Thuy Phuong Nhi Tran

COMPONENTIZED VISUALIZATION COMPONENTS WITH MICRO-FRONTENDS

Faculty of Information Technology and Communication Sciences (ITC)
Bachelor's thesis
Supervisor: Kari Systa
Publish date:

ABSTRACT

Thuy Phuong Nhi Tran:
Componentized Visualization Components With Micro-Frontends
Bachelor's thesis
Tampere University
Bachelors Degree Programme in Natural Science and Engineering
Publish date:

With a rapid increase in demand of using the Internet, web application has become a key and essential role in today's world. In the near future, billions of devices will be connected to the Internet for sharing values. Therefore, having cutting-edge web technologies urges companies and researchers to find out a novel approach for developing, testing, and deploying web application.

The Microservices concept has become a "trending" concept that almost all developers are familiar with this architecture for solving problems caused by a single monolith application in the server side. With the idea of decomposing a traditional large monolith application into a smaller independent application, Micro-Frontends architecture has been introduced as "microservices" for client side. It has gained remarkable popularity and has been adopted by several established companies such as Starbucks, Ikea, and many others. However, The concept of Micro-Frontend is quite new and many organizations still hesitate to adopt this architecture due to a lack of knowledge and skills.

This thesis will provide a basic concept concerning the advantages of adopting Micro-Frontends architecture in some cases. Also, it brings an observation on a real project developing application with Micro-Frontend patterns.

Keywords: web application, micro-frontend, visualizations, microservices.

PREFACE

I would like to express my gratitude and appreciation to all VISDOM project members, especially my supervisor, Kari Systä for his support during my time working as Research Assistant, and research and writing this bachelor thesis. Also, I would like to show a special esteem to Ville Heikkilä, Henri Bomström, and Hong Nghia Duc for offering valuable knowledge and tools that help me during the conducting experience and researching phase.

Tampere April 2022

Thuy Phuong Nhi Tran

CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

SPA Single-page application, web application that interacts with users by dynamically loading parts of the current page, rather than reload the whole page.

MQTT Message Queueing Telemetry Transport- a lightweight, publish-subscribe network protocol that transports messages between devices.

EKG Electrocardiogram, a type of visualization imitating heart’s electrical activities.

URL Uniform Resource Locator, a reference to a web resource that specifies its location on a computer network, or the Internet.

1 INTRODUCTION

In the rapid development of the Internet, where web application plays a crucial role, a demand for a new and innovative web technologies become a major and important task for many industrial companies. Speaking about the world of client-side, over years, there are many new front-end frameworks that are continuously introduced such as static HTML files, server-side rendering (SSR), or Single-page application (SPA). These traditional architectures work well for the traditional web applications due to their small number of users, complexity, and low real-time communication between visualizations. However, most of them become monolith front-ends, which leads to many problems such as when the application grows in both size and complexity, it becomes hard to scale when multiple teams try to modify the same front-ends simultaneously, or in another situation when the front-ends are needed to be reused in another application. Therefore, looking for an innovative ways to develop, and maintain the application becomes a crucial task.

Micro-Frontends was introduced as one of the innovative ways to change the a front-end application implementation approach since the introduction of SPA. Micro-Frontends helps solving many challenges of web application when it gets bigger in size and complexity in many situations. The objective of this thesis is to provide an introduction of Micro-Frontends and its strengths, benefits, and potentials in multiple aspects. Furthermore, the thesis also discusses the adoption and implementation of Micro-Frontends in an existing research project.

This document is structured as follows. Chapter 2 discusses briefly the background of Micro-Frontends regarding its architecture, strengths, and benefits of adopting Micro-Frontends in some situations. Chapter 3 gives a deep understandings of overall goal and current architecture overview regarding Micro-Frontends of an existing research. The way of implementing and designing prototype visualizations with Micro-Frontends architecture that are use in the project will be discuss more preciously in chapter 4. Chapter 5 will analyse the visualization implementations with the project architecture, and the current prototypes with the concept of micro services. With all the analysis and observation of the adoption of Micro-Frontends architecture in this research project will lead to the general conclusion, which will be discussed briefly in the last chapter of this thesis, chapter 6

2 BACKGROUND

Development teams have always been looking for new way to improve the performances of the application to continuously deliver their values to the customer. In recent years, Micro-Frontends architecture has gained a great attention thanks to its potentials for solving the current challenges in web application development. This chapter will introduce briefly the Micro-Frontends architecture and its strength, benefits, and potentials for future development.

2.1 Micro-Frontends in a nutshell

Micro-Frontends, which was introduced in 2016 [1], enables a large and complex monolith application to decompose into smaller individual or semi-individual front-end applications, called "micro-apps". Each of these applications consists of different frameworks or libraries that work loosely together.

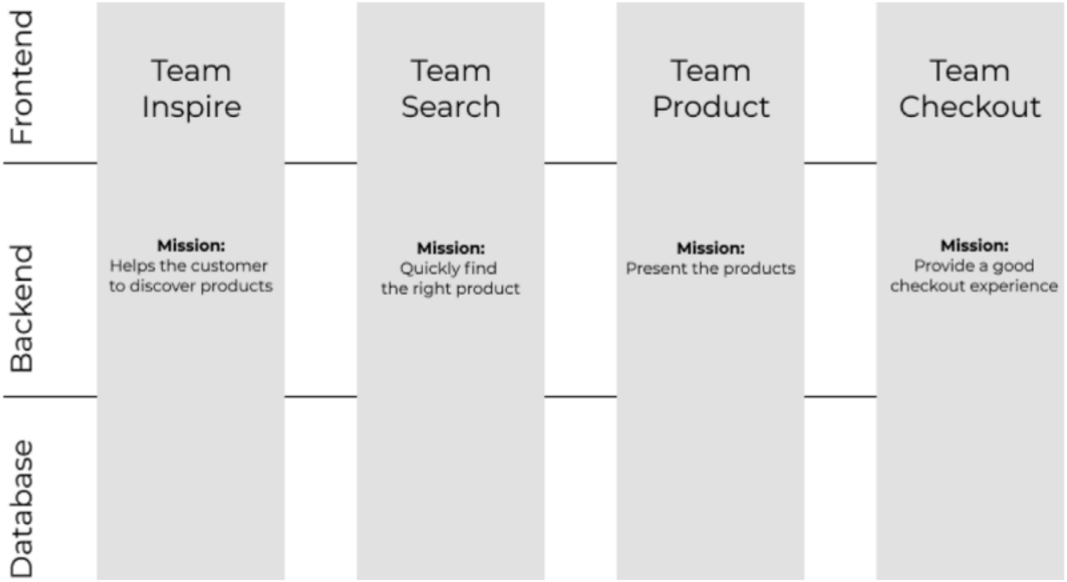


Figure 2.1 End-to-end example teams with Micro-Frontends architecture![2]

Microservices architecture is an architecture style that allows one big multitasking application to decompose into smaller applications that are responsible for only one task. To serve a single user request, the request will be proxied to server, then microservices-based application will make use of many internal microservices applications to compose the response [3]. With microservices architecture, the responding time to the request sent by client side can be reduced significantly as in stead of one monolith server application processes the request step-by-step, multiple micro applications can process the parts of request at the same time.

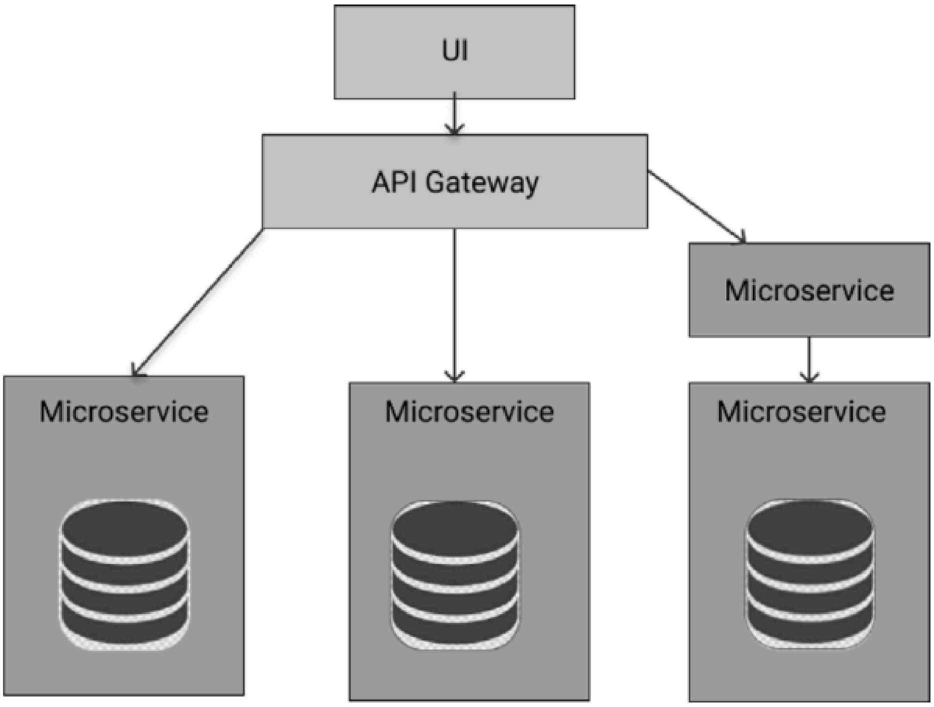


Figure 2.2 *Microservices architecture [2]*

Micro-Frontends extends the concept of micro-services of backend to the frontend side. The idea is to break down a large simple monolith application into smaller applications that can be developed and implemented individually with different technologies, and single-spa helps in achieving that goal. Single-spa takes inspiration from modern framework component life-cycle for the entire applications [4].

Micro-Frontends shares main principles, benefits, and issues of microservices [1]. The current trend is to build a web application that is feature-rich, powerful, and business sub-domain, which matches with the concept of Micro-frontends. The main idea of Micro-Frontends is simply the web applications that are the composition of features, which are owned and developed independently by a development team. Each team should have only one domain to take care of and develop end-to-end. With ability to develop, test and deploy independently, development teams are able to build applications that are isolated and coupled services [2]. One of the Micro-Frontends architecture purposes is to divide the development teams responsibilities vertically, while it is horizontal division in traditional monolith front-end architecture, as shown in **Figure 2.1**. The developers can be technologically agnostic and develop their own "micro-app" with their own technologies since Micro-Frontends is independent of technologies.

The idea of Micro-Frontend is not new. In the past, there was long, bulky term Frontend Integration for Verticalised Systems [5], which share the same approach, however, Micro Frontends is clearly a more friendly and memorable term. Micro-Frontends architecure is nowadays adopted by several large industrial companies such as DAZN, Ikea, New Relic, Starbucks, Zalando, and many others [2].

2.2 The strengths and benefits of Micro-Frontend architecture

While using traditional front-end frameworks such as SPA, each modification leads to redeploy the whole application, and full deployment takes long time or causes many unexpected issues. There for developers will block on each other with small changes. Therefore, there are must be reasons why Micro-Frontends draw attention since its introduction. Here are some of the benefits of Micro-Frontends:

- **Simple, decoupled codebase:** With Micro-Frontends architecture, each individual micro front-end applications definitely has smaller code dependency and complexity compared to the traditional monolithic front-end architecture. Additionally, the complexity from unintentional and inappropriate coupling between components is avoided [6].

