



Development of a feedback service

Jesse Kämäräinen

Bachelor's thesis

December 2021

Information and Communication Technology

Bachelor's Degree Programme in Information and Communication Technology

Kämäräinen, Jesse

Development of a feedback service

Jyväskylä: JAMK University of Applied Sciences, December 2021, 48 pages

Engineering and technology. Degree Programme in Information and Communication Technology. Bachelor's thesis.

Permission for web publication: Yes

Language of publication: English

Abstract

Knowledge-based management has long been a vital part of the marketing strategies of companies. It essentially means decision-making based on facts as in collected data. Modern websites, especially enterprise sites are prepared for this procedure, meaning they usually contain internal analytics tools. It is common to collect data on purchase counts of products to potentially adapt the selection and prices based on the data. Customer satisfactory is also important, and tools that allow rating are a good fit to collect this kind of data. Ratings can be either number- or comment-based, however, number-based ratings tend to yield better results from data analyses.

The task was to develop a widget suited for customer satisfactory data collection, that would be easy for clients to set up on their websites with minimal technical know-how. The widget works similarly to e.g., a chat window that pops up in the bottom of the page. A website visitor can select a rating number, and the data will be collected to an external results website. The clients are also able to create their own, website-specific widget. The goal was to build a full-fledged service, where a client can manage their widgets and look at reports on the results site to support business decision making.

The product was implemented using modern solutions of web- and mobile development, such as React.js for frontend and Google Firebase for backend.

Due to very limited time and the lack of specific materials the project was unfortunately left partly unfinished, therefore some chapters have been written in a more hypothetical manner. Some of the solutions are partly implemented or completed, but not to the point where they could be used in production or even presented. The project will be finished at a later date, as it is a paid customer project, however, since the thesis was much more urgent, the project couldn't be finished within the thesis' time frame. It is still fortunate that a lot of material could be amassed even from the unfinished product.

Keywords/tags (subjects)

Business intelligence, web development, full stack development, full stack, Firebase, React, data analysis, data collection, user feedback

Miscellaneous (Confidential information)

Kämäräinen, Jesse

Asiakaspalautteen keräyspalvelun kehitys

Jyväskylä: Jyväskylän ammattikorkeakoulu. Joulukuu 2021, 48 sivua.

Tietojenkäsittely ja tietoliikenne. Tieto- ja viestintäteknikan tutkinto-ohjelma. Opinnäytetyö AMK

Julkaisun kieli: englanti

Verkkojulkaisulupa myönnetty: kyllä

Tiivistelmä

Tiedolla johtaminen on kauan ollut tärkeä osa yritysten markkinointistrategiaa. Tiedolla johtamisella tarkoitetaan päätöksentekoa perustuen faktoihin – toisin sanoen kerättyyn dataan. Nykyajan web-sivustot, varsinkin yritysten kohdalla, ovatkin usein valmisteltu tähän tarkoitukseen eli ne sisältävät sisäisiä analytiikkatyökaluja. Yleistä on kerätä mm. dataa siitä, mitä tuotteita asiakkaat eniten tai vähiten ostavat ja potentiaalisesti muokata verkkokaupan valikoimaa ja hintoja datan perusteella. Myös asiakastyytyväisyys on tärkeää ja sen tiedon keruuseen soveltuvatkin hyvin arvostelut. Arvostelut voivat olla ns. numero- tai kommenttimuodossa, mutta parempaa data-analytiikkaa voidaan suorittaa numeroarvostelujen pohjalta.

Tehtävänä oli kehittää asiakastyytyväisyysdatan keräämiseen soveltuva ”vimpain” (eng. widget), joka asiakkaiden olisi helppo sijoittaa omille sivustoilleen minimaalisella teknisellä tietämyksellä. Työkalu toimii samaan tapaan kuin esim. usealla sivustolla käytetty alareunaan pomppaava chat-ikkuna. Widgetissä on mahdollisuus valita arvostelunumero ja data tullaan keräämään erilliselle tulossivustolle. Asiakkaan täytyy myös pystyä luomaan oma sivustokohtainen widgetinsä. Tavoitteena oli saada valmiiksi kokonainen palvelu, jossa asiakkaan on mahdollista hallita omia widgetejään kokonaisvaltaisesti, sekä tarkastella raportteja tulossivustolla päätöksenteon tueksi.

Toteutuksessa hyödynnettiin moderneja web- ja mobiilikehityksen ratkaisuja, kuten etupäässä React.js -kirjastoa, lukuisia npm-kirjastoja ja -paketteja sekä palvelinpäässä Googlen Firebasea. Kyseessä on siis niin kutsuttu full stack -projekti.

Projekti jäi ajan ja tietyn materiaalin puutteesta johtuen osittain kesken ja monessa luvussa onkin selitetty ratkaisut enemmän hypoteettisesti ajatellen, joskin joitain kyseisistä ratkaisuista on jo ehditty työstää, tosin ei aivan käyttökelpoiseksi asti. Projekti tullaan saattamaan loppuun myöhemmällä ajalla, onhan kyseessä kuitenkin maksettu asiakasprojekti. Opinnäytetyötä varten projektia ei saatu ajoissa valmistumaan, sillä opinnäytetyöllä oli tässä tapauksessa tiukempi aikaraja. Plussaa on kuitenkin se, että aiheesta sai kokoon hyvin materiaalia projektin kesken jäämisestä huolimatta.

Avainsanat (asiasanat)

Tiedolla johtaminen, web-kehitys, full stack -kehitys, full stack, Firebase, React, datan keräys, datan analysointi, käyttäjäpalaute

Muut tiedot (salassa pidettävät liitteet)

Sisältö

1	Introduction.....	4
1.1	Background	4
1.2	End goal.....	4
1.3	Definition.....	5
1.4	Research.....	5
1.4.1	Research questions	5
1.4.2	Research method	6
2	Project overview.....	6
2.1	Tools and technologies.....	6
2.1.1	Npm and the most important programming principle	6
2.1.2	React	7
2.1.3	Object-oriented programming & JSDocs	8
2.1.4	Google Cloud Platform	10
2.2	Project hierarchy	10
2.2.1	The monorepo.....	10
2.2.2	Yarn workspaces	11
2.2.3	The management application, or “the editor”	13
2.2.4	The analytics site, or “results site”	14
2.2.5	The widget	14
3	Backend.....	15
3.1	Using serverless computing.....	15
3.2	Selecting the right cloud provider	15
3.3	Building the backend	16
3.3.1	Environments and Firebase.....	17
3.3.2	Cloud Functions.....	19
3.3.3	Firestore versus Realtime Database	21
3.3.4	Database design	22
4	The widget.....	25
4.1	Maximizing user-friendliness.....	25
4.2	Implementation.....	27
4.2.1	Getting started with the code	27
4.2.2	Build and deployment	29
4.3	Integration to the project.....	30
4.3.1	Preventing misuse, a.k.a. handling security	30

4.3.2	Database connections	32
5	Management system	33
5.1	Integration to the existing system	33
5.1.1	Access levels and security	33
5.1.2	Preventing loss of the client's existing data	35
5.2	Widget editor	36
5.3	Use cases	37
5.3.1	Use case 1, the admin	37
5.3.2	Use case 2, the organization manager	38
5.3.3	Use case 3, the organization member	39
6	Results website.....	39
6.1	The purpose	39
6.2	Implementation.....	39
6.2.1	Handling potentially big data	40
6.2.2	User-friendliness: a bit on UI/UX design	41
7	Conclusions.....	45
7.1	Time efficiency	45
7.2	Results	45
7.3	Final thoughts.....	46
	References	47

Figures

Figure 1.	Job market for Angular vs Vue vs React	7
Figure 2.	The "Widget" class	9
Figure 3.	Repository folder structure	11
Figure 4.	Npm-yarn popularity comparison	12
Figure 5.	Workspaces in package.json.....	13
Figure 6.	A simple Firebase "onCall" Cloud Function	20
Figure 7.	Firestore database model.....	24
Figure 8.	Synchronous and asynchronous operations.....	26
Figure 9.	Example of an asynchronous and a synchronous function.....	26
Figure 10.	embeddable-react-widget's starting point in closed and opened state	28
Figure 11.	An example widget.....	29
Figure 12.	Widget script file inside its storage bucket.....	30
Figure 13.	How the widget communicates with the backend	32

Figure 14. The organization view	34
Figure 15. A simple form for creating widgets	36
Figure 16. Creation of a widget with the drag-and-drop editor	37
Figure 17. The organizations view.....	38
Figure 18. Widgets view with navigation	42
Figure 19. Example of information chunks.....	43
Figure 20. Different common "Share" icons.....	44

Tables

Table 1. Cloud service overall comparison	16
Table 2. Development environments	18
Table 3. Firebase database feature comparison	22
Table 4. Management application security levels	35

1 Introduction

1.1 Background

This thesis focuses on the build process of a service, or product, that is used to collect customer satisfactory data for companies in different industries. Said data could then be used for knowledge-based management on the respective company's side, but also for decision-making on their customer's side.

The subject of the thesis is a client project of Bromeco Oy. Bromeco is a small, young software development company based in Kuopio, Finland. The company was founded in 2017, and currently has 3 employees. It was formed by brothers Ilari and Lauri Methuen, from who the name also comes from (Brothers Methuen Company). The company specializes in web and mobile development along with consultation services.

As a software development company, Bromeco mostly deals with client projects. The project of the topic comes from Apec Yrityspalvelut Oy (Apec), which is the client. Apec specializes in the visual aspects of marketing, which could be for example content production, press products, social media marketing, websites, and graphics design. The subject of this thesis is very closely related to another project from the same company, though this document will not be going through said project in any form, but rather focus on a new service from the same company.

1.2 End goal

The end goal of the project was to have a fully working feedback collection service, intended for Apec's clients. Another sub-goal was to finish within a specific time frame and budget. Unlike in school projects, which could in theory be worked on "24/7" up until the deadline, in working life, higher efficiency is required – this essentially means tracking working hours and staying within the client's budget. After all, a lot of employers pay the employees but also bill their customers *per hour*.

The service consists of 3 parts: a management application, or "editor", which is used for creating widgets. The widget itself is the second part. Widgets are small "blocks", that usually appear along

the edge of a website, and are not too obstructive – similar to those chat windows that can be seen on many websites, where a bot asks the visitor “how they can be of service”. The last piece is the results website, which will function to present the collected data in an easily digestible form. The results website will work a bit like Trustpilot, which’ primary use is for customers to gain insight on companies to make buying decisions, but also for the respective companies to have data for business intelligence and comparisons between other companies of the same industry.

The aim is to provide the companies using the service an advantage in the form of knowledge-based management. Ylisiurunen (2021) has summarized that knowledge-based management is based on facts and real data – not assumptions, luck or guesses, and the data should heavily support decision making. The companies would see reliable data not only on the customer satisfaction of their own service, but their competitors as well. Depending on the marketing strategies, if used correctly, this can give said companies an edge.

