. 2021-04-20]. Available at: https://jestjs.io.

[9] Formidable Labs, L. *Victory* [online]. 2020 [cit. 2020-12-16]. Available at: https://formidable.com/open-source/victory/.

[10] GitHub, I. *GitHub* [online]. 2021 [cit. 2021-04-20]. Available at: https://github.com/.

[11] Group, R. *Recharts* [online]. 2020 [cit. 2020-12-16]. Available at: http://recharts.org/.

[12] Inc., F. *React, A JavaScript library for building user interfaces* [online]. 2020 [cit. 2020-12-09]. Available at: https://reactjs.org/.

[13] Inc., G. *Nivo GitHub Reprository* [online]. 2020 [cit. 2020-12-20]. Available at: https://github.com/plouc/nivo.

[14] Inc., G. *Recharts GitHub Reprository* [online]. 2020 [cit. 2020-12-20]. Available at: https://github.com/recharts/recharts.

[15] Inc., G. *Victory GitHub Reprository* [online]. 2020 [cit. 2020-12-20]. Available at: https://github.com/FormidableLabs/victory.

[16] International, E. *Standard ECMA-262* [online]. Ecma International, july 2018 [cit. 2020-12-12]. Available at: https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

[17] Labs, G. *Grafana* [online]. 2020 [cit. 2020-12-17]. Available at: https://grafana.com/.

[18] Microsoft. *TypeScript* [online]. 2020 [cit. 2020-12-12]. Available at: https://www.typescriptlang.org/.

[19] npm, I. *NPM Library* [online]. 2021 [cit. 2021-04-20]. Available at: https://www.npmjs.com/.

[20] Red Hat, I. *Ansible Automation Platform* [online]. 2020 [cit. 2020-10-20]. Available at: https://www.ansible.com/products/automation-platform.

[21] Red Hat, I. *Patternfly Charts* [online]. 2021 [cit. 2021-03-26]. Available at: https://www.patternfly.org/v4/charts/about/.

[22] Reviewers, T. *Chartlog Review – Best Day Trading Software for Analytics?* [online]. 2020 [cit. 2020-12-20]. Available at: https://www.tradingreviewers.com/chartlog-review/.

[23] Swagger. *OpenAPI Specification* [online]. 2020 [cit. 2020-10-17]. Available at: https://swagger.io/specification/.

[24] Valcheva, S. *Types of Graphs and Charts And Their Uses* [online]. 2020 [cit. 2020-11-08]. Available at: http://www.intellspot.com/types-graphs-charts/.

[25] Wright, A. *JSON Schema: A Media Type for Describing JSON Documents* [online]. 2020 [cit. 2020-10-17]. Available at: https://tools.ietf.org/html/draft-wright-json-schema-00#section-4.2.