- **Deployment and development independence:** It is believed that each Micro-Frontends have its own continuous delivery pipeline, which builds, tests, and deploy [3] or any other ways that are independent with the other Micro-Frontends. Therefore, as the whole large monolithic application is decomposed into smaller front-end applications, so the delivery of each of these application do not affect the other ones.

- **Better scalability and incremental upgrades:** In Micro-Frontends architecture, one person or one development team will take care of one micro

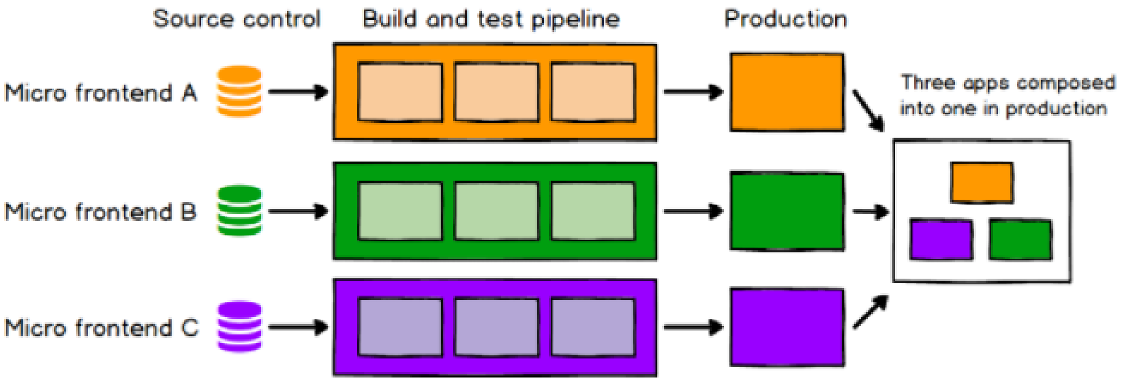


Figure 2.3 Each micro frontend is deployed to production independently [6]

front-end application, so they know the application appropriately. Therefore, the Micro-Frontends can be upgraded, modified, or even rewritten more smoothly and causes less code-breaking risk.

- **Autonomouse development teams :** In the traditional monolithic front-end applications, in a large application, there might be teams that are responsible for one specific aspect such as styling, validation. However, with Micro-Frontends architecture, one teams can have full ownership of the whole Micro-Frontends application [6]. As they only have one domain to handle, the development process can move quickly an effectively.

- **Reusability:** One component/visualization can be built and deployed independently, so many teams can re-use the code in different application or places.

- **Technology agnosticism:** With Micro-Frontends, it is not a big problem when different developers in a team use different technologies such as React, Vue, Angular for building visualizations, since Micro-Frontend is independent of technologies.

- **Learning Curve:** For people who join in the late development phase, it is much easier for them to understand the smaller applications than the one big monolith application with thousand lines of code and dependencies.

As mentioned above, the developments teams can benefit a lot when adopting the Micro-Frontends architectures. However, every technologies have its benefits and issues. Since different projects or applications differ in scale and purposes, understanding the issues and whether trade-offs overcome the drawbacks when using Micro-Frontends is important.

2.3 Micro-Frontends implementation approach

There are different frameworks that can speeds-up micro front-end application development. For example, Bit is one of the most popular frameworks for developing micro front-end application. It allows developers/development temas to create front-ends by using independent components, which are then can be used by the other development Teams [7]. This section only introduce single-spa framework, which is used for developing micro front-end application in the research project ,VISDOM, in this thesis. The reason is VISDOM requires the micro front-end implementation option that allows micro applications, not just a component, (i.e. visualizations) to be implemented independently and with different front-end frameworks.

2.3.1 Single-Page Application

In Single-page Application (SPA), only one HTML file is loaded to the browser [2] and it does not need to refresh or load the whole page when the user interacts with the page, which gives the user a smoother user experience. The overall design of SPA is almost the same as the traditional design, the presentation logic resides in the client, and server transactions can be data-only. The single-page term refers to the single HTML file, which is loaded fully once by the browser. It will provide a rooting point (basically a single HTML element [2]) for JavaScript applications to be loaded with HTML files (also with images, css, and so on). The views are portions of the Document Object Model (DOM), not the whole page, so when the user makes a change, the view is generated in the browser and dynamically attached to the DOM. In short, when there is a need for change in view, it will compare two DOM and only re-render the needed part.

In Single-Page Application, the server is the bridge for the front-end React app to connect and communicate with the database. The server can be implemented in multiple programming languages such as NodeJS, Python, C, and so on (it depends on developers preferences). The server provides the API for the front-end application. There might be several endpoints, which will provide specific data or responses based on React app needs/ requests. One option used widely in React App is using Axios for query data or sending the request to the back-end.

2.3.2 Overview of Single-spa

Single-spa is a framework that brings together multiple JavaScript micro front-end application in a single front-end application. Multiple Applications are combined into one single application regardless of the framework or library.

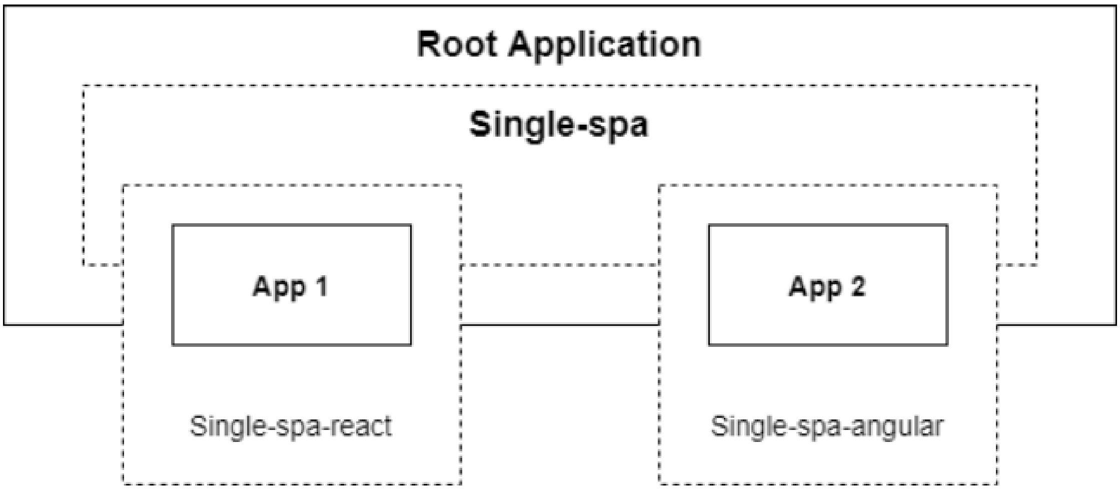


Figure 2.4 *Application using single-spa framework [8]*

A single-spa application consists of two main components: root applications (also known as root-config), and a number of application. Firstly, an applications part can be thought as single-page application packaged up into modules. Each application must know how to mount, unmount, and bootstrap itself from the DOM. It seems like a SPA, but the main difference between a traditional SPA and single-spa applications is that all applications are able to works properly with each others even they do not know the other's work flow and HTML page. Lastly, the other main component is a single-spa root config, which renders the HTML page and the JavaScript code that registers applications [4]. Each application should be registered with three parts: a name, a function to load the application's code, and a function in order to determine the active status of the application. For example, the React SPAs are applications in this case, when they are active, they should be able to listen to URL routing events and put the content on the DOM. Otherwise, when they are inactive, they do not listen to the URL routing and are totally removed from the DOM [4].

It is worth to note that single-spa is an advanced architecture, which is different from all typical behaviours of front-end application. Therefore, it requires changes to existing paradigms and a good understanding of underlying tools. Single-spa applications are available in multiple web browsers such as Chrome, Firefox, Safari, Edge, and IE11 (with polyfills) [4].

3 OVERVIEW OF THE RESEARCH PROJECT- VISDOM

One of the main purposes of the project is to create configurable dashboard that can display various visualizations with different contents based on user roles and preferences. A dashboard is not only for directed data display, but it can also include diagnostics tools that help the stakeholders to investigate possible problems in their projects [9]. The VISDOM project adopts the Micro-frontends architecture, where each visualization is developed and deployed independently by one development team.

This section will introduce the overall architecture of VISDOM project and discuss deeply the how the visualizations negotiate the constraints the configurable dashboard with the Micro-frontends concept. This will show how developers benefit and face challenges with Micro-Frontends in a real project.

3.1 Overall of VISDOM project

"Visualization is a powerful method for the internal communication within a team of developers, but even more, it is useful in cross-disciplinary communication with various stakeholders, such as operations and business management" [10]. There are many software development tools that already provide many kinds of visualizations. However, utilization of data from multiple source is still on another level of research prototypes. VISDOM introduce a novel and innovative approach of developing new types of integrated visualizations that combines data from several sources. The VISDOM project provides various visualizations from simple to advanced ones.

VISDOM contains six work packages which are geared towards the real-time visualizations development. The visualizations are aimed for improve efficiency, and customer satisfaction [10]. The relation of six packages is depicted in **Figure 3.1**.

The main objective of WP1 is gathering requirements for developing real-time visualization in WP2-WP4. The implementation of visualizations in WP3 requires data which is collected, processed, and analyzed by techniques provided in WP2. The implementation from both WP2 and WP3 are used in WP4 for demonstration, experimentation and validation [10].

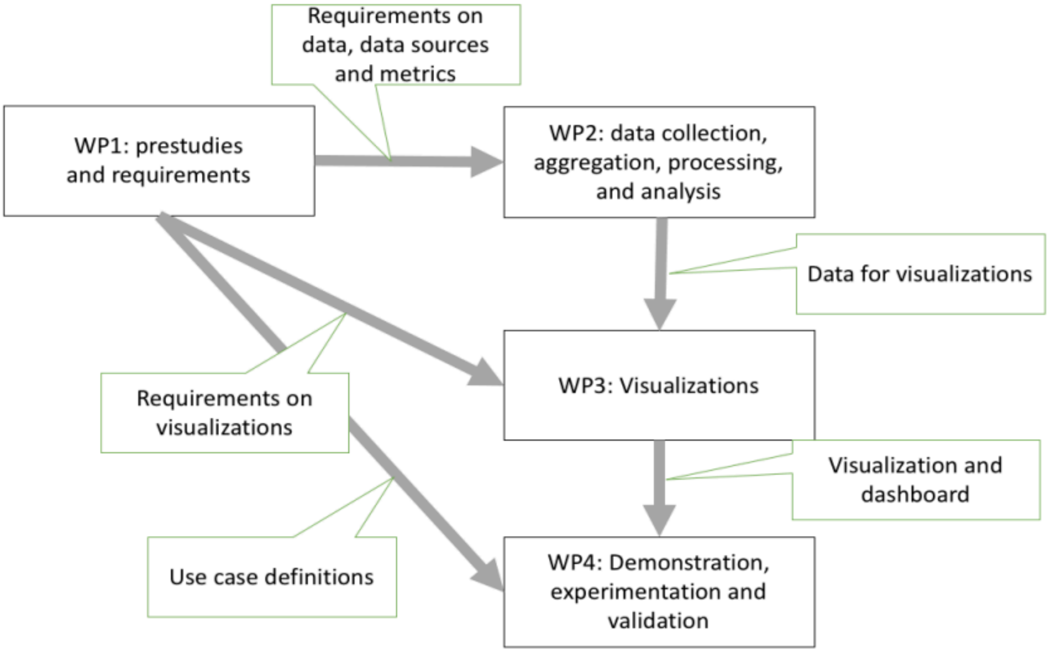

Requirements on data, data sources and metrics
WP1: prestudies and requirements
WP2: data collection, aggregation, processing, and analysis
Data for visualizations
Requirements on visualizations
WP3: Visualizations
Visualization and dashboard
Use case definitions
WP4: Demonstration, experimentation and validation

Figure 3.1 *Work packages of VISDOM project [10]*

WP1 conducts state-of-practice investigations starting with use cases where VISDOM has identified the needs for real-time visualisation based on aggregated data from different sources. Below is a identified tentative list of use cases, which shows that VISDOM has experts in software visualizations in both universities and companies cases.