1.3 Definition

The purpose of this thesis is to dive into the technical aspects of full-stack development: the “how’s” and “why’s” of different tools, and the reasoning behind using them. Another aspect the document will be looking at is general project work in working life. Any sensitive or disclosed information related to the client or numbers will be excluded. Important methods of development will rather be explained in a more general way than snippets of code. Screenshots of the completed applications are shown, but potentially sensitive information will be blurred out.

1.4 Research

1.4.1 Research questions

The main research questions for the thesis are “which technology or tool should be used for task X, and why”, and “how to best meet the client’s requirements”. Working life for programmers is all about efficiency: sometimes a solution must be brought up in very little time. This requires common knowledge of viable technologies, and a developer may find themselves recalling certain same packages, libraries or tools time and time again.

1.4.2 Research method

The thesis mostly focuses on developing a product for a client. The most important sources of information are electronic sources, as in documentations for different tools and services, right next to client and employer feedback. Also, other electronic sources, such as blog posts from industry experts will be referred to.

2 Project overview

2.1 Tools and technologies

The following chapters will go over the major tools, libraries and services that are utilized in the project. The document will not be going through every single *npm package*, as more details will follow in the implementation chapters of each part of the project.

2.1.1 Npm and the most important programming principle

Many programmers would most likely agree, that in programming (or coding), *reusability* is what separates good code from bad. This means for example that programmers should avoid *hard-coding* (code where e.g., a user ID is coded to the project files) and write their functions in a way that isn't too specific to something whenever possible. This enables the use of those functions in as many places as possible, eliminating the need to copy the same code to multiple places. Although in some rarer cases it might be hard to avoid hard-coding, it should usually be avoidable with smart planning and thinking.

Perhaps the biggest reason why copied code is bad is the lack of maintainability. When almost the same function, or almost the same code resides in multiple places in a project, then each time it needs to be changed, the developer has to remember that the same function exists in other files as well. This could lead to critical bugs and confusion among other developers working on the same project.

Many programmers might have heard the phrase “don't reinvent the wheel” – in coding it translates to “someone has already solved the problem for you, so no need to solve it again”. This is where libraries and packages come in. Npm, also known as Node Package Manager, is a package

registry currently containing over a million packages made by coders, for coders. Some of the packages only solve very minor issues, therefore it is up to the developers to decide whether they want to solve the issue themselves, or by using a package made by someone else. What should be kept in mind is that sometimes it can be more efficient to write proprietary solutions for minor issues, rather than go through other peoples' code and documentations, which can sometimes be virtually non-existent. In the opposite end are full-fledged libraries, that have been maintained for years by multiple developers, are well-documented and solve big issues and needs in all kinds of projects. Using these kinds of packages can, and usually will, save hundreds of hours of work. One major consideration is the package's weekly downloads: if the number is really low, it can be concluded that it has a *very* niche use and it is most likely better to write a proprietary solution.

2.1.2 React

React.js is a widely acknowledged modern JavaScript library used to build websites (web applications) and user interfaces that is being developed by Facebook. It was initially released in 2013 and thus has had time to evolve. React has a strong community of maintainers consisting of both individual and Facebook's own developers, along with developers from other companies. This leads to it having great support due to a lot of developers using it. React's key benefit is indeed reusability, hence why it's as popular as it is. Other similar, known technologies of the same kind used in modern web development include *Vue* and *Angular*. Out of the three, React and Angular are the most popular, which leads to most employers looking for a person skilled in either of them. Figure 1 shows job market demand for each technology in the US.

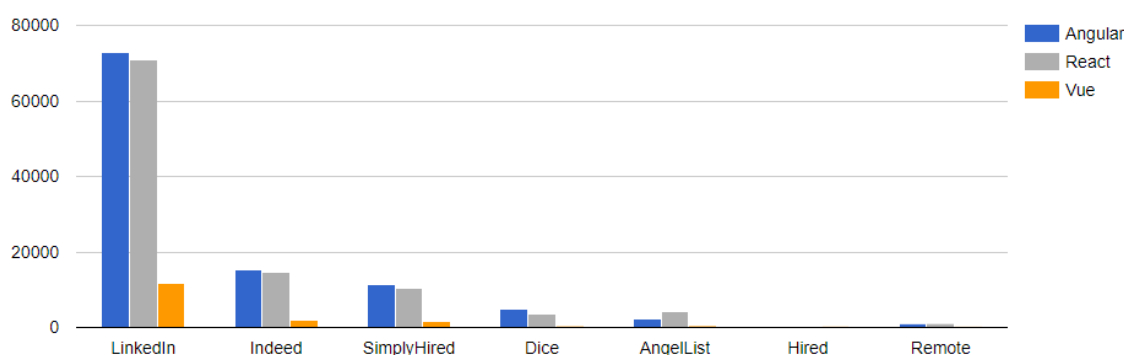


Figure 1. Job market for Angular vs Vue vs React

There are several reasons why React.js is used on this project. First and foremost, it is standardized at Bromeco, and used for most projects, both mobile and web. With React, clean and concise code can be written with minimal copying, using JavaScript and JSX. React offers a very logical way of building applications in the form of *components*. Components can be thought of as the building blocks of an interface; a single button can be a component, but can also be a part of a bigger, more complex component. Modern React components are basically just functions. They may take in parameters, just like a normal function, and transform the data just like a normal function would. The main difference is that React components always return *JSX*. JSX is a syntax extension to JavaScript. It allows writing “HTML” inside JavaScript, which then renders just as a normal HTML document would.

2.1.3 Object-oriented programming & JSDocs

Gillis & Lewis (2021) have summarized object-oriented programming (OOP) as a “computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior”. While JavaScript technically isn’t an object-oriented programming language, like for example C#, it is possible to adopt this programming style into JS as well. After all, JS also allows the creation of classes, static methods etc. The difference in a JS and a C# class is that a JS class isn’t as strongly typed as a C# class and is more easily (or accidentally) broken into a plain object. The OOP style is even more apparent in TypeScript, an “extension” of JavaScript, where typing is much stronger than in JS, and programmers with a background in languages such as C# may become familiar with it rather quickly.

Typing essentially means that objects have custom types, where every object must follow the ruleset of the given class. Take for example a class named “Person”: it would be obvious for every person to have a property called “name”. A person could also have internal methods called “Walk” or “Talk”. Properties that clearly belong to the class should also be kept inside the class, instead of being outsourced. JavaScript allows dynamic property assignment even to class constructs, which can be both a blessing and a curse (though usually the latter). This is the main reason why JS can’t be called an object-oriented language.

The OOP style helps keep object models together and enables writing internal methods, removing the need to write external static methods for every occasion. In Figure 2 is an example of a class that is used in the project (Widget). The class has merely a constructor and a static method that allows instantiating a Widget object using json data – this kind of static method is implemented in each class that is used in the project.

Above the constructor and the method are JSDoc comments. JSDoc is a tool to help developers find out at a quick glance, what properties and method a class or a “typed” object contains. It is a good practice to use these comments, as it helps especially those developers who aren’t familiar with the project.

```
export default class Widget {  
  
  /**  
   * @param {string} id Firestore document ID  
   * @param {string} name Widget name (not visible to website users)  
   * @param {string} organization ID of the owner organization  
   * @param {number} createdAt Timestamp when the widget was created  
   * @param {string} industry Industry category for analytic purposes  
   * @param {'values' | 'rating'} type Question type of the widget  
   * @param {string} question The main question, e.g. "How satisfied are you with our service?"  
   * @param {WidgetQuestionType} [ratingStyle] Defines what kind of buttons will be shown (star, smiley, etc.), if type is "rating"  
   * @param {string[]} [values] The widget's values, if type is "values"  
   */  
  constructor(id, name, organization, createdAt, industry, type, question, ratingStyle, values) {  
    this.id = id  
    this.name = name  
    this.organization = organization  
    this.createdAt = createdAt  
    this.industry = industry  
    this.type = type  
    this.question = question  
    this.ratingStyle = ratingStyle  
    this.values = values || []  
  }  
  
  /**  
   * @param {*} json  
   * @returns {Widget}  
   */  
  static fromJson(json) {  
    const { id, name, organization, createdAt, industry, type, question, ratingStyle, values } = json  
    return new Widget(id, name, organization, createdAt, industry, type, question, ratingStyle, values)  
  }  
}
```

Figure 2. The "Widget" class

JSDoc also acts as somewhat of a type-defining element, and though it does not *enforce* typing in any way, the developer is able to see what properties the class object is limited to, so they don't have to guess and accidentally insert wrong kind of properties. Using JSDoc, it is also easy to "print" out a full documentation for the project.

2.1.4 Google Cloud Platform

Google's Cloud Platform (GCP) was chosen as the serverless service provider. GCP is a cloud platform like Amazon Web Services (AWS) and Microsoft Azure. Out of the three, which are considered the biggest ones, GCP appears the most user- and especially beginner-friendly. This is due to clear user interfaces, generous free tiers for services like Firebase, and generally easy to understand processes. Also, Google's documentations for their services are usually very clear and concise.

Obviously, like every service, GCP has its flaws, but is easily sufficient for a project of this scale. Authentication, Cloud Firestore, Functions and Storage make up a neat whole that is easy to set up quick, without having to set up complicated permission systems or requiring a deeper understanding of concepts like CORS, proxies, etc.

2.2 Project hierarchy

The feedback service that this document describes is comprised of several smaller React applications. In one typical case, a service would be a single website that provides something for the user or customer. This project, however, consists of 3 applications, with 1 other application related to the same product. The following is an overview of folder hierarchies and application parts.

2.2.1 The monorepo

The whole project is organized inside a bigger Git repository containing all the components (see Figure 3). These kinds of repositories are commonly called *monorepos*. Monorepos as a system mainly exist to keep relevant files together in version control, but there is more to them as well. More complex application wholes can have for example *models*, or object schemas, that are to be used in more than one application. It would then be convenient for these model files to exist

somewhere in the monorepo where every application part has access to. The problem is, they cannot simply be imported into a React project, because they exist outside the *src (source)* folder of said project and React disallows imports outside the source folder. Fortunately, there are a couple ways around this. One way is to turn the *common* package into a npm package. This can be done locally, without exposing the code publicly to the npm registry. In the general working life consensus, application code is virtually always owned by someone, which is the reason it is kept private. Another, more recent method of achieving a similar result is by using *yarn workspaces*.

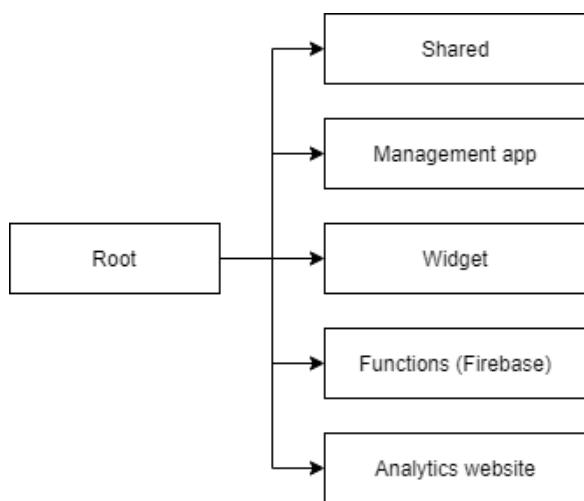


Figure 3. Repository folder structure

2.2.2 Yarn workspaces

Yarn is another package manager, like npm. Opidi (2020) has written that the idea behind development of *yarn* was to overcome the performance issues of npm, both speed- and security-wise. Both package managers still have their perks and uses, and npm still seems to be more popular, as can be seen in Figure 4. According to Opidi, npm has been lately bridging the gap between it and *yarn*, and currently the differences aren't that big anymore.



