



JACOBS
UNIVERSITY

Picture Collage & Time-Lapse Web Application

by

Esti Sojati

Bachelor Thesis in Computer Science

Submission: December 4, 2020
Prof. Francesco Maurelli & Conrad Zeidler

Supervisors:

Jacobs University Bremen | Department of Computer Science and Electrical Engineering

Statutory Declaration

Family Name, Given / First Name	Sojati, Esti
Matriculation number	30000548
What kind of thesis are you submitting: Bachelor-, Master-, PhD-Thesis	Bachelor-Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

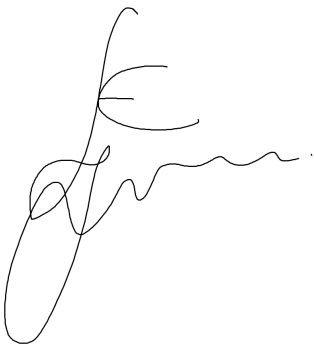
This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

December 4, 2020



Abstract

Growing fresh food in space or other extra-terrestrial environments is hard thing to do, but really essential. Fresh food provides essential vitamins, minerals and other useful macromolecules such as bio-active compounds to support crew health, and thereby functions as a countermeasure for the stresses associated with deep space exploration. Considering the symbiotic relationship between humans (carbon dioxide emitters) and plants (carbon dioxide absorbers), plant growth modules will also provide valuable oxygen to the crew and remove harmful carbon dioxide. EDEN ISS of DLR Bremen is an initiative that has been running since 2011, focused on Bio-regenerative Life Support Systems, especially greenhouse modules, and how these technologies can be integrated in future space habitats. The EDEN research group operates an experimental greenhouse facility near the polar research station Neumayer III of the Alfred Wegener Institute in Antarctica as part of the EDEN ISS project. In a semi-enclosed greenhouse system, innovative technologies and processes for the long-term stay of humans in space are tested in so-called analogue missions under Martian/Moon-like conditions regarding harsh environments, logistics and crew situations.[17] The container-sized greenhouse of the EDEN ISS project will provide year-round fresh food supplementation for the Neumayer Station III crew.[18] There are numerous cameras placed inside the greenhouse, whose main purpose is watching over the plant growth, but also other compartments that are within their frame. This thesis is about the making of an application that provides the DLR EDEN ISS team with a better observation solution of the plants and their changes through these cameras.

Contents

1	Introduction	1
2	Requirements	1
3	Website Design	2
3.1	Top Bar Menu	2
3.2	Time-Lapse	3
3.3	Image Slideshow	4
4	Implementation	6
4.1	Image Retrieval	7
4.2	Layout Custom Components	8
4.2.1	Container	8
4.2.2	Column	9
4.2.3	Row	9
4.2.4	Cell	9
4.3	Layout Creation	10
4.4	Image Movement	12
4.5	Image Slideshow	13
4.5.1	What Was Removed	14
4.5.2	What Was Added	15
4.6	Top Bar Menu	16
4.6.1	HOME	16
4.6.2	Layout Menu	17
4.6.3	Timer Text-Field	17
4.6.4	Filters Menu	18
4.6.5	TIMELAPSE	19
4.7	Time-Lapse	20
4.8	Control File	21
4.9	Automation	23
4.9.1	React Client Windows Application	23
4.9.2	HTTP Server Windows Application	24
4.9.3	Python Automation Script	25
5	All Layouts	25
5.1	Layout: 4x4	25
5.2	Layout: 3x3	26
5.3	Layout: 2x2	26
5.4	Layout: 1x1	27
5.5	Layout: 1+12	27
5.6	Layout: 2+8	28
5.7	Layout: 1+7	28
5.8	Layout: 1+11	29
5.9	Layout: 18+1	29
6	JavaScript + React.JS + Node.JS	30
6.1	What is JavaScript and its Uses	30
6.2	Reasons to Use JavaScript for Modern Web App Development in 2020	31