- **Use case 1: visualization of quality aspects.** In quality aspect, DevOps has two facets, which require visualizations to achieve the goals. Firstly, the first facet is the internal quality of a software product and its associated process. The other one is the users perceive quality of the product based on thier interactions and experience. The overall goal is to provide high-level visualization that help developers improve their development process and code quality (i.e. technical debt, consistent style) [10].
- **Use case 2: Entering international service business.** With data, metrics, visualizations and dashboard can be a valuable tool for changing the mode of operation, which should reduce the obstacles to enter the international business. In this use case, the usefulness of different types of visualization methods is observed. For collecting the feedback from stakeholders, a demonstration of how visualization dashboard can help com-

pany to lower the barrier of entering and extending international market [10].

- **Use case 3: visualization in teaching.** This thesis only focus on this use case. During the learning process, the problems that students facing remain unknown until the exam takes place or an assignment is due. At that point, it is too late for course staffs to help these student to pass the course. And the situation even worse when students have distance learning, where there is little or even no direct contact. Even in the cases where there are good contacts to students, the students may be intentionally or unintentionally overly optimistic about their current situation [10]. In this case, visualizations can be utilized in some courses in the same way as in software companies. With visualizations, course staffs can identify the abnormal of course implementation or students with the changes in visualization patterns.

With motivation mentioned in use case 3, an observation is conducted on Programming 2 course which is implemented by Tampere University. There are two course implementation that are currently in use: a two-period implementation, and a one-period one (also known as double-speed). The initial idea is to help teacher to look into students that are falling behind a certain threshold, and to find a better way to help these student to get back to the expected track. There are many attributes that are used in visualizations to support the idea of the teach case. For example, the GitLab provides the number of commits, the number of submissions, and points of each exercise for each student. The visualizations will display these data in multiple ways that can help teacher to identify pain-point in a mass course specifically from the viewpoint of tasks. Additionally, visualizations helps teachers to have a comparison between different course implementations. In idealistic world, teachers also would like to reach out to students who are falling from the course. However, it is not possible due to a vast number of students attending the course, so the idea is to identify the exercises/tasks that are overly difficult for a large portion of attending students. Then, the teachers can make modification or give more support to help these student to complete the course.

Among visualizations, for teaching case, there are two visualizations that bring most valuable information for teachers: Status and EKG view:

- **Status view:** has four different modes: points, commits, exercises, and submissions. Points and exercises modes shows total and missed points/exercises of all students by stacked bar; each bar presents for one student.

The other modes indicate the number of commits or submissions of each exercise by color schema. The more commits or submissions are, the darker the color is. An example of Status view with course implementation 90 in week 9 with submission mode is shown in **Figure 3.2**.

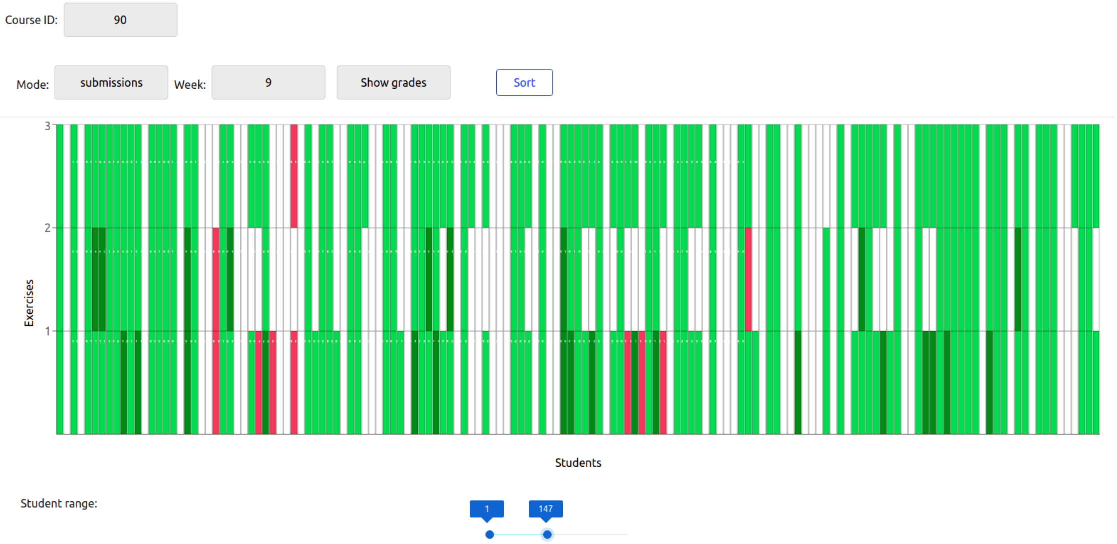
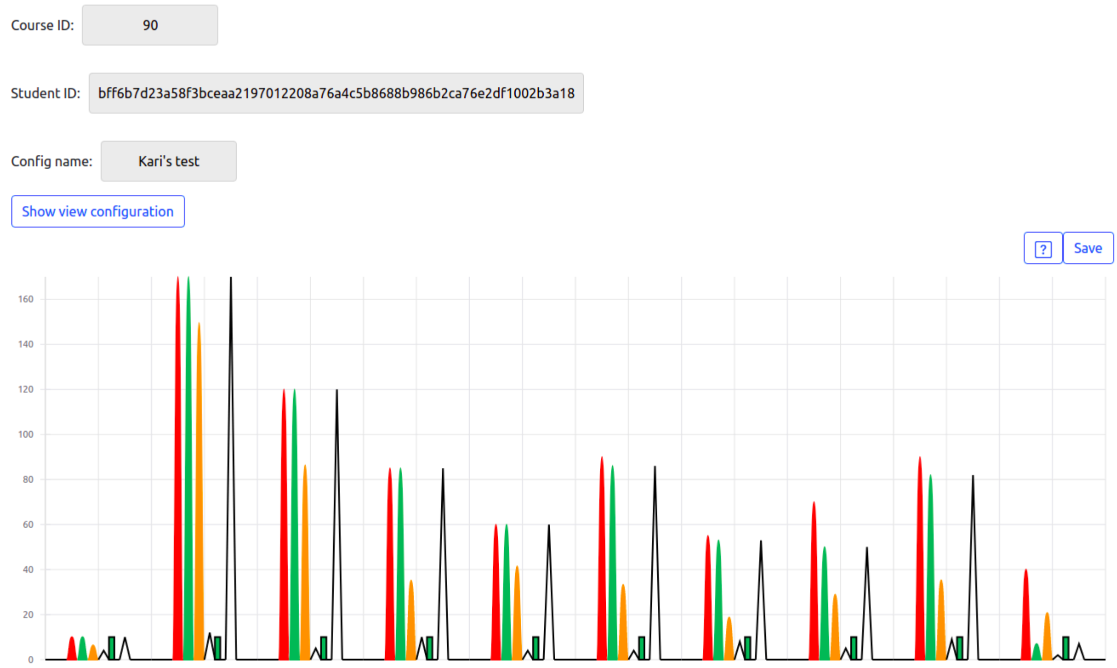


Figure 3.2 *Example of status visualization*

- **EKG view:** Electrocardiogram is a method recording the heart electrical signal for the doctor to determine the health status of patients. EKG in VISDOM is based on a similar idea. A set of input data is used to generate visualizations based on users' interests and preferences that different stakeholders and expertise can analyse. The EKG view is a good example of genetic visualization as it can be configured to show multiple inputs in the same graph based on different stakeholders' interests and preferences, which is also the main purpose of the Dashboard Composer. From an abnormal point in repeated patterns, stakeholders can detect and analyse the problems from the root. The EKG is utilized for teaching case to find out the workflow, workload, and progress of a single student with repeated patterns that problems can be quickly determined as variations in patterns. Therefore, course assistants or teachers can help that student to get back to the suitable track as soon as possible. An example of EKG visualization of course implementation 90 is shown in **Figure 3.3**.

There are at least two roles in teaching case. First role is teachers who are responsible for course implementation. They may show interests in the overview of the course for identifying the difficulty of exercises or tasks, which helps them to adjust these exercises in an appropriate way. The other role is teacher assistant (TA), who helps students with their tasks or project during the course.



Course ID: 90

Student ID: bff6b7d23a58f3bceaa2197012208a76a4c5b8688b986b2ca76e2df1002b3a18

Config name: Kari's test

Figure 3.3 *Example of EKG visualization*

Therefore, in this case, visualization providing further information of a single student will be more suitable for the TA. For teaching case, there are two dashboards that are currently in use: Dashboard Composer utilizing Micro-Frontends architecture, and educational dashboard which is a monolith front-end application.