Figure 4. Npm-yarn popularity comparison

Yarn has a very convenient feature for managing projects within a monorepo, called *Workspaces*. Workspaces “allows you to setup multiple packages in such a way that you only need to run yarn install once to install all of them in a single pass” (Workspaces documentation). What is convenient about it, is that the project can have a dedicated “shared” (or, e.g., “common”) folder, which includes object models, possibly shared SASS or CSS styles and anything else that the developers may want to use in multiple projects within the same repository. When setting up projects this way, models or styles do not have to be copied to each project, leading to much better maintainability.

Yarn workspaces are set up in package.json files. Virtually every Node-based application has a package.json file. Package.json is kind of a metadata file, which includes things like which packages to install when running “yarn install” or “npm install”, license, package name, version, scripts (for starting the application, building etc.), and many more things. A *workspace* is set up by creating a package.json file to the *root* of the monorepo. This package file must then have a property called “workspaces”, which accepts an array of strings, which in turn represent the project names (see Figure 5).

```
{  
  "private": true,  
  "workspaces": ["workspace-a", "workspace-b"]  
}
```

Figure 5. Workspaces in package.json

Each different part of the project must be appropriately named for Workspaces to recognize them. Here, “workspace-a” and “workspace-b” represent different projects inside the monorepo. Under the hood, each workspace acts basically like a npm package: workspaces simply allows listing them as dependencies.

2.2.3 The management application, or “the editor”

The management application serves as kind of a content management system for Apec’s clients. Access to the management application (from now on, editor) is an exclusive area for paying clients. The editor has multiple levels of privileges: admin, editor and member. Admin rights are only given to Apec’s management, and for rather obvious reasons, the developers. Admins are allowed to manage organizations, the responsible personnel for each organization, organization members and widgets, which means basically everything. Editors are the responsible personnel of their own organizations that have privileges to manage widgets and members within their organization. Normal members only have “read access” within their organization.

Besides member management, the purpose of the editor is to allow composing of widgets with a custom editor. The “widget builder” allows for 2 different kinds of widgets: “ratings”, which include a star or smiley rating view on a scale from 1 to 5, and a number style rating with a scale of 0 to 10. The reason a scale of 0 to 10 is used is because it can be used to measure Net Promoter Score, which is a universally accepted measurement tool of customer satisfactory and loyalty. NPS is calculated by subtracting the percentage of “detractors”, or unhappy customers that have given a rating of 0 to 6, from the percentage of “promoters” (very happy customers, who will be likely to put out good word-of-mouth), who in turn have given a rating of 9 or 10 (What is NPS? Your ultimate guide to Net Promoter Score).

The other type of widget is called “values”. The builder gives a pre-set list of values that the user can select from, which will be displayed in the widget as simple “thumb up - thumb down”- questions (i.e., “Reliability”, or “Responsibility”). In addition to the question type selector, one must describe the main question. This could be something like “How well do our values match our image?” for a values-type widget, and “How satisfied were you with our service?” for rating-type widgets. The editor is also used for another service of the same company but is not a part of this thesis.

2.2.4 The analytics site, or “results site”

At the moment of writing, there was no specific given name for the analytics, or results site. The purpose of the results site is to display all the sent customer feedback in an easily digestible form. This means basically graphs – be it bar, pie or line charts. When multiple companies of the same industry receive feedback, they can compare their results, while common users of the site can compare ratings of companies to make buying decisions, for example. The site will work much like Trustpilot.

Though most data on this site will be public, an exclusive member-area will be implemented. During this stage of development, it is unclear what kinds of features the member area will consist of. Apec’s clients should be able to get deeper insights from this member area, and possibly control their “company view” on the public site. Due to time restraints, the analytics website was left unfinished, and is set to be finished at a later date.

2.2.5 The widget

The last part of the whole is the widget itself, which is another React application, albeit not one set up in a typical way. Where a lot of “React apps” are created using the official “create-react-app” bootstrapper or expo, the widget has a more complex custom configuration utilizing webpack. It is basically a small React app that can be placed in another app, be it a React or Vue app, or even a plain HTML document. The goal is to have the widget function on any site without messing up the site’s styling, and to be easily “copy-pasteable” also by less technical personnel. The widget is loaded from a script uploaded to Firebase Storage.

3 Backend

This section dives into the backend of the project. The backend is responsible for handling “behind-the-scenes” actions, such as storing and fetching data to and from the database. The section will go over different approaches to backend setup and building.

3.1 Using serverless computing

Web development as a whole has become simpler and more accessible throughout the years, with more and more tools available. In earlier times, setting up a more complex web service would require digging deep into server technologies, which is not only a lot of work, but can get costly due to having to maintain expensive physical devices virtually non-stop (excluding maintenances). Some organizations (especially public ones) and companies still use the “custom server” approach for specific reasons – usually the incentive is to have absolute full control of the backend.

In modern development, however, more and more developers resort to cloud computing. This is especially true in software companies who primarily focus on pure software and application development instead of complete solutions, where *everything* including the servers are built on-demand. Microsoft Azure’s (also a cloud service provider) website has a concise, to-the-point description of cloud computing:

Simply put, cloud computing is the delivery of computing services—including servers, storage, databases, networking, software, analytics, and intelligence—over the Internet (“the cloud”) to offer faster innovation, flexible resources, and economies of scale. You typically pay only for cloud services you use, helping you lower your operating costs, run your infrastructure more efficiently, and scale as your business needs change.

3.2 Selecting the right cloud provider

The three biggest cloud service providers are Google Cloud, Azure Web Services (AWS) and Microsoft Azure. Naturally, each have their own pros and cons. When selecting the cloud service provider for a project, multiple factors need to be considered. This is also a bit of a matter of opinion. What’s important is analysing the project and picking the best solution on a per-project basis. For some projects, using a huge AWS architecture with a more expensive database could be overkill,

especially when looking at the amount of work one needs to put in it, versus an easier cloud platform. An overall comparison of the major features of each platform is listed in Table 1, as presented by Harvey (2021).

Table 1. Cloud service overall comparison

Vendor	Strengths	Weaknesses
AWS	<ul style="list-style-type: none"> • Dominant market position • Extensive, mature offerings • Support for large organizations • Extensive training • Global reach 	<ul style="list-style-type: none"> • Difficult to use • Cost management • Overwhelming options
Microsoft Azure	<ul style="list-style-type: none"> • Second largest provider • Integration with Microsoft tools and software • Broad feature set • Hybrid cloud • Support for open source 	<ul style="list-style-type: none"> • Issues with documentation • Incomplete management tooling
Google	<ul style="list-style-type: none"> • Designed for cloud-native businesses • Commitment to open source and portability • Deep discounts and flexible contracts • DevOps expertise 	<ul style="list-style-type: none"> • Late entrant to IaaS market • Fewer features and services • Historically not as enterprise focused

Market shares for the third quarter (Q3) of 2021 indicate that AWS clearly dominates the cloud service provider market with a 32% market share, with Azure coming in second (21%) and Google third (8%) (Statista, 2021). AWS is therefore the dominant provider, but not necessarily automatically the “best” for every occasion.

3.3 Building the backend

As mentioned in 2.1.4, for this specific project, Google Cloud Platform was selected, more specifically, Firebase. Firebase is a platform under Google Cloud, which is primarily meant for mobile and web applications. It includes several products for different backend tasks. The ones used in this project are Authentication, Firestore (a NoSQL database), Functions and Storage.

Firebase Authentication is responsible for storing user accounts. Firebase has its own built-in methods for communicating safely with Authentication. This means that there is no need to set up custom encrypt- or decrypt- functions for password hashing or the likes, which would be the case when using a custom server. This however applies to user accounts only, and any other possible app-related passwords must be handled with external logic. Firebase Authentication is used to log in to the editor, and the results website's member area.

Cloud Firestore is Firebase's newest database solution, with the only other one being Realtime Database. Both are NoSQL databases, with a few differences.

3.3.1 Environments and Firebase

Generally, in software projects, there are multiple environments, which are basically different versions of the product. A deployment environment means that the software being developed can reside in different "locations": local (the developer's computer) or a server, but the important part is that different environments usually have different variables, and commonly even different databases etc. that are separated from the live version. Environments exist to help developers safely test and develop applications before they are deployed to production, i.e., testing a "mass delete" function cannot affect the live version and accidentally wipe out all the precious customer data. In a development environment, however, this would not be a catastrophe, as the data is simply dummy data created by the developer(s).

There are multiple different environments, and each have their own use. Environments work differently in different programming languages, and follow different naming conventions etc. This thesis is about React, Node.js and JavaScript, so environments will be explained based on that. Table 2 will go over the main functionality of the most common ones.

Table 2. Development environments

Environment	Description
Local	The local environment only exists on the developer’s workstation. This means that no other developer or user has access to the information used here. In a typical React application setup, the environment file would be named <i>.env.local</i> or <i>.env.development.local</i> . These local environment files are also usually added to <i>gitignore</i> , so they don’t accidentally get committed to version control and mess up other developers’ environments.
Development / debug	An environment for the developers only. Usually, the client has no access to this data, as it is meant for developers to code on and test new features before deployment to staging and production. These files are usually named <i>.env.development</i> and should be added to version control.
Staging	In a typical case, a staging environment is a “mirror” of the production environment but can use the same backend as the development environment. In the case of a website, it could be deployed to the staging environment so the client can test it “in hiding”, meaning a basic user still wouldn’t have access to it without proper permissions, a complex link or something else that is disclosed.
Production / release	The production environment is the “live” environment, where most of the traffic and usage happens – this is because it is (usually) open for public. The production version of the product has normally undergone extensive testing before getting deployed, and should be the most “perfect” and stable version of the product.

At the time of writing, Firebase does not yet have capabilities for multiple environments within the same Firebase project. Setting up environments for a project that uses Firebase is therefore rather simple, but laborious. This is due to the requirement of multiple Firebase projects, one for each environment. The different used environments are going to be called “dev” and “prod” from now on, standing for development and production environments.

Integrating to a Firebase project happens by using the credentials gotten from the project's project settings page. This is where environments step in: the credentials from the dev project are copy-pasted to your *.env.development*, and the prod credentials to *.env.production*. When using create-react-app to set up a React project, the environment variables should be named in a certain style, such as:

```
REACT_APP_NOT_SECRET_CODE=12345
```

So, let's imagine the project has a Firebase app ID of 12345; `REACT_APP_FIREBASE_APP_ID=12345` would be added to the file. These variables are exposed in the code through *process.env* and can be used such as:

```
process.env.REACT_APP_FIREBASE_APP_ID
```

One major upside of using environment variables is also the possibility of hiding certain sensitive data, such as API keys, from public users, so they cannot be misused.