6.2.1	Minimize the Complexity of Web App Development Process	31
6.2.2	Ease of Writing Server-side Code in JavaScript	31
6.2.3	MEAN Stack: 4 Major Components in a Single Pack	32
6.2.4	Hassle-Free Integration of Multiple Transpilers	32
6.2.5	Ability To Develop Responsive Web Pages With JavaScript	32
6.2.6	A Broad Access of Libraries and Frameworks	32
6.3	Popular JS Frameworks That You Can Use For Web App Development	33
6.4	What makes React so fast	33
6.4.1	Virtual DOM	33
6.4.2	Diffing Algorithm	34
6.4.3	Single-way Data Flow	35
6.5	What makes Node so great	35
6.5.1	Single programming language	36
6.5.2	Large Community	36
6.5.3	API	36
6.5.4	Scalability	37
6.5.5	Real-time web applications	37
6.6	Conclusion	37
7	Material-UI vs Bootstrap	37
7.1	Bootstrap	37
7.1.1	Pros	38
7.1.2	Cons	38
7.2	Material Design	39
7.2.1	Pros	39
7.2.2	Cons	39
7.3	Conclusion	40
8	FTP Server vs HTTP Server	40
8.1	FTP	41
8.1.1	Pros	41
8.1.2	Cons	42
8.2	HTTP	44
8.2.1	Pros	44
8.2.2	Cons	46
8.3	Conclusion	47
8.4	Is There a Better Solution that Outperforms Both?	47
9	Problems during Development	47
9.1	Grid Component	47
9.2	Python Application	47
10	Conclusions	48

1 Introduction

With a huge amount of pictures to look into and analyze on a daily basis, it is sure that it would be a challenge for German Aerospace Center (DLR or Deutsches Zentrum für Luft- und Raumfahrt) EDEN ISS project to browse through every single one of them. An ideal image summary should contain as many informative regions as possible on a given space[19]. To make this process easier, more efficient, less painful and to make the pictures more observable, we needed to address this problem. Picture collage maker tools provide with one simple layout using direct code compilation[3] or a ready-to-use application that saves a picture created out of a collage of pictures[88]. However, the purpose of the application we will talk about is solely for observation, so we do not save, create, or add anything to the pictures as the information is already attached to each of them by default. We also need different layouts for an easier and versatile observation.

The application is web-based, which itself allows for a more versatile observation, not-dependent on the location of the person using it. There is no log-in authentication required and nor does one have to work at DLR to have access to it, so it is a picture-observation solution made simple for everyone who wants to use it.

DLR has a dedicated server containing information and data regarding the EDEN ISS project. The server provided by DLR is a File Transfer Protocol (FTP) server. Located inside it, is a complex folder structure, containing data related and not related to our picture observation application. The server has been there since the EDEN ISS team was founded and it is full of data, but we only need to access those relevant to us, i.e. images.

However, the FTP server was not fully compatible with our project programming languages and the nature of the server itself did not allow for a good solution of getting the pictures[2], so how could we access them and later server them to display in the Graphical User Interface (GUI)? Usually, all the data that can be served in a web-based application is provided from an HyperText Transfer Protocol (HTTP) server[67]. But how could we connect two servers (FTP and HTTP) with each other and get the useful information while also maintaining a good performance? Since our application is to be used by everyone, we do not need to create a authentication screen, which FTP requires[67] and for HTTP it is not mandatory[67]. How should we also handle this issue? Furthermore, when using an image from the FTP server, there exists no meta-data[9], just the raw binary, but HTTP server provides meta-data[9], which is useful to us when trying to retrieve the latest picture.