3.2 Overall Architecture

Overall, the architecture of VISDOM project comprises of three main elements: data sources, data management system, and Visualizations and Dashboard. The technology value chain is depicted in **Figure 3.4** below:

First, Data sources are software engineering databases, repositories, and tools such as GitHub, Trello. In this project, methods and tools (open source by default) are built for collecting data from multiple sources [9]. Data management system provides unified interface for collecting and storing data. The data collected from the data sources will be processed and transformed in Data management system, which provides the formatted data to the last main component: Visualizations and Dashboard. Lastly, the last element comprises of two smaller components: configurable dashboards, and visualizations. These visualizations are responsible for displaying data with interactive graphs, charts,

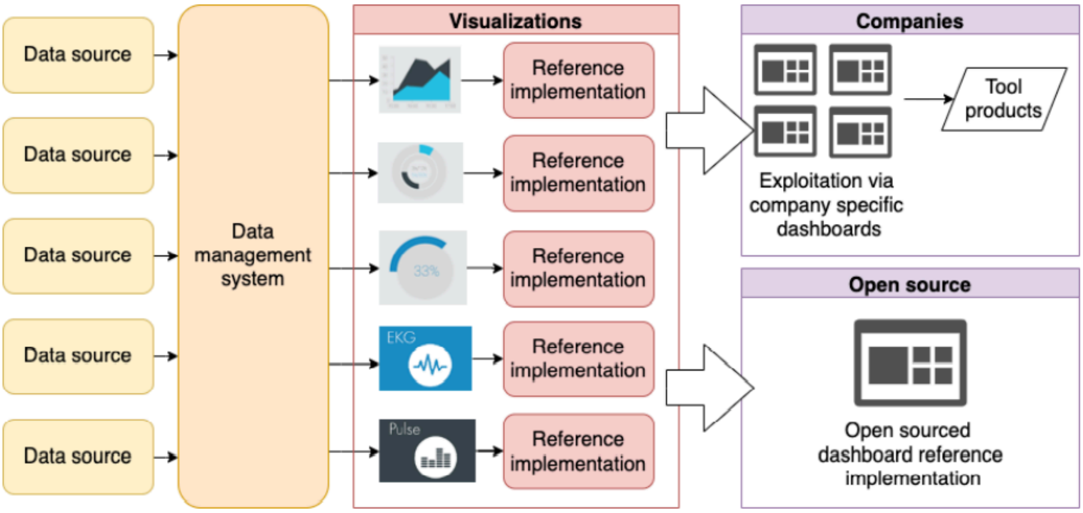


Figure 3.4 *Technology value chain of the VISDOM project [9]*

etc. Furthermore, they provide advanced functionalities such as responding to different stakeholders' input such as zooming in or out, and can be configured based on stakeholders' references. Dashboard composer matches the stakeholders' need with the corresponding visualizations with customized view. Users can request for multiple visualizations at the same time.

This thesis will focus on the last component among three main components: Visualizations and Dashboard. With this component, it can tell how this research project benefits from the Micro-Frontends architecture and how different visualizations get the external data and communicate with each other. Therefore, being familiar with the technology value chain is important for the implementation of Visualizations.

3.3 The reference architecture

In general, reference architecture in software development field provides a methodology or set of templates that are used as constraints for more concrete architecture [11]. The reference architecture provides guidance or solutions for applying specific patterns to solve further particular problems happened during the development phases. In short, the reference architecture gives a development team the way that they are shooting towards.

In the VISDOM project, one of the most crucial task is to provide different visualizations with different configurable features in multiple dashboards based on the role of users. There are more than one dashboards that display the same visualization. The idea of building multiple single monolith dashboards can cause many problems. It can be simple and quickly at the beginning phase of project. However considering a case when one new feature is added to a visualization of a dashboard, this also calls for modification of this visualization in another dashboard, which can be negotiable if there are only two dashboards. However, problems raise when the number of dashboards increases, multiple modification can cause process delayed and unexpected issues. Therefore, the designs of the visualizations should accompanies with the reusable reference implementation [9].

Micro-Frontends architecture can solve problems that are mentioned above from re-usability to visualization components independent implementation and parallel deployment. With Micro-Frontends architecture, the case of a single monolith components that can handle and response to each and every requests is avoided.

3.4 Proof-of-concept implementation

From the discussion in the previous section, the VISDOM project can gain many benefits from adopting Micro-Frontend architecture. This section will provide discussion of the detailed design of visualization components with Micro-Frontends and how a configurable dashboard composer selects and displays specific visualization based on users' interest.

3.4.1 Visualizations

VISDOM project takes inspiration from medical diagnostics for visualizations. Conventional visualizations often present a single viewpoint to the underlying data [9]. The goal of VISDOM project is to provide visualizations that are intuitive and easy to understand for variety of stakeholders [10]. Therefore, novel visualizations should provide a good summary of the most essential information of a certain aspect of software process. The information should be shown in the way that every stakeholder can straightforwardly understand regardless their background, so that stakeholders can investigate the details or specifics to detect the issues from the root through visualization. As mentioned, the one of the main purposes of this research project is to display visualizations based on role of the users and their preferences. Therefore, these visualizations can be configured for stakeholders references and especially for their role, for example in teaching case, if the user is a teacher assistant, he or she cannot access to all the modes of one visualization such as saving configuration preference. The limitation for specific stakeholders does not only come from the visualization components, but also the adapter, where the data is queried. When the users make requests to the Micro-Frontend component, these will be proxied to the data adapter. Based on the user token, the adapter provides different data formats for the visualization components.

There are reasons why this project benefits from adopting Micro-Frontends architecture for visualization components. Currently, all the visualizations are only implemented by ReactJS; however, different visualizations are developed independently by different teams in different universities, which means a modification in one visualization should not affect the other visualization. The continuous development and deployment are priorities. Each development team is familiar with different front-end framework and the visualization can be exchanged between teams. Additionally, from the architecture point of view, individual visualization is treated as black boxes that each handles its own data retrieval, processing, and presentation [9]. Their outputs are optimized and controlled by the dashboard composer. There are other architectures or technologies that aims at re-usability, however, each visualization is built as a complete and independent application not just a component. With these requirements, Micro-Frontends paradigm provides features that best fit for the implementation of visualization components, where individual independent visualizations works together to create larger front-end, dashboard.

Communication between micro front-end visualizations

In Micro-Frontends architecture, it is important to define the communication between micro-frontends. One method is to use an event emitter injected into each micro-frontend [2]. When one of the micro-frontends changes its state or emits an event, the others, which have subscription of that particular event, will react appropriately. Another way for micro-frontends to communicate with each other is using message broker, for example MQTT or RabbitMq. Alternatively, the temporary (session storage) or permanent (local storage) web storage can be used to store the information that is shared between micro-frontends. By utilizing the local storage and the same event-driven mechanism, it would release the pressure on implementation and maintenance when new features added. However, there are also drawbacks, for example the life cycle of the react component is hard to handle and the web application should better be client independent, that need the careful consideration before choosing web storage as means of communication.

In the VISDOM project, for teaching case, there are many visualizations, which show the progress of one student or the status of all students registered for a course, that work together in the dashboard. They follow the Micro-Frontend architecture, so the communication or information exchange is important. For example, as mentioned, there are two course implementations, when one of the visualizations changes the selected course implementation, the others, which also have selection of course implementation, should change their view according to the new selected course implementation. By this way, it will implement a synchronization between visualizations, which improve users' experience as they do not need to select the configuration mulitple times.

In order to facilitate communication between visualizations, MQTT broker is utilized in the VISDOM project. DockerHub provides Eclipse Mosquitto image, which is an open source implementation of a server for the MQTT protocol [12]. MQTT broker is used for some reasons. It is designed as an extremely light-weigh publish/subscribe message transport [13]. Additionally, for the communication between Micro-frontends components, MQTT can be simply set up and modified based on future requirements. The relation between micro-frontends and MQTT broker is shown in **Figure 3.5**

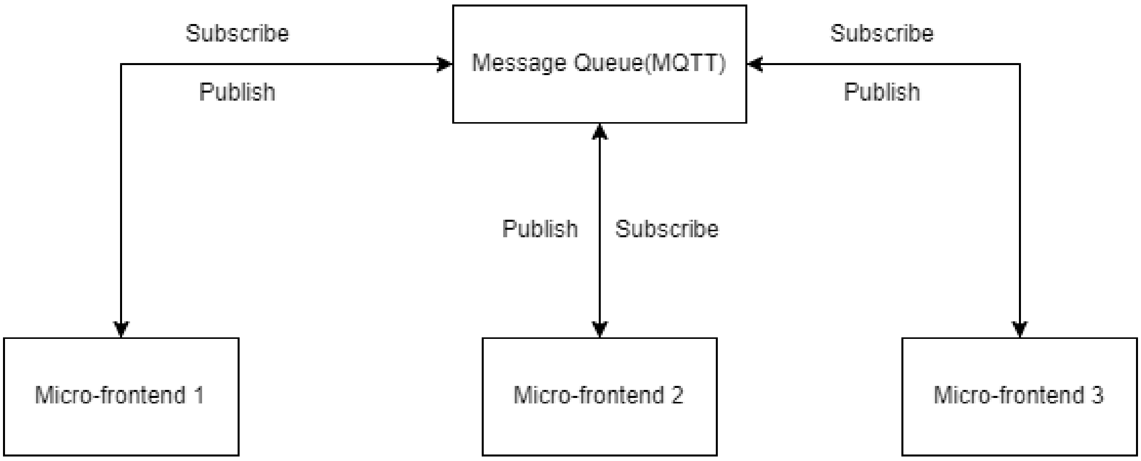


***Figure* 3.5** *Relation between micro-frontends and MQTT broker*

The core of MQTT is the MQTT broker (Message Queue as in **Figure 3.5**) and the MQTT client (micro-frontend application). An MQTT clients publishes a message, which includes a topic, to the broker and other clients subscribing to that topic will receive messages. The MQTT broker is responsible for dispatching messages between the sender and the according subscribers [14]. The broker uses the topics and the subscriber list to dispatch the messages to appropriate clients.

Each micro-frontend acts as both publisher and subscriber. Every visualizations can publish a new state value to one topic, and every other visualizations follow or subscribe to that topic will change their state according to the new state value. For example, in the teaching case, as mentioned above, the visualizations show the status of one student specified by unique ID or the whole course progress with multiple view modes. When one of the visualizations changes its state such as student ID, or viewing mode, or different course implementation, the other visualizations subscribing to that topic will also change their states, which leads to a change in the view according to the new states. By this way, it allows synchronization between micro-frontend applications, which is also one of the key requirements in developing the configurable dashboard.

Data Management System

The data management system collects data from various software engineering tool, then transforms and processes them to provide the suitable uniform data interface for visualization consumption. The **Figure 3.6** shows the Data Management System that is currently developed in the VISDOM project.

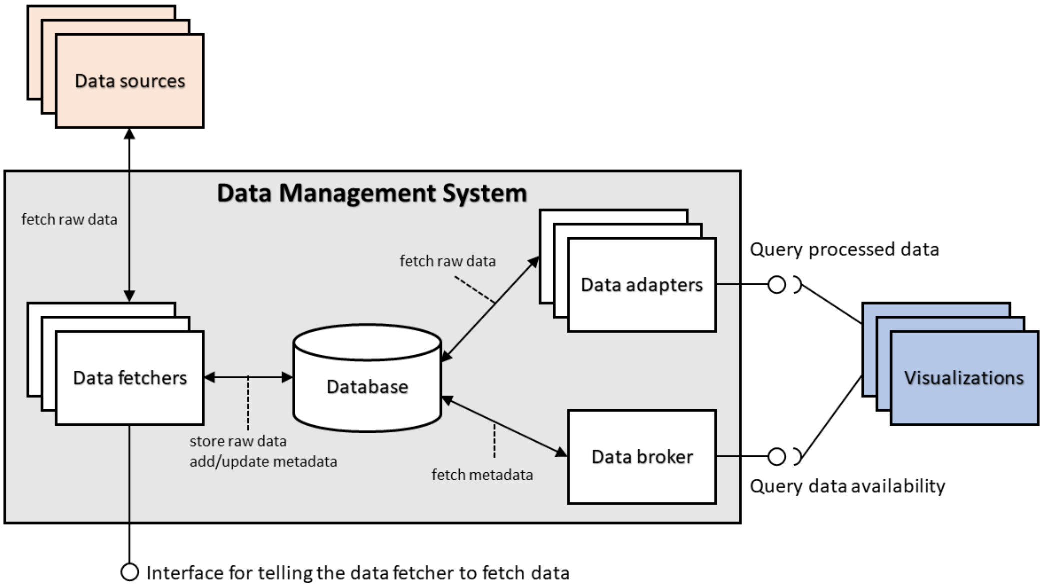


Figure 3.6 *Data management system architecture [15]*

There are four main components in Data Management System: Data fetcher, database (MongoDB), data adapters, and data broker. From **Figure 3.6**, the data fetcher pulls the raw data from the data sources and store the data in database [15]. Then, the data adapter can read and fetch the raw data from database. Based on the visualization components, the data adapter will process and transform the raw data into appropriate format in order to provides them to visualizations.