3.3.2 Cloud Functions

Among cloud service providers, an important feature are server-side functions, also known as serverless computing. These can be named differently on different providers: AWS for example has Lambda functions, Azure has Azure Functions, while Google's equivalent is called Cloud Functions. The purpose of these functions is to execute server-side code. These functions normally have access to a lot more data than their respective client has, for obvious reasons: clients are easily reverse engineered. One common rule of full-stack development is to "never trust the client". Of course, this means a website, a mobile application, a game or whatever application is accessing the backend, not a human or company client. The users of these applications can easily transform the data the client sends, leaving weakly built backends vulnerable. This means that the backend should be responsible for handling as much as possible, while the client should only send information like a user's username and password, with possible additional authentication data in request headers.

Cloud Functions are coded pretty much like normal Node.js backend functions. Functions also support Python, Go, Java, .NET, Ruby and PHP; basically, it doesn't matter which language is used in the backend, but for the sake of consistency, using JavaScript in a JavaScript application keeps the codebase the most coherent. There are two types of functions: HTTP and event-driven, a.k.a. "on call -functions" (such as the one in Figure 6). HTTP functions are invoked via HTTP requests, while the latter ones are called from applications using the Firebase APIs.

```
exports.getWidget = functions.region(region).https.onCall(async (data, context) => {  
  
  const validationSchema = Joi.object().keys({  
    id: Joi.string().required(),  
  })  
  
  const { id } = await validationSchema.validateAsync(data)  
  const widget = await admin.firestore().collection('widgets').doc(id).get()  
  if (widget) return widget  
  
})
```

Figure 6. A simple Firebase "onCall" Cloud Function

Functions can be configured in many ways; their region can be configured, they can be set up as scheduled functions, etc. These functions are then deployed using the Firebase CLI (command-line interface) and will reside on Google's servers. An important thing to note about Cloud Functions is that every function has admin privileges – this means that every function will bypass any security rules that are set up in the database. This must be considered when writing functions, because in many cases, security must be handled within the function itself.

Cloud Functions should be used to perform more advanced tasks in the backend, where simple API calls using the Firebase SDK and Firebase's security rules are just not enough. This is because of the fact that Firebase rules do not have access to nearly as many properties as a Cloud Function has.

3.3.3 Firestore versus Realtime Database

Up until October 2017, only one database option existed in Firebase – the Realtime Database. It was designed with mobile and web applications in mind. Both have their own use cases, and the developer should consider on a per-project basis which one to use. It all depends on the application and functionality it needs to handle. A previous use case for Realtime Database was a chat on a website. The feature had to have a clear, rapidly changing structure for multiple regions, that could be emptied with a single click of a button. Realtime Database makes operations like this easy, as only the “root item” needs to be deleted to also delete everything under it. This is not the case with Cloud Firestore; deleting a document that has a subcollection of documents is a much more complex task and should be handled by Cloud Functions.

Realtime Database is very simple to use, and also fast. However, it is poor at handling more complex security rules – for something like server-side spam protection without the use of Functions, a more complex solution had to be implemented. Cloud Firestore’s security rules on the other hand are much more logical to write and allow for better security checks. It would seem that Google has actively listened to user feedback among other things and built Firestore to be “Realtime Database 2.0”, with many quality-of-life improvements.

The Firebase website has a tool for selecting the database that is better for the project. Table 3, referenced from Firebase’s documentation website, also demonstrates the differences between the two options. Kerpelman (2017) also has summarized the main differences in a blog post: he first points out that Firestore has “better querying and more structured data”. Firestore is also designed to scale, which is a good feature when the developer wants to be prepared for a smash hit, however unlikely that may be. Other edges over Realtime Database, according to Kerpelman, also include “easier manual fetching of data”, “multi-region support” and a “different pricing model”.

Table 3. Firebase database feature comparison

Feature	Realtime Database	Cloud Firestore
Data model	One large JSON tree	Collections of documents
Realtime and offline support	Offline support for iOS and Android clients	Offline support for iOS, Android and web clients
Presence	Supported	Not supported natively
Querying	Deep queries with limited sorting and filtering functionality	Indexed queries with compound sorting and filtering
Writes and transactions	Basic write and transaction operations	Advanced write and transaction operations
Reliability and performance	A regional solution	A regional and multi-region solution that scales automatically
Scalability	Scaling requires sharding	Scaling is automatic
Security	Cascading rules language that separates authorization and validation	Non-cascading rules that combine authorization and validation
Pricing	Charges only for bandwidth and storage, but at a higher rate	Charges primarily on operations performed in your database (read, write, delete) and, at a lower rate, bandwidth and storage

3.3.4 Database design

There is a reason people are hired to *specifically* design databases. As applications grow, so do the database models usually. This means that if the database isn't well-designed from the start, it will

get messy and complicated over time. On databases that have millions of users interacting with them, a redesign isn't usually an option, as it can mess up relations among other things and cause all sorts of issues. There are many general guidelines to database and object model design, which depend on the *type* of the database.

A database can be relational, or non-relational. A good example of a relational database system is MySQL, which is among the most popular ones. In a relational database, the *tables* are connected to each other with *primary* and *foreign keys*. A table represents a collection of documents or objects. Each document has a distinct value that is used as the primary key, for example, a "Users" table could have a collection of "User" items, where "userId" would be unique to each document. This is the primary key. A foreign key, on the other hand, "is a column or a set of columns in a table whose values correspond to the values of the primary key in another table" (Primary and foreign keys N.d.). The biggest upside of a relational database is integrity. A "User" can be connected to multiple, say, "Order" documents. In an Order document, the foreign key would be the user's userId. This forms a relation between the documents. Because the order is dependent of that specific user, a user document cannot be deleted without deleting all the related orders first. This means the database cannot be easily broken, which in turn could lead to potential application crashes.

The other database type is non-relational. In these types of databases, there are basically no restrictions on the object models. There could be a table named Users, but each User document can have differently named columns, for example, because non-relational databases do not force an object model. This does, however, force the developer to maintain caution when writing code. Sometimes a changed object model can cause hours of headache due to the bugs it introduces. Non-relational databases are typically called NoSQL databases. The most popular NoSQL database system is MongoDB. NoSQL databases usually have *collections* of documents instead of tables. Tables and collections work similarly, as they are just "containers" for multiple documents. Even though some call the lack of integrity in NoSQL databases a downside, the relational system of MySQL and other relational database systems can be mimicked in NoSQL databases as well by using *validation* and generally clever design. To properly validate and force object models in NoSQL systems, it is common to use a third-party package, such as Joi. Joi is a npm package meant for Node.js, where the rules for an object model are described, and the incoming request is validated

by passing the model schema, along with the request, into Joi's validation function. Joi is commonly used in backend functions, such as Lambdas or Cloud Functions.

Microsoft has published a useful guide for database design on their support section. The general recommendation is to start by thinking, what is the purpose of the database? In this case, the intent is to enable creation of widgets for Apec's customers, displaying collected data and managing the client's organization, such as access levels, who has the privileges to create and delete widgets, etc. The next step is gathering and organizing the required information.

The database of this project consists of 3 major collections: organizations, users and widgets. The organizations collection stores data of Apec's clients. The users collection stores information of the management system's users, such as privilege levels, which organization they belong to, etc. However, as storing access roles in a database can potentially be very unsafe, due to the ease of editing values in the client, additional custom role tokens are stored in Firebase Authentication. The last collection, widgets, stores the widget information used for displaying the widgets. The database structure is visualized in Figure 7 below.

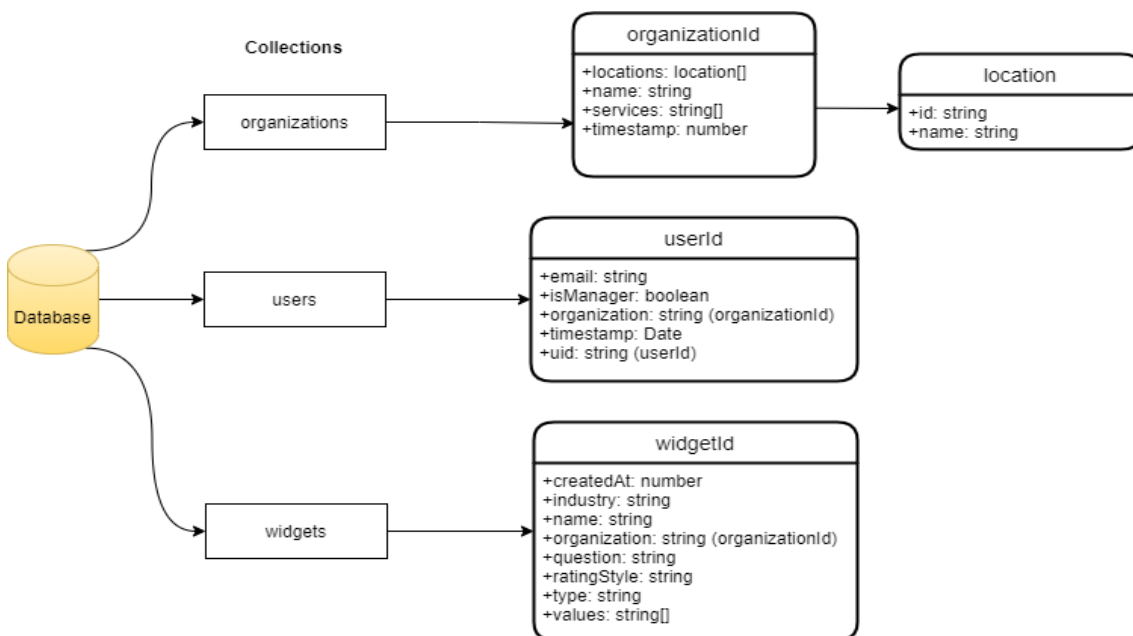


Figure 7. Firestore database model

4 The widget

This chapter describes major points regarding the widget's development.

4.1 Maximizing user-friendliness

As said before, the widget isn't built for Apec itself, but rather Apec's clients. To reduce the need for instructions from Bromeco's behalf, naturally the service should be as simple as possible.

When the project was just starting, development had to start from "how do I make it so that people need to have minimal technical know-how in order to add this to their site?". React was in mind from the start, after doing some research that these kinds of projects can be made with it. React aside, it needed to work on *any* website.

It couldn't be made into a React component, because in the end only a fraction of websites are built with React. So, what do all websites have in common? Every website, React-based or not, has an index file, usually named *index.html*. In a React application, this file is found in the *public* folder. Within that html file, there is a possibility to load external scripts. A rather obvious choice was to have the editor build a widget-specific embed code that would live in the index html file.

Many developers could agree that script placement matters. Regarding script placement, there are basically two options: between the head tags, or between the body tags. Placing the script between the head tags means the script will be loaded before the rest of the page – this is due to the fact that code is read from top to bottom. In the case of the widget, however, this is not what was wanted; instead, the widget can take its time to load. Everything else on the page should be requested and loaded first, and the widget would come last. The widget's code should be *non-blocking*.