2 Requirements

DLR EDEN ISS team provided the project with a list of requirements with the following structure:

1. General Requirements:
 - Size of visualisation adaptable to screen size
 - Should be possible to put on website, i.e. web-application
 - Programming language up to me
2. Picture visualization task requirements:

- Take pictures from DLR FTP server
- Visualize pictures from last day as default
- X various layouts to choose ("button" to switch layouts)
- Picture should stay X seconds (would be good to set number in GUI) and move afterwards on screen to different location like slideshow or disappear in case not all pictures displayed
- Create automated timelapse with the last X pictures (choose picture + dedicated menu)
- If you click on picture in layouts, this picture should pop up as new layer over the whole screen and in background everything stays the same

Everything part of the thesis that is not mentioned in the requirements, is an extra part of this application.

3 Website Design

Coming up with a perfect design for your website is always difficult, all the more when one does not come from a design-related background. This section of the thesis thoroughly explains the design aspect of the web-application and how it came to be. Before starting the project, there was already a thought prototype of what it would look like. The most important design principle followed was minimalism. Ever since the beginning, minimalism is what shaped the way of thinking and the design of the web-page. The reason why minimalism was the main focus is because the website was meant to be simple, easy to follow, and not make the user tired when looking at it for a long time.

3.1 Top Bar Menu

During the first phase of the design prototype, it was thought to use a navigation menu. There are limitless options to display a bar menu these days, but it all comes down to the context of use. Since we wanted our application to be simple and intuitive, it was decided that a modal[37] component that could be shown and hidden with the use of a button or hovering feature was a no go. So, after deciding to be an always shown component, the question arises: Where should it be located on the screen? The two obvious options available are on the top or on the side of the screen. After further evaluation and investigation on the entire application design, it was best thought to be displayed on top. If it would be located on the side, then we would need a way to hide it so to not obstruct the observation and it would go back to just being a modal component again. On the top of the screen we could "hide" it by just scrolling down. One would say that you can also scroll horizontally with various ways, but scrolling vertically is always going to be easier, intuitive, and have less steps to do so.

It was also thought as best to use a simple, one-colored navigation bar component taking the entire width of the web-page in order for the design to be uniform. The color was decided to be a blue menu with white text to have a colorful look, but with a good contrast so it does not hurt the eye, e.x. when using yellow background and red text. All the buttons located inside the navigation bar that can change the view of the page in any way would have a white outlined border, whereas the buttons that would not do such thing would not

have the border. This was done to distinguish different types of buttons used and try to create an artificial intuition of the bar menu. Using this logic *HOME*, $\{\text{folders selected} / \text{total nr. of folders}\}$, *TIMELAPSE* buttons are outlined, and *Layout : {layout name}* are not.



Figure 1: Top Application Bar

3.2 Time-Lapse

It was decided that the *TIMELAPSE* button would be part of the top menu and not open a new website tab, because opening a completely new tab would increase the complexity of the project and require more effort connecting components, attributes, and states in the background. Furthermore, it is also required to open an additional server endpoint for the new page. But more importantly, our way is easier both in terms of accessibility and quickness. It would take too much time for the user to switch to different tabs and if there are many tabs opened, it is hard to keep count of what the actual web-page tabs contain. This way, it allows the user to have access to the whole features of the application even in full-screen mode, the mode our application is supposed to be used in.

When one goes to the time-lapse section, you can see that the blue top bar is still visible. However, all the buttons but *HOME* and *TIMELAPSE* are missing now, because they have no use in this part of the application.

There were 3 requirements thought for this particular time-lapse implementation:

- To be able to choose the folder/camera
- To be able to modify the speed of the time-lapse
- To be able to modify the number of pictures

Thus, there are three inputs located underneath the top bar that are part of time-lapse creation tool. Since everything below the blue top bar is a white space, it was thought to give these buttons an blue outline. The outline would help people to see how big are the buttons and what is the area of input, and the blue color to match with the overall theme of the application. Lastly, there is the *Create* button that was given the blue background and the white text to visually differentiate the functionality from the input fields and also to match the theme.

Since this section was not going to be used for observation purposes and only to create a quick time-lapse for presentation, the inputs and the button did not need to be integrated in the top bar. Furthermore, since there are two buttons in the top bar menu, *HOME* and *TIMELAPSE*, and the folder drop-down menu size is dynamically adapted to the folder name and length, it would be better to have as much horizontal space as possible for different window sizes. Under all the components, there is enough space for the Graphics Interchange Format (GIF) to be displayed in the center of the page, more specifically 1920×1080 .