In this thesis, the database is the Gitlab database, which is pulled directly from the repositories hosted by Tampere University's Gitlab, and A+ database. The database contains various collection, for example commits' information includes the number of commits of each task, time and content of each commit, and points that students gain from single exercise. The adapter will read these data and transform them based on different visualizations' needs. It provides different endpoints for different data query purposes. For instance, the adapter provides an endpoint that single-student-progress visualizations can query only the information related to a specific student through student ID. This shows the original idea of using the adapter that it eases the pressure on processing raw data from the front-end side. Additionally, the query time will also be reduced as the only the necessary parts of data are queried. With the new data adapter architecture, the memory cache simply stores the query parameters and the corresponding response in memory. With memory cache, even if the data sent over the network is reasonably large, or each micro-frontend application queries processed data separately and try to make many requests to the server, it would not burden the server.

As visualizations follow Micro-Frontends design pattern, so there are also drawbacks in the data query aspects. Even though the data adapter provides multiple endpoints for multiple data needs, however the client side still need to process the data multiple times due to the format given by the adapter does not suitable for displaying purpose in many visualizations. Additionally, whenever users input changes, the whole data is queried and processed again, which leads to the noticeable loading time.

3.4.2 Configurable dashboard

The key concept of the dashboard composer is the dashboard designed for different stakeholders, roles, needs, which means that it can be configured based on users input and preferences. The dashboard consists of multiple visualizations in the same page, which constitutes a view satisfying the users' needs. Different roles have different dashboard functionality limitations.**Figure 3.7** below shows the dashboard composer architecture.

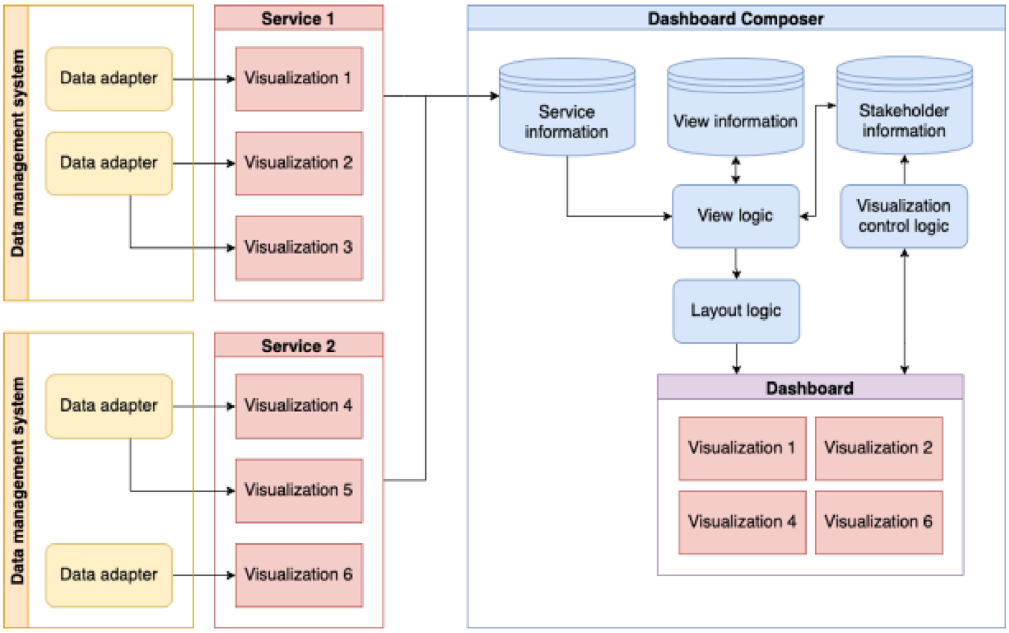
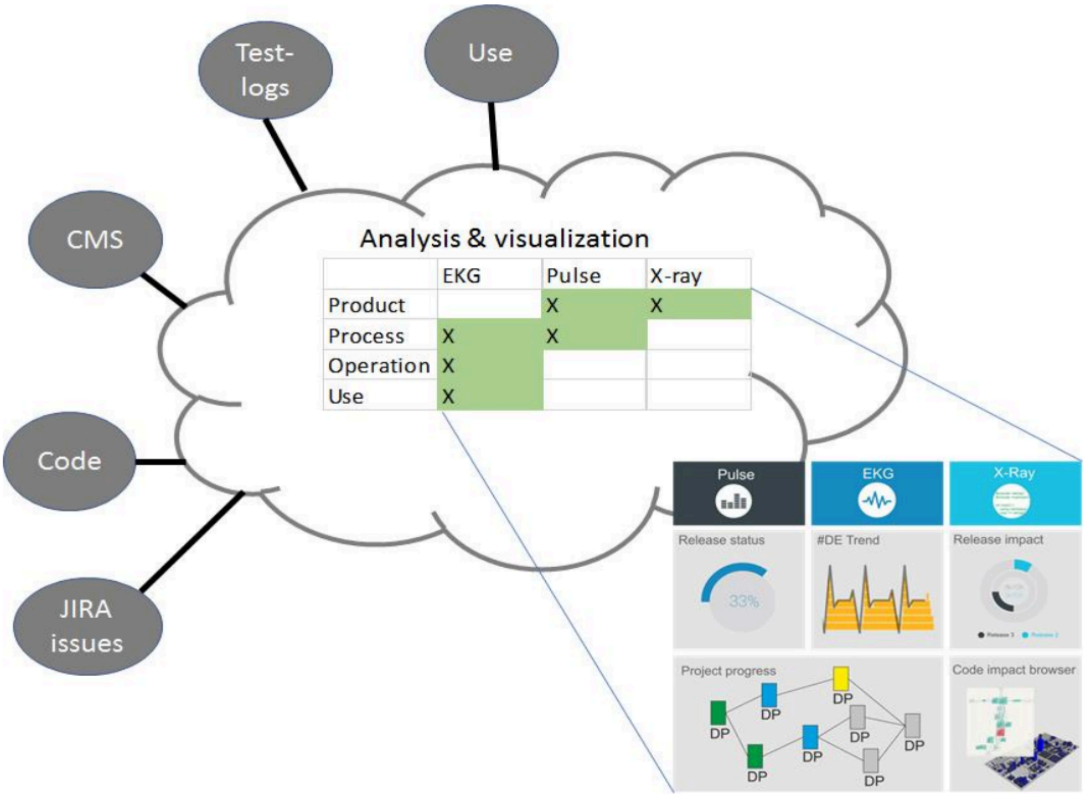


***Figure 3.7** The dashboard composer architecture. [9]*

The yellow box presents for the Data Management System which is discussed in the previous sub-section. The two red boxes in the middle contain visualizations that follow the Micro-Frontents architecture. Individual visualization is implemented as visualizations services by different development teams and other entities interested in specific visualizations concepts or domains [9]. Last but not least, the main topic in this subsection, the dashboard composer specifics are in the right box which is in blue. It provides a necessary infrastructure, which allows the services to register their available visualizations for use within the dashboard composer. These applications are registered as micro-frontends and can be developed and maintained separately by different teams. The dashboard composer makes use of import maps, which allows web pages to control the behavior of JavaScript imports [16], to include the source code of multiple visualizations.

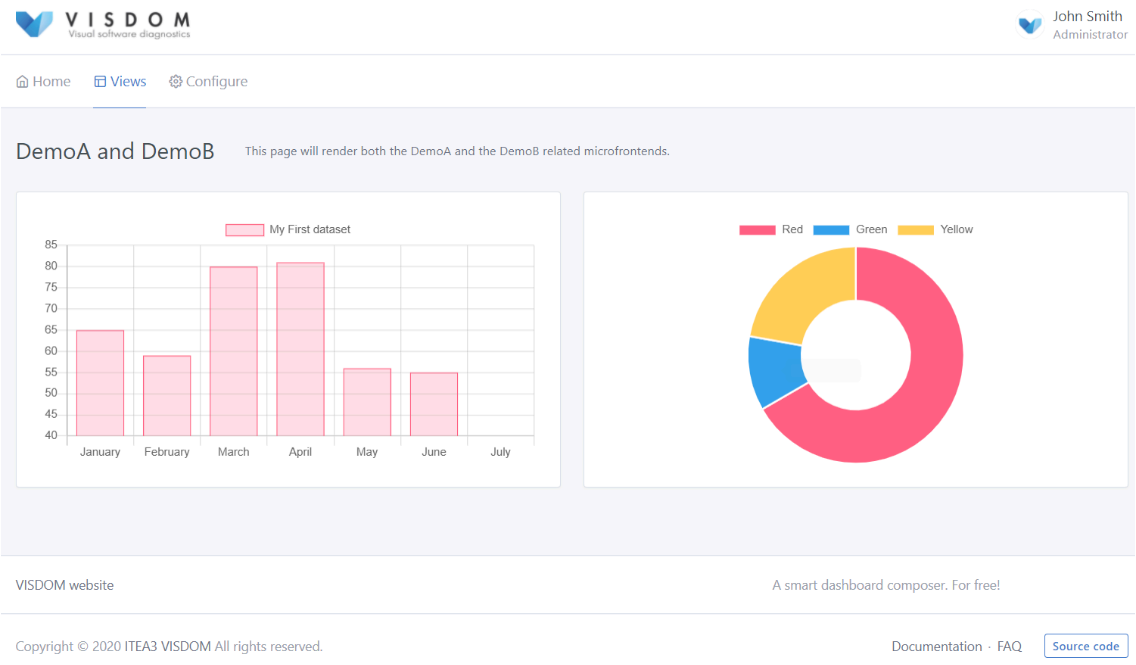
The components of dashboard are depicted in **Figure 3.7**. Service information is responsible for creating available views based on user's roles, preferences; it also includes metadata of what and how the front-end applications should be viewed. View information includes a selection of visualizations, pre-configured perspectives, and layout information based on user's specifies [9], which then can be rendered as individual dashboard page view. Stakeholder information, as its name, is the collections of stakeholder information including set of assumptions and guess of which visualizations should fit and relate to one specific stakeholder. From this, the pre-configured and default dashboard view can be improved. View logic component creates, manages, and renders views configurations. Also, it provides the logic for selecting individual visualizations or group of visualizations perspectives for different user roles [9]. Layout logic provides necessary functionalities for the layout of multiple visualisations in the same page view including responsive and customized grid for displaying a single application. Lastly. visualization control logic implements the necessary functionality for inter-visualization communication and relaying of events, such as changes in visualization perspective, to all the visualizations present in the current view [9].