Blocking code means the code is *synchronous*, and non-blocking code is *asynchronous*. Synchronous code essentially means, that if a function is synchronous, the execution of everything else is paused until said function has finished. Asynchronous functions are common in for example calls to an API (application programming interface), where data is requested from. In an asynchronous function, the request is sent, but the major difference is that code execution isn't paused, but instead continues, and the response to the previous request arrives once it's ready (see Figure 8).

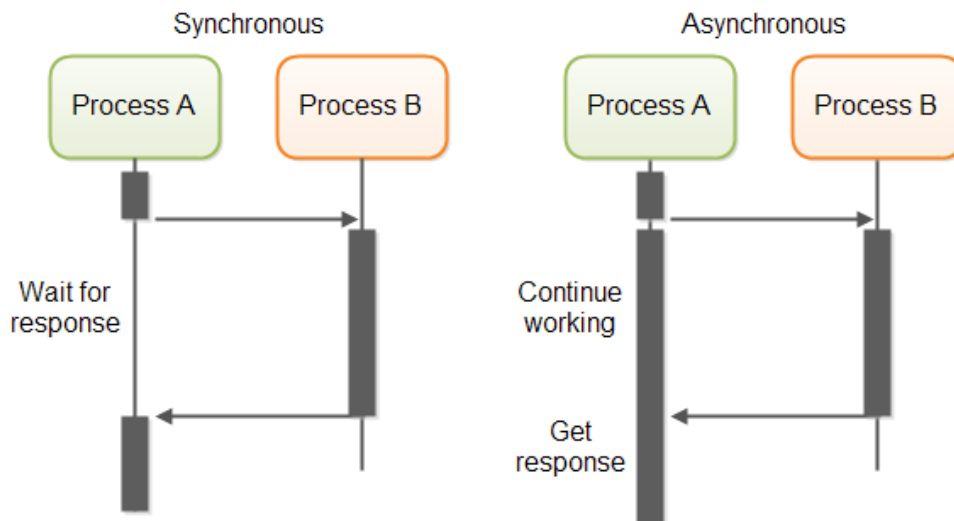


Figure 8. Synchronous and asynchronous operations

In many modern applications it is important to design around asynchronous behaviour. On websites that depend on APIs, the required data may not be available on page load, therefore it can't be used, and trying to use non-existent data is a recipe for an application crash. Figure 9 demonstrates usage of both an asynchronous and a synchronous function in web development. Both request data ("posts") from a fictional API.

```
// An asynchronous function, which enables the use of "await"
async function getPosts() {
  const response = await MyApi.getPosts()
  const { posts } = response.data
  // Do something with posts
}

// A synchronous function doing essentially the same thing
function getPosts() {
  MyApi.getPosts()
  .then((posts) => {
    // Do something with posts
  })
}
```

Figure 9. Example of an asynchronous and a synchronous function

The widget's user interface is obviously another facet that needs to be considered when thinking user-friendliness. It should have as little information as possible, and "navigation" should be streamlined. The solution was to simply use 2 elements: the question, and a row of buttons. The widget could ask something like "How satisfied are you with our products?", following a row of five star or possibly smiley buttons. When the user clicks one of them, a request is made to the server to store the answer.

4.2 Implementation

4.2.1 Getting started with the code

There were multiple things to consider before starting to code the widget application. The first was selecting the right tools for the job and finding out if React is actually fit for a use case like this. Another thing to consider was efficiency: should everything be made from scratch? Luckily, a "boilerplate" widget project was found, called `embeddable-react-widget`, made by Benjamin Boudreau. The starting project made by Boudreau wasn't created with CRA, but instead features a custom webpack setup – this made things a little harder than usual, due to getting accustomed to a typical CRA-bootstrapped application.

The project was set up with many conveniences, such as `Cleanslate`, which prevents style conflicts between the widget and its host site. `Cleanslate` is simply a stylesheet that "is used to reset the styling of an HTML element and all its children, back to default CSS values. It is composed exclusively of `!important` rules, which override all other types of rules". It is commonly used in JavaScript widgets, due to the nature of widgets (external "applications" inside other web applications that could easily mess up the styling on its host website) (`Cleanslate's Github page`).

The starting point of the widget was very simple and minimalistic, as can be seen in Figure 10. Starting templates with minimal code are often easier to start customizing to the project's needs, if the developer wants to build a custom product instead of just e.g., prototyping.

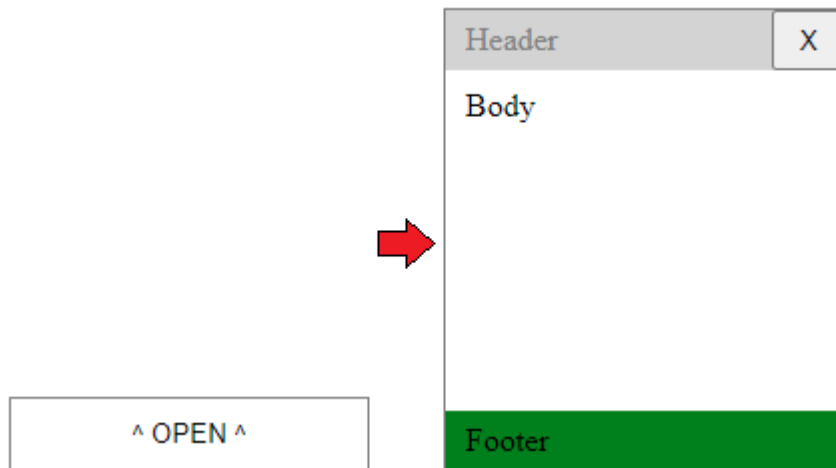


Figure 10. embeddable-react-widget's starting point in closed and opened state

As each widget should be different, the same exact code could obviously not be used, but instead the widget data must be loaded first before displaying the widget at all. The widget data defines what is shown (or asked) in the widget, and where does the answer data get stored. As each widget has a custom identifier (ID), the data is fetched from Firestore using that ID. The ID is included in the embed code of the widget. The data is loaded asynchronously; this means that the widget will not appear right as the page itself loads, but rather when it receives the required data, possible a couple seconds later. Loading a widget this way, with the ID directly available in the website's source code, however, introduces a couple concerns, which will be discussed in 4.3.1.

Once data is loaded, it is stored to the application's state. The state is responsible for storing information on what to display on the widget: the question, type of buttons, etc. Most of the work on the widget affected the front end, as in how the widget looks and feels. Apec, as a company that also specializes in graphic design, provided plans for the layout. Figure 11 shows a "rating" widget, with the value 4 out of 5 selected.

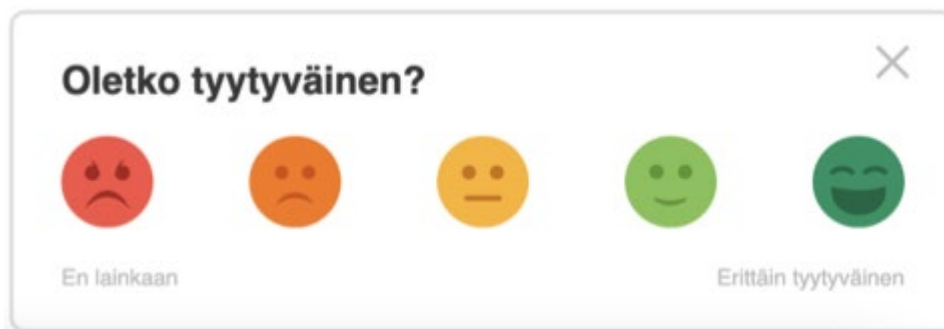


Figure 11. An example widget

4.2.2 Build and deployment

Usually, a React application is deployed by first building it using the build script and deploying it either manually by possibly drag and dropping it to a cloud storage bucket, or by using a CLI. This is the case in an application created with *create-react-app*, or “CRA”. CRA is a bootstrapper, which enables developers to set up a new project with a single command, without the hassle of custom *webpack* setups which require a lot of additional learning. The downside of CRA is that *webpack* configuration is “hidden” and thus inaccessible without first *ejecting* it. In the case of the widget, however, things are a little different. The build script for the widget is configured to spit out a single JavaScript file in the *dist* (distribution) folder. This file has everything the widget needs, images excluded. This is the script file that each widget-including website will be loading from Firebase Cloud Storage.

The script file is uploaded to the Firebase project’s Cloud Storage bucket (Figure 12), which in turn has a set URL to access it. Cloud Storage also obeys the same kind of security rules as Cloud Firestore, so this file must have read privileges without authentication. Even though the file is in no danger of being modified by additional parties, having public read access means it’s also available for anyone to load from the bucket. Chapter 4.3.1 will go over more security-related concerns regarding the widget.

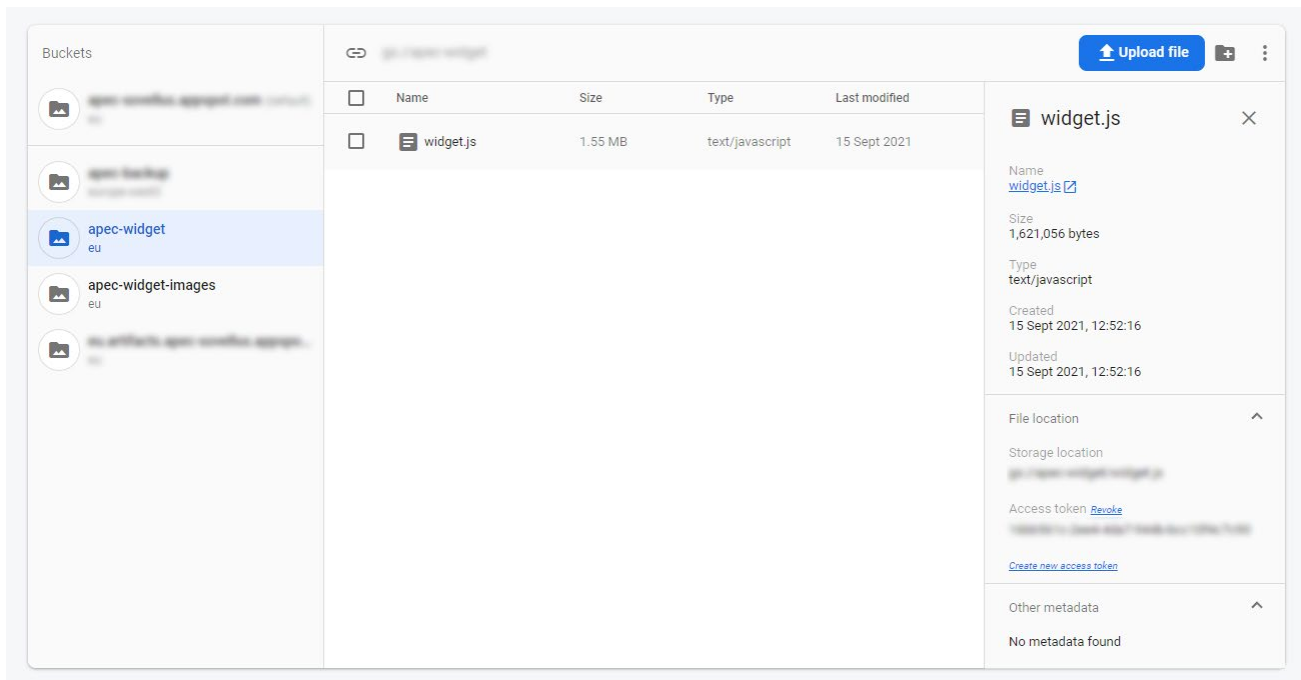


Figure 12. Widget script file inside its storage bucket

4.3 Integration to the project

4.3.1 Preventing misuse, a.k.a. handling security

It should be noted that this chapter mainly talks about hypothetical solutions or solutions on a planning level, as the project was not yet finished by the time of writing. Some of these methods have been tried and/or implemented, but have been commented out in the project, as more research and testing need to be done to confirm the best solutions.