Since one might experiment how the time-lapse looks and behaves with different inputs and picture numbers, it was thought not to automatically download the time-lapse, but

to let the user decide by themselves. If one wants to download it, it can do it save it as a normal internet image with the option: "Save image as...". If an automatic download would start save it locally in the user's machine after creation, then the user would need extra steps to play the file. Assuming the best case possible, the user needs an extra click once the file has been downloaded. Assuming the worst case possible, the user has a machine which does not have the right video player or the file is not supported by it. Then the user would need to download a proper video player adding more possible extra steps. With this method, the application is dependent on the user to function properly as it should. But in our case they can see the time-lapse in the web-page already.

The picture below shows the thought use case of the time-lapse, scrolling down to hide the top bar and to work with different inputs and still see most of the useful information in the GIF.

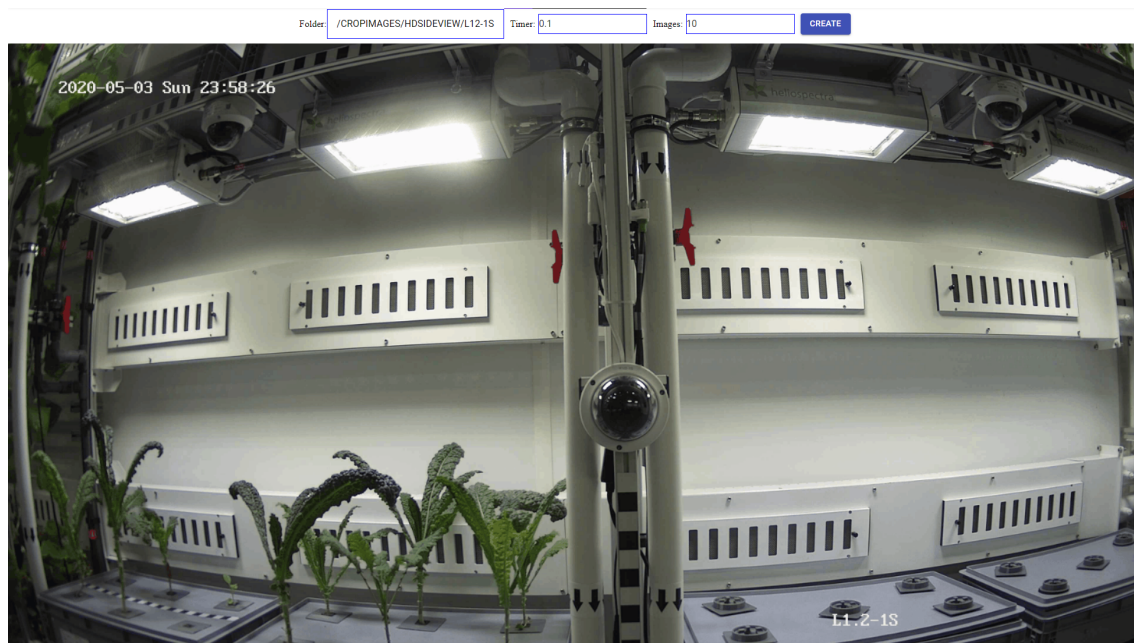


Figure 2: Time-Lapse Section

3.3 Image Slideshow

The image slideshow appears whenever you click a picture from the current layout. All the images that appear in the different layouts are the latest pictures retrieved from their particular folder nested in the HTTP server. Whenever you click a picture that is currently displayed, an image slideshow is opened that makes you see all the images in that specific folder that the picture was taken from and it occupies the entire screen (i.e. full-screen).

It has a minimalist look that displays the whole picture and a blue *X* on the top-right corner. The *X* was decided to be blue for consistency and usually, for modal parts of React applications, it is located on the top-left side, but this was changed with the customer's request. This button is always shown, overlapping the image, for easier access and faster close of the component.