	EKG	Pulse	X-ray
Product		X	X
Process	X	X	
Operation	X		
Use	X		

Figure 3.8 *VISDOM dashboard allowing creation of custom dashboards from same data in any viewpoint [10]*

Before accessing to the dashboard, the user need to log in to the dashboard with valid credentials. Based on the provided credentials, the composer provide the mechanism for selecting and configuring the visualizations based on current user's role, preferences, and the other factors. By this way, it is possible to retrieve the suitable default dashboard based on the user's role and interests, and stakeholder information. Additionally, the included visualizations can also be customized and set limitations based on personal requirements and roles, which means that not everyone can configure the dashboard and these visualizations completely in the same way. It is possible for the stakeholders to save the configuration of their customized dashboard and visualizations for further use, which needs a database and back-end for; however, in the current version of dashboard composer this feature has not been implemented. The dashboard currently supports for roles: administrator, teacher, teaching assistant, and student. Each role will have different default dashboard and functionalities/features in an individual visualization.



DemoA and DemoB

This page will render both the DemoA and the DemoB related microfrontends.

A smart dashboard composer. For free!

Copyright © 2020 ITEA3 VISDOM All rights reserved.

Figure ***3.9*** *The dashboard composer example view*

Figure 3.9 gives an example of multiple visualizations displaying in a single view page. The position and size of each visualization can be customized based on personal usage and preferences. All these configurations will be saved for further needs, which means that if user log in with the same credentials next time, the user will no longer need to re-configure the dashboard.

In the current version of dashboard composer, it has many uncompleted features, which should be implement to ensure the original concept of dashboard:

not everyone is accessible to all micro-frontends. For example, the configurations and preferences storing part is in view information, which should use the personal token to fetch the configurations and preferences from database. Using role/user restriction that a user can only access to a set of predefined micro-frontends. Currently, every roles can access to all functionalities of all visualizations, which is in contrast with the initial purposes of the dashboard. From the visualizations perspective, there should be ways for them to receive the user's token that they can set restrictions in features based on stakeholders information.

4 PROTOTYPE VISUALIZATION FOR VISDOM PROJECT

From the discussion in the previous chapters, as each Micro-Frontend application is implemented and developed independently and development teams is technology agnostics. Therefore, there are multiple approaches to implement a Micro-Frontend applications. For the VISDOM project, all visualization applications are implemented by ReactJS framework using Single-SPA. This section provides the detailed implementation steps of visualizations, which shows the benefits of Micro-Frontends in implementing the visualizations that are compatible with the dashboard.

The EKG and Status visualizations will be discussed as an example visualization in this section as all the Micro-Frontend visualization applications share the similarities in set up and implementation steps. These visualizations, in this case, use the data from the Programming 2 course implemented by Tampere University for displaying multiple variables related to students' status during attending the course such as points, number of commits and submissions, and so on.

4.1 Setting up single-spa

As discussed in the previous section, single-spa consists of application and single-spa root config, and single-spa shares many similarities with SPA. Therefore, before converting application to single-spa application, ensure that the SPA does not contain any rendering errors or warnings.

Webpack is a free and open-source static module bundler for JavaScript application [17]. The React application requires environmental variables declared in its configuration, which are used to create Webpack config. Webpack config is a JavaScript object [18] allowing configuring and extending Webpack's basic configuration. All the required variables are listed as below:

```
ADAPTER_HOST=
MQTT_HOST=
CONFIGURATION_HOST=
DATA_TOKEN=
VISUALIZATION_KEY=
```

Program 4.1 *Environmental variables for Webpack config*

The name of these variables are quite straightforward that can explains its meanings. `ADAPTER_HOST` is a URL that visualization applications do the data query with valid `DATA_TOKEN` for authentication purposes, while `MQTT_HOST` defines the host connection where micro-frontend applications communicate with each others. The detail of information transfered between visualizations will be introduced later in this section. For EKG visualization, it has a feature of storing users' configuration, which requires a server for this functionality. The URL to that server is defined in `CONFIGURATION_HOST`. Lastly, as the storing configuration server requires a unique key specifying for a visualizations to access to the correct database, the special key is stored in variable `VISUALIZATION_KEY`.

In the traditional React SPA, these environmental variables can be access with `process` which is a global variable is injected by the Node at runtime and represents the state of the system environment of the application. However, as mentioned above, when converting traditional SPA to single-spa, it also creates a shareable, customizable webpack config that adds react-specific configuration to webpack-config-single-spa [8]. The Webpack config files has structure as displayed in Programe 4.2.

```
const webpackMerge = require("webpack-merge");
const singleSpaDefaults = require("webpack-config-single-spa");
const webpack = require("webpack");
require("dotenv").config(); // use dotenv to access environment
   variables

module.exports = (webpackConfigEnv) => {
  const defaultConfig = singleSpaDefaults({
    // The name of the organization this application is written for
    orgName: "visdom",
    // The name of the current project.
    projectName: "ekgview",
    //Environment Variables
    webpackConfigEnv,
  });

  return webpackMerge.smart(defaultConfig, {
    // modify the webpack config however you'd like to by adding to
        this object
    ],
  });
};
```

Program 4.2 *Webpack config for visualization*

The last thing left of setting up single-spa application for React is converting the visualization, whick allows the application to be downloaded as an in-browser ES module with "entry file".

```
// Create our SingleSPA application.
const lifecycles = singleSpaReact({
  React,
  ReactDOM,
  rootComponent: App,
  errorBoundary(err, info, props) {
    // Customize the root error boundary for your microfrontend
       here.
    return <ErrorBoundary />;
  },
});

export const bootstrap = lifecycles.bootstrap;
export const mount = lifecycles.mount;
export const unmount = lifecycles.unmount;
```

Program 4.3 *"entry files" for creat single-spa application*

This entry file not only allows the visualization applications to be in-browsers ES module, but also satisfies the single-page application requirements: it must know how to mount, unmount, and bootstrap.

4.2 Communication setup between Micro-Frontend visualization applications: MQTT

There are multiple message brokers that can be utilized for storing, routing, and delivering messages between visualizations in VISDOM project. However, MQTT broker is chosen thanks to its lightweight publish/subscribe message transport [13], and simple setup and implementation. DockerHub has an image for the Mosquitto broker.

The setup of MQTT message broker can be done directly by executing commands to the docker container or MQTT client can be set-up to the local machine. The second approach is more suitable in this case, it can be done by utilizing a docker-conpose file as in Program 4.4.

```
version: "3.0"
services:
    mqtt:
        image: eclipse-mosquitto
        restart: always
        ports:
            - "1883:1883"
            - "8898:8898"
        volumes:
            - ./tmp/mqtt:/mosquitto/data
            - ./mosquitto.conf:/mosquitto/config/mosquitto.conf
```

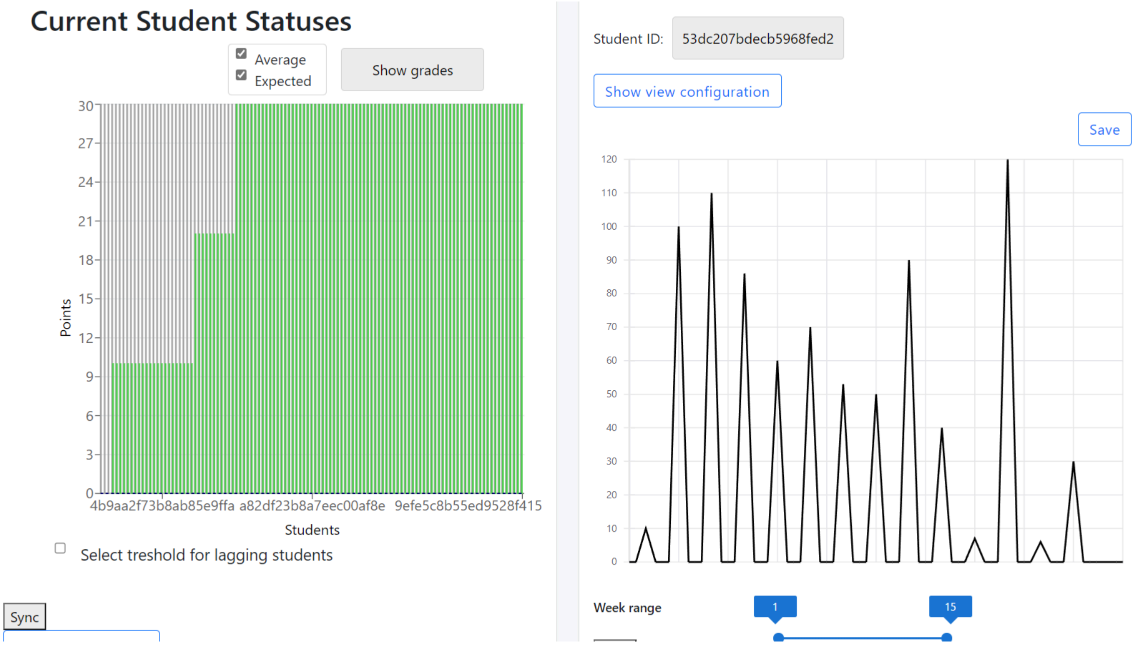
Program 4.4 *Docker-compose for MQTT setup*

`version: "3"` denote that version 3 of Docker Compose is used with appropriate features. `services` defines the container that will be created, in this case the container name is mqtt. As mentioned, mosquitto provided by DockerHub is utilized, which is a pre-built image. `port` defines the mapping between container's host and machine host. In this case, the MQTT broker is exposed to port `ws://127.0.0.1:8898`, which can be assigned to `MQTT_HOST` in environmental variables file above for connection.

```
import MQTT from "async-mqtt";
const MQTT_TOPIC = "VISDOM";

const setupMQTTClient = (client, dispatch) => {
  \\TODO: Define the client setup with available events
  client.subscribe(MQTT_TOPIC);
};

export const MQTTConnect = (dispatch) => {
  return MQTT.connectAsync(MQTT_ADDR)
    .then((client) => {
      setupMQTTClient(client, dispatch);
      return client;
    })
    \\TODO: Disconnect with the client is the connection is fail
};

export const publishMessage = (client, messageObj) => {
  return client.publish(MQTT_TOPIC, JSON.stringify(messageObj));
};
```

Program 4.5 *MQTT connection in visualization components*

When one of the visualization changes its state, it will publish an message with specific MQTT_TOPIC and message is the state object, which is converted into string type. Other visualizations subscribe to that topic will receive the string type message. Then they will compare their current state with the incoming message, if any differences are detected, the visualizations will render the view again based on the shareable state. By this way, it establishes a real-time synchronization between visualizations.