The widget is a simple React app that has no authentication. However, it has to send and receive data to and from the backend. What is important to consider here, is that the embed code for the widget is also publicly displayed to anyone who looks at the site's source code. So, what prevents others from copying that code and using the same widget on their own website for malicious purposes?

One solution is domain, or origin locking. The editor has its own field for origin. This is stored to the database along with the rest of the widget data. Whenever the widget is loaded on a website,

the Cloud Function responsible for getting the widget would confirm that it's coming from the correct origin. If it's not, the widget will not load at all. Though it would seem like a logical solution, this solution also has a flaw. As mentioned before, "never trust the client". Because the origin of the request is sent in the request headers, it is possible to alter the request and use a fake origin that is the defined origin for the widget. With the current "easily copy-pasteable embedding system", there is unfortunately no easy way around this.

Another concern in this kind of widget is spamming. What's stopping a malicious user from just spamming bad reviews for the company, possibly even using a script? The widget's whole idea is to be as minimally obstructive as possible and very simple to use. That leaves spam-protection tools like CAPTCHA out of the equation, which wouldn't even fit inside the little widget window. One way to slow spammers down is by using the application cache, or *cookies*. Whenever a user completes the small widget survey, information of completion is stored as a cookie. This way, if the user reloads the page, the survey will appear as completed, as long as the user doesn't clear the site data.

Another way is to utilize the *X-Forwarded-For* header (also known as XFF). This header tells the IP address where the request is coming from and can be used to slow down requests; in an extreme case only the completion of 1 survey could be allowed per IP address. The XFF information would be stored with the survey answer and each answer would be validated by the following logic: if an item with the same XFF value exists, don't save the item. This again raises concerns on other aspects: how will it affect speed and cost efficiency? On every completed survey, the database needs to be queried to potentially find the item that has the same IP address. For maximum speed optimization, an index could be used where the primary key is XFF.

Naturally, as the methods described above are dependant of the client, they can be circumvented. They should prevent *manual* spamming efficiently; however, it wouldn't be too difficult for an experienced malicious user to write a script that automatically clears site data or changes the XFF header each time. Of course, it is up to that malicious user whether they see the need to actually go through the trouble.

Though mentioned that CAPTCHA is out of the question, there actually is an option for Google-reinforced spam protection – the *invisible reCAPTCHA*. This works a little differently than the normal “I’m not a robot” checkbox or image selection usually seen on websites, albeit using the same engine. The difference is, it requires no user interaction, at least in the form of confirming you’re not a robot.

4.3.2 Database connections

Communication between the client (widget) and the backend is fairly straightforward. On the initial page load, the widget requests data based on its ID, via a Cloud Function. The function validates the incoming data: it checks which ID requests the data, and the origin the request is coming from. If they match, widget data is returned. Figure 13 visualizes how the widget communicates with the backend.

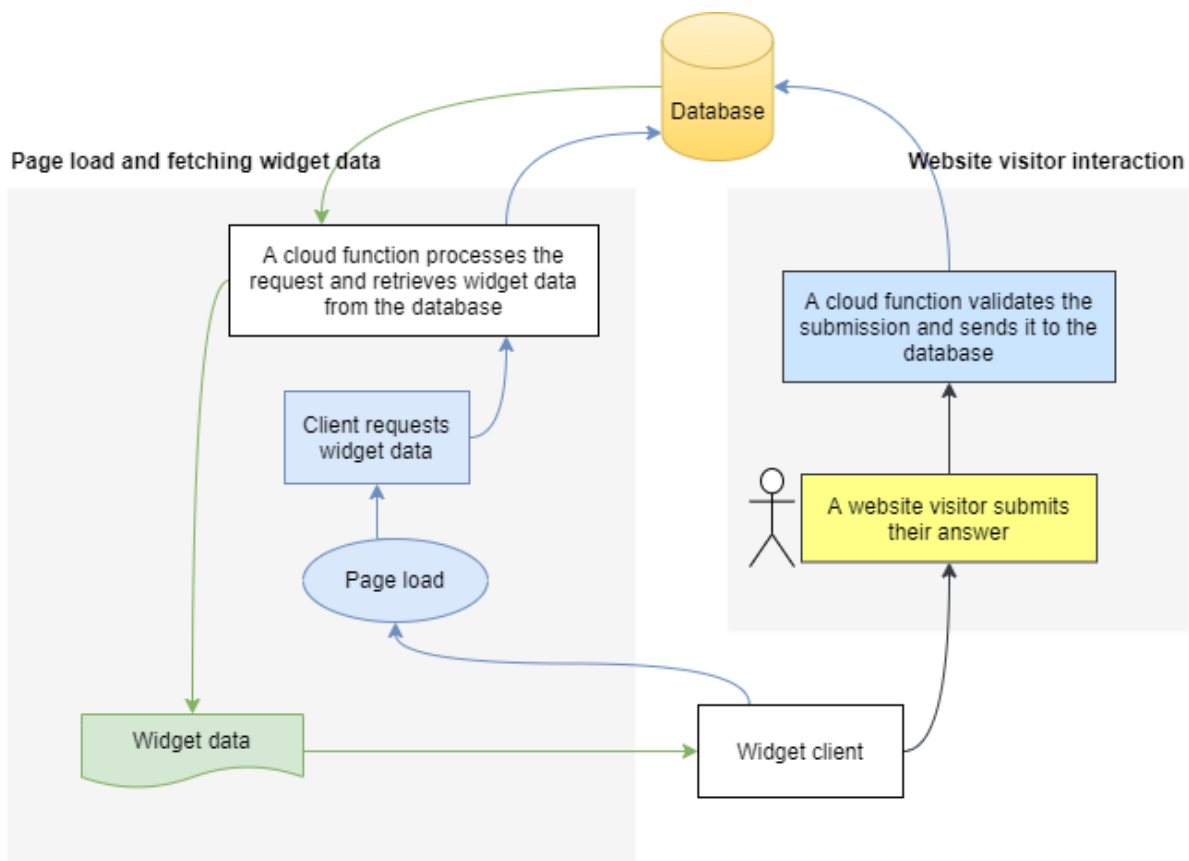


Figure 13. How the widget communicates with the backend

5 Management system

The management application basically consists of multiple table views, where each user has access to more views depending on their role or security level. More details on user permissions and views can be found in 5.1.1.

5.1 Integration to the existing system

As mentioned before, the management system houses a couple different services, and Widgets is one of them. By existing system, it is meant that the Widgets section is merely just built on top of the existing system – the management application itself already existed. Widgets is a new section added to the navigation sidebar, and on each organization's service list.

5.1.1 Access levels and security

The management application is a kind of content management system with multiple levels of access: admin, organization manager and organization member. Admins include high-level personnel of Apec and the developers (Bromeco). When Apec enables the service for a client of theirs, an organization must first be created. An organization represents the company of the client and can have multiple members within it. Admins have access to everything: they can create organizations and manage every organization's members and widgets.

An organization manager on the other hand can only manage their own organization, the lower-level members, and widgets. A lower-level member only has "read access", meaning they can only view specific information. Figure 14 shows the only view accessible to below-admin level users, as they cannot see outside their own users, locations or widgets.

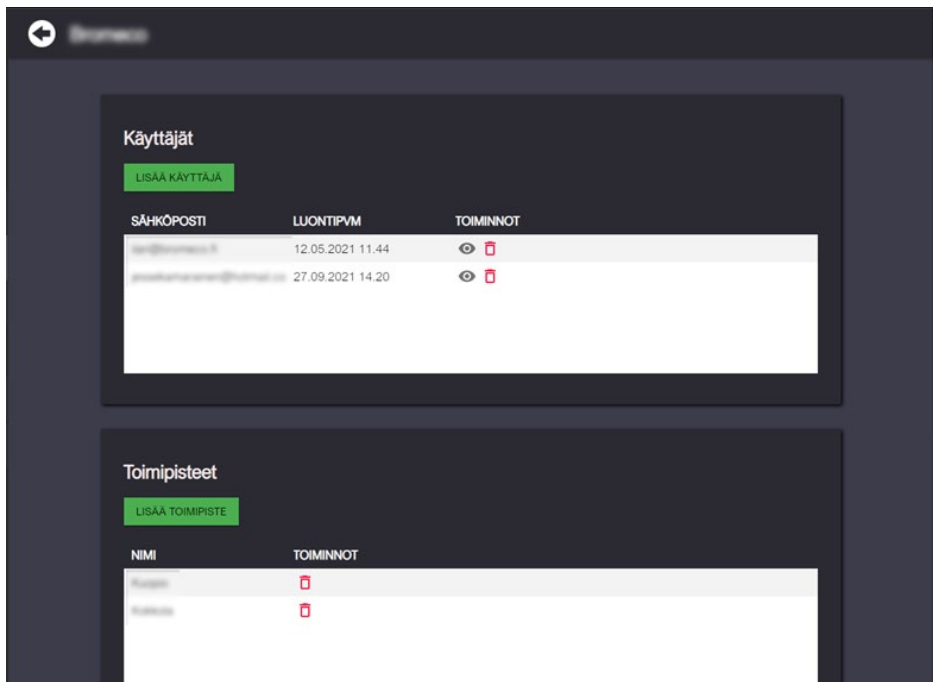


Figure 14. The organization view

What's important in a system with multiple access levels is making sure that nobody with insufficient permissions is able to send requests they shouldn't be able to. This is where Firebase Authentication and security rules come in. Firebase Authentication comes with an admin-only feature: *custom user claims*.

Custom user claims are attributes linked to user accounts that are meant to control application access, commonly in the form of roles. Despite being "admin-only", these claims cannot even be accessed, added or removed from the console; instead, they are managed with Cloud Functions. The security rules of Firestore allow for custom user claim validation. If a user has a claim called "isAdmin" set to *true*, they have access to specific sections of the database only admins should have. Claims can also be read client-side but should not be used as the only validating step, due to the untrustworthy nature of an application client. In the end, all validation should happen in the backend (Control Access with Custom Claims and Security Rules, N.d.). Table 4 lists the management application's sections along with their required access levels.

Table 4. Management application security levels

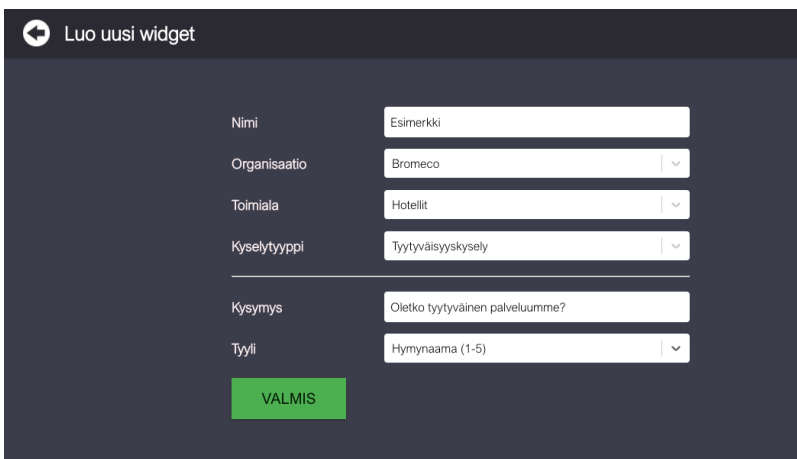
Management site section	Permitted operations by role
Users (all)	Admin: read, write, update, delete
Organizations	Admin: read, write, update, delete
Widgets (all)	Admin: read, write, update, delete
Users (organization)	Admin: read, write, update, delete Manager: read, write, update, delete User: none
Locations (organization)	Admin: read, write, delete Manager: read, write, delete User: read
Widgets (organization)	Admin: read, write, update, delete Manager: read, write, update delete User: read (get embed code)

5.1.2 Preventing loss of the client's existing data

As an application is being developed, many things tend to change, one of them sometimes being the data structure. The widget service was built next to another Apec's service, and both services use the same database, because they share some common components: *organizations* and *users*. When expanding a database's data structure, the developer must be very careful when working with the new implementations, as a mistake can potentially overwrite existing data and render the other service unable to work properly. This is rarely an issue with relational databases, but Firestore follows the NoSQL collection-document-model. To tackle this issue, it is very much recommended to use different environments for development and production. Security rules should also be in place to validate incoming requests, and to prevent accidental deletions and overwrites.

5.2 Widget editor

The widget editor was built from another similar implementation included in Apec’s services. The editor had to be simple to use, as it was designed for non-technical people. A key factor in simple user interfaces is minimalism, meaning there should be as little information as possible, but without leaving the user confused. The form in Figure 15, for example, is quite minimalistic, which makes it simple to use.



The screenshot shows a dark-themed web form titled "Luo uusi widget". It contains the following fields:

- Nimi: Text input with "Esimerkki" entered.
- Organisaatio: Dropdown menu with "Bromeco" selected.
- Toimiala: Dropdown menu with "Hotellit" selected.
- Kyselytyyppi: Dropdown menu with "Tyytyväisyyskysely" selected.
- Kysymys: Text input with "Oletko tyytyväinen palveluumme?" entered.
- Tyyli: Dropdown menu with "Hymynaama (1-5)" selected.

A green button labeled "VALMIS" is positioned below the "Kysymys" field.

Figure 15. A simple form for creating widgets

The editor is a simple page with a form. In the editor it is defined what the widget will look like, and what the company wants to ask the website’s visitors. There are two types of surveys: a rating survey, and a “values” survey. In the latter, the visitor is presented with (potentially multiple) company values, with a thumb up and a thumb down button. The creator may reorder the value blocks in a drag-and-drop style editor (Figure 16). When the widget is ready and all the required fields have been filled, a “Save” button is enabled.

The screenshot shows a dark-themed web interface for creating a new widget. At the top left, there is a back arrow and the text 'Luo uusi widget'. The form contains several input fields: 'Nimi' with the value 'Esimerkki', 'Organisaatio' with a dropdown menu showing 'Bromeco', 'Toimiala' with a dropdown menu showing 'Hotellit', and 'Kyselytyyppi' with a dropdown menu showing 'Arvokysely'. Below these is a 'Kysymys' field containing the text 'Toteutuvatko yrityksenne arvot?'. A button labeled 'LISÄÄ ARVO' is positioned below the question field. Underneath, a drag-and-drop editor is visible, showing two widget cards. The top card is labeled 'Arvo' and contains a dropdown menu with 'Luottavuus'. The bottom card is also labeled 'Arvo' and contains a dropdown menu with 'Rehellisyys'. A hand cursor is shown over the 'Rehellisyys' dropdown. At the bottom of the form, there is a green button labeled 'VALMIS'.

Figure 16. Creation of a widget with the drag-and-drop editor

5.3 Use cases

This chapter demonstrates a typical use case for the management application from different perspectives: admin, organization manager and organization member.

5.3.1 Use case 1, the admin

Apec has made a contract with a new customer, from Company X. Apec's person in charge logs in to the management system. They are presented with a sidebar with multiple sections for navigation, and a list of organizations (Figure 17). The admin then adds the new customer's organization to the list. This enables the creation of widgets for said organization; however, it will not enable that organization's personnel to actually see or do anything within the application. The admin switches to the "users" tab and creates an organization manager-level user for the customer they made the contract with. This gives the customer access to the system.

Organisaatioiden hallinta

LISÄÄ ORGANISAATIO













NIMI	LUONTIPÄIVÄMÄÄRÄ	TOIMINNOT
Yhteinen Helsinki	20.05.2021 10.47	 
Yhteinen Group Oy	07.06.2021 12.06	 
Yhteinen	11.05.2021 11.57	 
Yhteinen Helsinki	21.06.2021 10.37	 
Yhteinen	11.05.2021 11.58	 
Yhteinen	13.09.2021 14.10	 

Figure 17. The organizations view

5.3.2 Use case 2, the organization manager

The organization manager logs in for the first time. What they see is a view of their organization page – members, locations and widgets. Other members of the same company need to have permissions to view detailed reports of the collected data on the results website – therefore the manager creates accounts for them by adding them to the organization’s users list.

The manager proceeds to create a widget for the company’s website. They ask the question “How happy were you with our service?” and select “Rating” as their survey type. After submitting the form, they are thrown back into the organization view. This time the new widget is listed, with a few buttons next to it. The manager clicks the button representing code and is presented with a modal displaying the embed code.

The manager copies the code, and being a non-technical person, forwards it to the website’s developer. The developer’s responsibility is to add the embed code to the correct section of the website, which is right before the closing `</body>` tag.

5.3.3 Use case 3, the organization member

The member, possibly a sales specialist within the organization, logs in to the management application. They see the same organization page as the manager, but without the members list. While the results website's member section has more detailed data, the management app's purpose is to create, edit and delete widgets and manage members; therefore, the only accessible option for a basic member is seeing the embed code.

6 Results website

6.1 The purpose

The results website will include a public and a member section. The public section's purpose is to function similarly to websites like Trustpilot. A visitor may go check reviews of a company to make decisions, whether they want to use their services or not, based on user feedback. On the member section, Apec's clients are able to see more detailed reports on their data, and manage certain parts of their company's page, or "card", on the site.

User feedback is essential in product development. Based on feedback, important business decisions can be made. The detailed reports would be able to show graphs and information, and enable interpretation for e.g., the following:

- How many good or bad reviews they got?
- When, or in which kind of time windows they got the good/bad reviews?
- What was going on with the product, company, or anything related when the good or bad reviews were given?

This again allows the company to adjust back and forth, depending on the factors and events leading to the good or bad reviews.

6.2 Implementation

The results website is a React application, bootstrapped with CRA. At the time of writing, it only has a draft of a frontpage and a couple of routes configured, but due to missing layout and some

other specifications, the website is a stud that is yet to be fully implemented in the following months. The following chapters, similarly to 4.3.1, speak more in a hypothetical manner.

6.2.1 Handling potentially big data

When an application is new, and there aren't too many users, the amounts of data and traffic are going to be rather low. But what about when a few widgets are added to websites with potentially tens of thousands of visitors each month? At that point the importance of good database planning rises greatly. The widget requires potentially 1 click from a visitor to add another document to the database. Before long, there could be thousands of documents. Firestore allows collections inside documents, so every widget has its own collection of answer documents.

Problems arise when that data needs to be processed differently, such as listing all answers from a specific *industry* instead of a specific company, or widget. Reading and writing into the database costs money, and both operations should be optimized to stay as low as possible. Not only the cost, but speed will suffer as well, if the queries are too complicated and for example go through multiple nested collections using a *for loop*. If possible, it should be planned beforehand, what are the important properties of the documents that will be frequently queried.

A major player in database optimization are *indexes*. As Codecademy puts it, "Indexes are a powerful tool used in the background of a database to speed up querying. Indexes power queries by providing a method to quickly lookup the requested data". Indexes in databases work like indexes in a library – they are pointers to the data. They are essentially *tables* which' job is to contain information on where the desired data is found.

Imagine walking into the Library of Congress and being given the task to find a specific publishing within 10 minutes. Would you be able to complete this task within the given time frame? The Library of Congress is considered the largest library in the world and it houses approximately 170 million items. Now, the Library of Congress is not a regular library where the public can check out books at will, but if you are like us, you know the challenge should not be too difficult. In fact, the first thing we would do is ask for access to the library's index because indexes contain all the necessary information needed to access items quickly and efficiently. (What is a Database Index?)

In this project, the answer documents inside the collections included in widget documents would be indexed. This is because first querying each widget document to then query the answers collection inside them would be very inefficient, and over time, costly. Indexes are used solely by the results website, which is responsible for digesting the potentially huge amounts of data into an easily readable form.

6.2.2 User-friendliness: a bit on UI/UX design

As the results website is a public site, accessible to virtually all web users, it is more important here than anywhere else to have a solid *UX* (user experience). As more features and other building blocks are added to the application, the need for *UI* (user interface) elements grows. It is crucial to try to minimize shown information but at the same time to not leave anything important out, to get the best and most accessible experience.

Unfortunately, since the results website was left unfinished at this stage, there is no visual documentation, however, the same rules apply to other parts of the project, such as the management application. It is important to note that some of the visual content was designed by a graphics designer.

Nick Babich, a UX architect, has listed 4 golden rules of UI design on Adobe's (a leader in UI/UX design software) website:

1. Place users in control of the interface
2. Make it comfortable to interact with a product
3. Reduce cognitive load
4. Make user interfaces consistent

The first rule could be summarized into a few key things: ease of navigation, as in, the user should be able to go back and forth and stay on the map with visible information of the application's state. This can be implemented in many ways. The URL on the address bar should show which "directory" (a.k.a. route) the user is in, but elements like headers should also be used on the actual website to demonstrate the current state. Figure 18 shows the whole view of the management application: the navigation drawer (left side), with the current route selected (Widgets), and the top bar shows where the user is at within that route.