☑ Average
☑ Expected

☐ Select treshold for lagging students

Student ID: 53dc207bdecb5968fed2

Week range: 1 – 15

Figure 4.1 *Status and EKG in the same page view*

For example, in Programming 2 course, a large number of students attending the course, so finding an exact studentID among more than hundred others is difficult. The idea is when status and EKG visualizations are in the same browser page, users do not need to change the student in two views separately, but changing one of visualizations, the other one should update its view asynchronously for better user experience.

Each bar in status view represents one student, and it can be selected for more detailed information in the pop-up dialog. When the user choose one student by clicking to one specific bar in status visualization, it can publish the message by the "sync" button below the graph via MQTT broker. The EKG view subscribing that "VISOM" topic will receive that studentID, then the visualization will automatically update its view corresponding to the incoming message.

4.3 Integrating visualization applications to VISDOM Dashboard Composer

The dashboard composes of two main components: Dashboard Composer, and Root Config. The Root Config is responsible for providing the infrastructure for registering applications to the VIDOM ecosystem [19]. Different applications can be registered as mirco-frontends and can be developed, implemented, and deployed independently by different development teams. Lastly, the dashboard composer is responsible for two main tasks [20]. The first task is selecting the suitable visualizations for the current user based on multiple criterias such as role, preferences, or interests. The second one is to handle the dashboard layout and rendering logic. All the available visualization components are configured as separate micro-frontend application in the composer and rendered according to stakeholders.

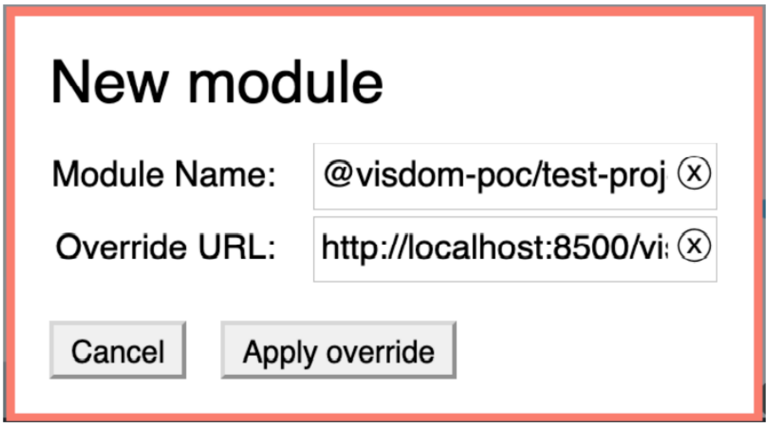
In the current version, the Dashboard Composer only provides configuration functionalities related to selecting visualization displayed in one page, and layout/sizing of each application. This configuration is stored in local storage. Therefore, there are no users' role limitations, every roles can access to the same visualization with similar features provided by each micro-frontend application. This is not the supposed behavior of Dashboard Composer. In future development, the authorization must be the priority, the proposal development will be discuss more carefully in Chapter 5.

4.3.1 The mechanism of deploying visualizations with Dashboard Composer

Once the application is started, the import map override can be set to the port that the application is running. Import-map-override is an browser and NodeJS JavaScript library for overriding import maps [16], which allows dynamically change to the URL for the modules by storing overrides in local storage.

When new visualizations are added to dashboard locally for testing purposes, the module name should be in this format: `@[organization name]/project name`. For example, with EKG visualization, the module name should be `@visdom-poc/ekgview`, where organization name is *visdom-poc*. After the visualizations are added successfully, new view can be added with the form shown in **Figure 4.3**.

From the above form, it is shown that the dashboard can be configured in mul-

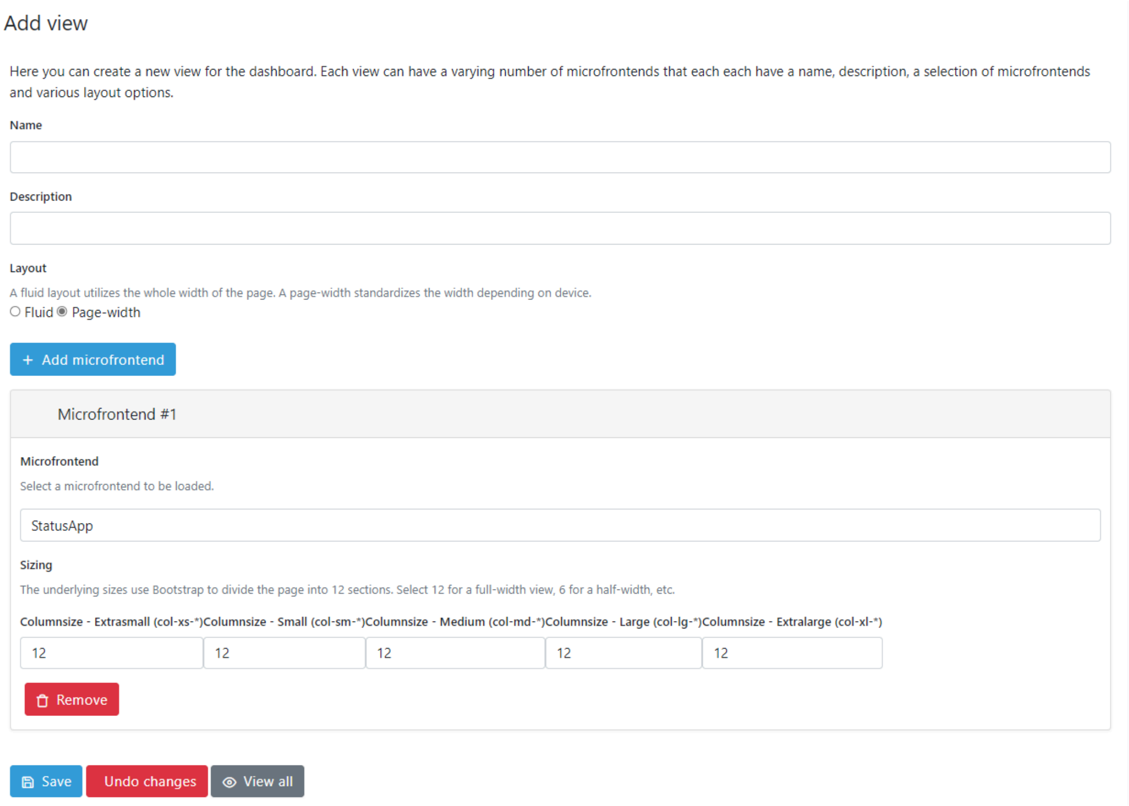


New module

Module Name: @visdom-poc/test-proj

Override URL: http://localhost:8500/vi:

***Figure 4.2** New module dialog in local development*



Add view

Here you can create a new view for the dashboard. Each view can have a varying number of microfrontends that each each have a name, description, a selection of microfrontends and various layout options.

Name

Description

Layout

A fluid layout utilizes the whole width of the page. A page-width standardizes the width depending on device.

○ Fluid ● Page-width

Microfrontend #1

Microfrontend

Select a microfrontend to be loaded.

StatusApp

Sizing

The underlying sizes use Bootstrap to divide the page into 12 sections. Select 12 for a full-width view, 6 for a half-width, etc.

Columnsize - Extrasmall (col-xs-*)	Columnsize - Small (col-sm-*)	Columnsize - Medium (col-md-*)	Columnsize - Large (col-lg-*)	Columnsize - Extralarge (col-xl-*)
12	12	12	12	12

***Figure 4.3** Add new view to dashboard*

tiple ways that satisfy users' needs, and interests. Users are able to choose how many visualizations that would be displayed in a single browser page by "add microfrontend" button. Additionally, users can choose the size for each visualization, in **Figure 4.3**, the size of micro-frontend application would be full width of browser page. It can be set to half by reducing all the sizing options two times.

4.3.2 Visualizations deployed with Dashboard

The Root Config component of Dashboard Composer provide `importmap.json` file, which allows micro-frontend applications to be added as default module instead of local development.

```
\\Below is just some example necessary modules
{
  "imports": {
    "react": "https://cdn.jsdelivr.net/npm/react@16.13.1/umd/react.
       production.min.js"
    "@visdom-poc/root-config": "/visdom-poc-root-config.js",
    "@visdom-poc/composer": "/visdom-poc-composer.js",
    \\Adding EKG view for deployment
    "@visdom-poc/ekgview": "/ekgview.js",
  }
}
```

Program 4.6 *importmap.json file for default modules*

5 SITUATION ANALYSIS

Within the thesis scope, it is clear that the VISDOM project has many benefits from adopting Miro-Frontends architecture. It also explain the reasons why Micro-frontends pattern has just been introduced recently, but has gained a great popularity among other architecture. This section will give a detailed discussion about the current design with original reference architecture and the concept of micro-services in the back-end side based on literature review.

5.1 Visualization implementation and reference architecture

In the current implementation, as discussed above, the Dashboard Composer allows users to configure the rendering layout of different visualizations in multiple ways. Based on their needs, interests, and preferences, users can select the number of visualization applications that can be shown in the same browser page. Additionally, the view applications can be configured in such ways that they can be displayed in the same row (side-by-side) or as a list. From the visualization point of view, for genetic and advanced visualizations such as status and EKG, they provide various functionalities that allow user to visualize data in multiple ways for further analysis and problems detection via repeated patterns.

With these features, both Dashboard Composer and visualization can satisfy all stakeholders' needs, however, despite Dashboard provides a login page for authentication, the credentials are hard-coded, which means the users' email and password are stored as pure string. The Dashboard Composer simply receives the user's credentials inputs and compares them with login information stored as string in the back-end side without any further authentication layout. Therefore, there is no restriction between users' roles in both Dashboard Composer and visualizations as they do not receive any authentication tokens from the Dashboard. For example, for teaching case, there are at least two roles, teacher who is responsible for the course implementation, and teacher assistants (TA) who take responsibilities for exercises graded and student support. Teacher has interests in course overview, while TA would like to have a more detail information of one student with his or her workload. Dashboard should capture these tendencies to come up with the best visualization application what can fulfill these needs. Furthermore, for visualizations, they should receive the users' credentials to limit their available features for different stakeholders.