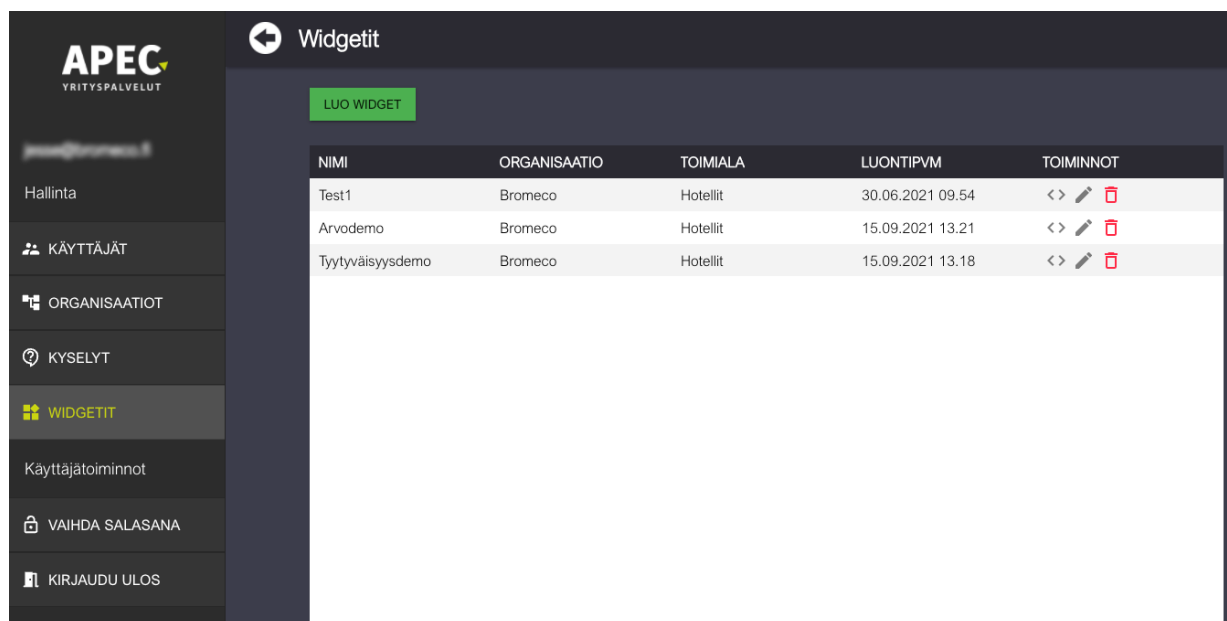


Figure 18. Widgets view with navigation

Ease of navigation is closely related to “making actions reversible”. When a user is browsing an application, they don’t want to be constantly scared of failing or breaking something, which makes a bad user experience. It is important to build the application in a way so that actions are easily backtrackable and/or reversible. This enables the user to explore the application with a peace of mind. A good example of this kind of functionality are the undo/redo functions usually found in editors. Another important thing to consider is user feedback. This is probably the most common in forms of any sort, where a text input field changes color if the text isn’t valid, or perhaps if a user tries to save the form, but the saving fails (or succeeds), a feedback text could be shown, or a redirection happens (Babich, 2019).

The second rule, “Make it comfortable to interact with a product”, comes down to a few key factors. One of them is displaying only the relevant information, as in keeping the UI elements minimal. When building user-friendly websites, they should not display more information than is needed. Another is not asking users to re-enter information they’ve already given, as most users consider this a major annoyance. Comfortability of use also includes accessibility. Accessibility could have a major section of its own, but it can be summarized as having the site accessible for people with disabilities, be it hearing, vision or motoric impairments (Babich, 2019).

The third rule Babich (2019) has written about, “Reduce cognitive load”, means that a website should require minimal mental processing to use. Information is a lot easier to process as chunks. Chunked information can show up for example in a phone number. When considering Figure 19, the bottom number is much easier to make sense of and dialing that number to a phone would lead to much less errors than the one above it.

16502388915
1 (650) 238-89-15

Figure 19. Example of information chunks

Among other cognitive load -inducing factors is the number of actions a user needs to take to get from point A to B. For this, a “three-click rule” exists, which states that a user should not have to click more than three times to get from that A to B, wherever on the site they may currently be (Babich, 2019).

Most UI designers have probably heard of Jakob Nielsen’s 10 usability heuristics advises. One of them promotes recognition over recall.

“Recognizing something is much easier than recalling it because recognition involves more cues in our brain (cues spread activation to related information in memory, and those cues help us remember information). Designers can promote recognition in user interfaces by making information and functionality visible and easily accessible. Visual aids, such as tooltips and context-sensitive details, also help support users in recognizing information.” (Babich, 2019)

In the modern web design world, heavier use of icons has raised its head. Icons mostly appear to be somewhat universal. The question is, do icons promote recognition, or recall? Take a “share” icon for example. iPhone users may have gotten accustomed to a specific icon Apple is using, while Android users are familiar with another. How does someone know that a specific icon means “Share”, when it is simply a twisted arrow, or a box that has an arrow coming out of it? A simple “share” icon can have multiple forms, as seen in Figure 20. In cases such as these, where people

have simply familiarized them, the correct term to use would most likely be "recall", as in "this icon was used in the share button of application X, and here's the same icon, so it obviously does the same thing". However, as stated above, adding a tooltip to describe what the button does kind of instantly turns it into recognizable.



Figure 20. Different common "Share" icons

The last, but definitely not least rule promotes consistency. This can mostly be interpreted as visual and functional consistency. Typography in itself would make another major section, but probably the single most important rule is to keep different fonts and font sizes to a minimum. "Using more than 3 different fonts makes a website look unstructured and unprofessional. Keep in mind that too many type sizes and styles at once can also wreck any layout." (Babich, 2017).

Functional consistency basically means, that interactable UI elements should work the same way. When pressing a button, the user doesn't want any surprises, which could lead to frustration due to confusion.

Apec's repository has a "shared" folder, which is intended to be used by every part of the application collection. The shared folder contains fonts, color and font size definitions and the like, which are then imported over to other applications (such as the widget or management app). This helps keep the styles consistent.

7 Conclusions

7.1 Time efficiency

Sometimes it is easy to “drift away” with development, meaning it is easy to start coding in things that weren’t even asked for. In client projects, budgets can sometimes be really tight, and the developer should first aim only for an MVP (minimum viable product) – this requires strict specifications on what and what not to code in. Negotiating with the client is key to staying within the given budget, as they are ultimately the shot callers; they don’t (or at least shouldn’t) want more features than they can pay for. However, due to clients not always being “tech-savvy”, due to possibly not being software engineers, the developer’s company may also give consultation and recommendations on what would be reasonable and assist the client in staying within the budget. Sometimes (more often than not, especially in the case of smaller companies) the client may ask for too much for too little, and in such cases the developer company may have to intervene.

Regarding this project up to this point, time efficiency did turn out fine; there weren’t too many setbacks that could’ve taken a lot of time, and coding stayed within the given specifications – up until there were “no specifications left”, which in turn halted development. Of course, some head-scratch inducing issues were encountered, as in nearly all projects, because no project is like the other and usually there is at least one new technology, package, library etc. to learn in each project.

The project was developed to its’ final stage (so far) over a few months. The actual hours and days spent overall, however, were much less, because other projects took a lot of time as well. Building on top of existing systems cut the development time significantly, and this is something developers should usually aim for: not everything needs to be made from scratch. Naturally, the use of third-party libraries and packages also cut the development time, due to not having to “reinvent the wheel”.

7.2 Results

The final product couldn’t unfortunately be finished within the given time frame for the thesis. This was not due to being too busy (or lazy), but rather due to not getting all the specifications and

required materials in time. The management application and the widget application were about 80-90% finished – perhaps even more functionality-wise. These were already demoed to the client. In the demo, an example was shown where a new widget is built from scratch using the management app, the embed code is copied to a sample website, and the website is run, displaying the widget. The widget would still have required layout specifications from Apec, so it could've been shaped satisfactory for the client appearance-wise. Most of the results website remained unfinished. The end of this thesis is not the end of the project, and the project will be finished at a later date.

7.3 Final thoughts

Starting the thesis process might have dragged on for a little too long, due to being employed full-time and being busy with other projects at the same time. Overall, however, the thesis seemingly turned out okay, given the small time frame. It is unfortunate that the actual project couldn't be finished before the thesis report, which caused the report to be more "hypothetical" in many parts, as in written on a "planning" or "thought" level.

References

Babich, N. 23.6.2017. 10 Tips On Typography in Web Design. Article on UX Planet. Accessed on 13.11.2021. <https://uxplanet.org/10-tips-on-typography-in-web-design-13a378f4aa0d>

Babich, N. 7.10.2019. The 4 Golden Rules of UI Design. Article on Adobe's Xd Ideas website. Accessed on 17.10.2021. <https://xd.adobe.com/ideas/process/ui-design/4-golden-rules-ui-design/>

Choose a Database: Cloud Firestore or Realtime Database. N.d. Help article in Firebase documentation. Accessed on 3.10.2021. [https://firebase.google.com/docs/database/rtdb-vs-firestore#what are some other important things to consider](https://firebase.google.com/docs/database/rtdb-vs-firestore#what_are_some_other_important_things_to_consider)

Cleanslate. Github page. Accessed on 16.10.2021. <https://github.com/premasagar/cleanslate>

Cloud infrastructure services vendor market share worldwide from 4th quarter 2017 to 3rd quarter 2021. Research report on Statista's website. Accessed on 13.11.2021. <https://www.statista.com/statistics/967365/worldwide-cloud-infrastructure-services-market-share-vendor/>

Control Access with Custom Claims and Security Rules. N.d. Help article in Firebase documentation. Accessed on 16.10.2021. <https://firebase.google.com/docs/auth/admin/custom-claims>

Gillis, S., Lewis, S. 2021. Object-oriented programming (OOP). Article on TechTarget. Accessed on 13.11.2021. <https://searcharchitecture.techtarget.com/definition/object-oriented-programming-OOP>

Harvey, C. 14.8.2021. AWS vs. Azure vs. Google Cloud: 2021 Cloud Platform Comparison. Article on Datamation. Accessed on 16.10.2021. <https://www.datamation.com/cloud/aws-vs-azure-vs-google-cloud/>

Kerpelman, T. 3.10.2017. Cloud Firestore vs the Realtime Database: Which one do I use? Article on The Firebase Blog. Accessed on 24.11.2021. <https://firebase.googleblog.com/2017/10/cloud-firestore-for-rtdb-developers.html>

Opidi, A. 12.8.2020. NPM vs. Yarn: Which Package Manager Should You Choose? Article on WhiteSource's blog. Accessed on 17.10.2021. <https://www.whitesourcesoftware.com/free-developer-tools/blog/npm-vs-yarn-which-should-you-choose/>

Primary and foreign keys. N.d. Topic on IBM's InfoSphere Data Architect website. Accessed on 16.10.2021. <https://www.ibm.com/docs/en/ida/9.1?topic=entities-primary-foreign-keys>

What is a Database Index? N.d. Article on Codecademy.com. Accessed on 16.10.2021. <https://www.codecademy.com/articles/sql-indexes>

What is cloud computing? A beginner's guide. N.d. Introduction page to cloud computing on Microsoft Azure's website. Accessed on 17.10.2021. <https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/>

What is NPS? Your ultimate guide to Net Promoter Score. Article on Qualtrics' website. Accessed on 3.10.2021. <https://www.qualtrics.com/experience-management/customer/net-promoter-score/>

Ylisiurunen, L. 1.3.2021. How does knowledge-based management enhance the decision-making process? Article on Gallant's website. Accessed on 17.10.2021. <https://gallant.fi/en/how-does-knowledge-based-management-enhance-the-decision-making-process>