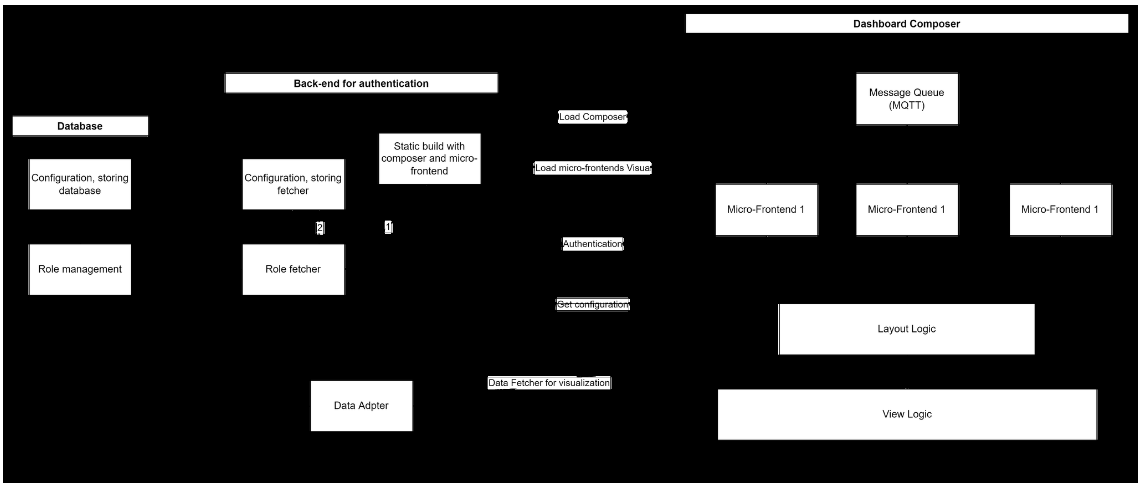


Figure 5.1 *Proposal architecture for getting configuration with authentication*

Firstly, the back-end loads the Dashboard and mirco-frontend application with features disability and without any data. After users login to the Dashboard with credentials, the server will receive these credentials and check if they are valid. If the users successfully login to the Dashboard, with role restrictions, Dashboard only provides available and pre-defineds visualizations. The server then will send the valid token to micro-frontend visualization applications. They will restrict the features and data that can be accessed by the current user to ensure that not everyone is accessible to the same micro-frontend application. For some visualizations, they allow users to save their configuration for further inspection, for example in EKG visualization, users can save their current configuration for later use without remembering and re-configuring the graph again for next time. The server will check the credentials sent by dashboard to fetch only the configuration/preferences available for specific users. By this way, the role of users will affect the functionalities and feature provided by Dashboard Composer and micro-frontend applications, which is also the key concept of the Dashboard.

5.2 Current prototype visualizations and microservices concept

From **Figure 2.1** and **Figure 2.2**, microservices and Micro-Frontend architecture share many similarities from the concept to implementation approach. These two architectures have the same purposes: decomposition of one large monolith application into smaller single-responsible-task applications that can be implemented, tested, and deployed independently.

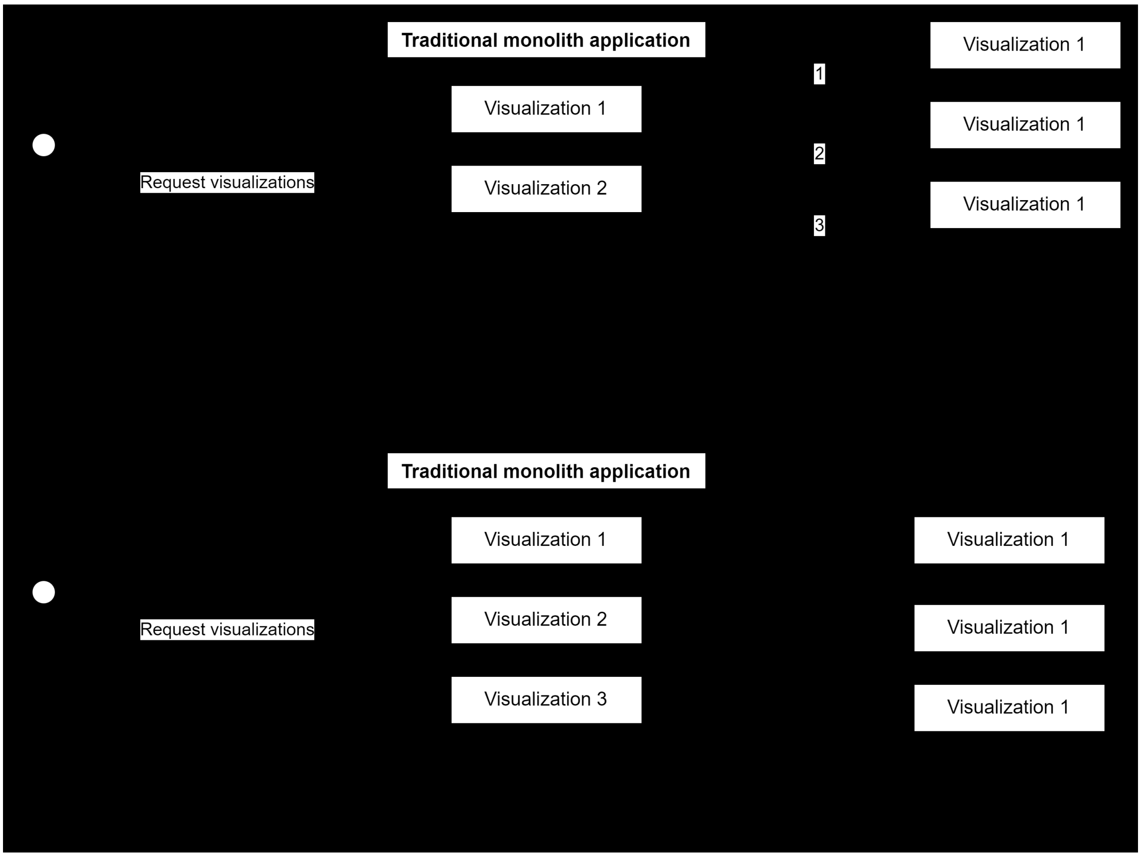


Figure 5.2 *a) Visualizations with traditional monolith application b) Visualizations with Micro-Frontends architecture*

Similar to microservices architecture, when the users successfully access to the Dashboard Composer and request for multiple visualizations displayed in the same browser page, with traditional web application implementation, one application is responsible for all steps from setting the layout of views to fetch necessary data for every single visualizations. Each view is loaded in order, therefore, it takes time to complete the loading of the whole page. The more visualizations users prefer to inspect, the more time it takes. However, with Micro-Frontend architecture, when the Dashboard receives users' request for visualization, it will call on visualization application to compose the request.In each micro-frontend application, the data is fetched and processed independently at the same time, which reduce the loading time significantly. For instance, in the VISDOM project, there are two dashboards to serves different visualization purpose at different project phase. One is the Dashboard Composer as discussed through this thesis for cooperating and sharing visualizations between universities, and the other one is teaching demo dashboard for visualizations experience in Programming 2 course. The former one follows Micro-Frontend architecture, and the latter one is a traditional monolith application. When inspecting the page containing all visualizations, the teaching demo dashboard requires much more time than the Dashboard Composer.

6 CONCLUSION

While microservices has been introduced and used widely, Micro-Frontends architecture is still a novel concept but has gain a huge popularity thanks to its innovative implementation approach for the front-end side. The most common motivation for adopting Micro-Frontend architecture amongst the others is the the growth of traditional monolith applications leads to complexity and difficulty in scaling and maintaining. Micro-Frontends extends the concept of microservices for server side to client side. However, despite having multiple advantages in many development aspects, Micro-Frontends architecture is not a silver bullet for designing web applications. For example, it may increase overall complexity of the whole system, or in general increase the development and cloud-related costs [2] for the small software teams or the traditional monolith application can fulfills all the web application requirements.

In this thesis research studies, Micro-Frontends architecture is adopted by VISDOM project development teams for sharing visualization between different teams and independent implementation, test, and deployment. Adopting such a novel concept helps solving many problems and developing requirements that a traditional monolith application such as teaching demo dashboard cannot fully provide the best performance. Having many separated visualization components that each of them takes responsibility for coming up with different visualized data approach benefits the project multiple ways as discussed in the previous chapters. Currently VISDOM project has seven visualizations and the number may increase in the near future. Therefore, by implementing each view as a micro-frontend application reduces the risks and challenges for integrating new visualizations application to the existing ones.

Not only VISDOM, the other well-known companies such as Netflix, Facebook also acquire the Micro-Frontend concept for web application development, which is a strong evidence for its advantages. However, it definitely has drawbacks when adopting this advance architecture. In VISDOM project, the simple implementation is traded up for scalability and maintenance from using Micro-Frontends pattern instead of traditional approach. Additionally, the pressure on the data adapter also increase as multiple request at the same time. No architecture is perfect, that why looking for an effective innovative way for software development becomes a crucial and major task.

REFERENCES

[1] "Technology radar: Micro frontends." (2006), [Online]. Available: https://www.thoughtworks.com/radar/techniques/micro-frontends. (Accessed: 20.01.2022).

[2] S. Peltonen, L. Mezzalira, and D. Taibi, "Motivations, benefits, and issues for adopting micro-frontends: A multivocal literature review," *Information and Software Technology*, 2021. [Online]. Available: https://www.journals.elsevier.com/information-and-software-technology.

[3] "What is microservices architecture?" (), [Online]. Available: https://cloud.google.com/learn/what-is-microservices-architecture. (Accessed: 26.01.2022).

[4] "Getting started with single-spa." (), [Online]. Available: https://single-spa.js.org/docs/getting-started-overview. (Accessed: 21.01.2022).

[5] M. Geers. "Miro-frontends." (), [Online]. Available: https://micro-frontends.org/. (Accessed: 20.01.2022).

[6] C. Jackson. "Miro-frontends." (), [Online]. Available: https://martinfowler.com/articles/micro-frontends.html. (Accessed: 10.01.2022).

[7] "Bit workflows." (), [Online]. Available: https://legacy-docs.bit.dev/docs/workflows. (Accessed: 23.04.2022).

[8] "Create-single-spa." (), [Online]. Available: https://single-spa.js.org/docs/create-single-spa/. (Accessed: 22.01.2022).

[9] K. Systa, O. Sievi-Korte, H. Bomstrom, and V. Lunnikivi, *Visdom reference architecture, deliverable 2.5.1*, 2020.

[10] V. members, *Fpp annex template.*

[11] "Understanding the value of reference architectures." (), [Online]. Available: https://doveltech.com/innovation/understanding-the-value-of-reference-architectures/. (Accessed: 20.01.2022).

[12] "Eclipse-mosquitto." (), [Online]. Available: https://hub.docker.com/_/eclipse-mosquitto/. (Accessed: 20.01.2022).

[13] "Mqtt: The standard for iot messaging." (), [Online]. Available: https://mqtt.org/. (Accessed: 20.01.2022).

[14] "Getting started with mqtt." (), [Online]. Available: https://www.hivemq.com/blog/how-to-get-started-with-mqtt/. (Accessed: 20.01.2022).

[15] "Components for the data management system." (), [Online]. Available: https://github.com/visdom-project/VISDOM-data-management-system/tree/master/documentation/adapters/general. (Accessed: 18.03.2022).

[16] "Import maps." (2021), [Online]. Available: https://wicg.github.io/import-maps/. (Accessed: 20.01.2022).

[17] "Webpack." (), [Online]. Available: https://webpack.js.org/guides/getting-started/. (Accessed: 23.04.2022).

[18] "An introduction to webpack configs." (2020), [Online]. Available: https://masteringjs.io/tutorials/webpack/config. (Accessed: 21.01.2022).

[19] "Visdom poc root config." (), [Online]. Available: https://github.com/visdom-project/VISDOM-PoC-Root-Config. (Accessed: 22.01.2022).

[20] "Visdom poc dashboard composer." (), [Online]. Available: https://github.com/visdom-project/VISDOM-PoC-Composer. (Accessed: 22.01.2022).