Image slideshow also contains small arrows that spawn from top to bottom with a small

width that are used to control the flow direction of the slideshow. The arrows are always shown and their clickable span only appears when you hover your mouse over to the corners of each image. This was done to make sure to get rid small "nuisances" when trying to look at the picture, and also because it is pretty intuitive for them to be on the side. When the slideshow opens for the first time, it shows you the image (latest one) that you currently clicked. All the other images are located on the left side to make it as intuitive as possible so you have to use the left-side arrow in the beginning.

Besides the virtual arrows located on the screen, it was made possible so the user could also use the keyboard physical arrow keys. This way it give the user more options to use the image slideshow depending on their preference. One would think that since the virtual arrows need to be hovered with the mouse to appear and disappear and hence makes the use of the mouse slower than the physical keys. However, this is not the case with this special component. Once you hover around the area of the virtual arrows, if you do not move the mouse away, they stay displayed and you can click them as fast as you can. The real reason why the keyboard physical arrows are faster is that it is easier to just click them as fast as possible, which you could also do with the mouse, but it would be harder and inconvenient as the mouse would be hard to keep in a fixed position and occasionally move from the virtual arrows.

The slideshow is stripped off of any animations at all, to make the transition phase as fast as possible and the user can pass scroll past a lot of images in a shorter time span using the two options stated in the paragraph above.

Shown in the pictures below are the visual differences between the slideshow stock component and the final modified component.

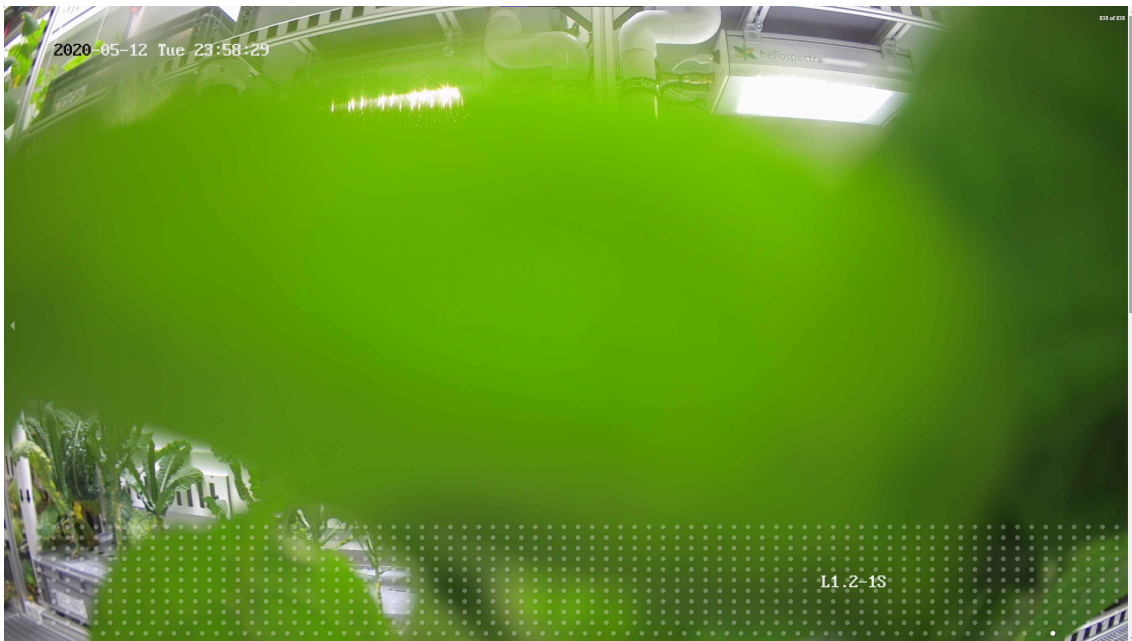


Figure 3: Slideshow Stock Component: Scrolled Up

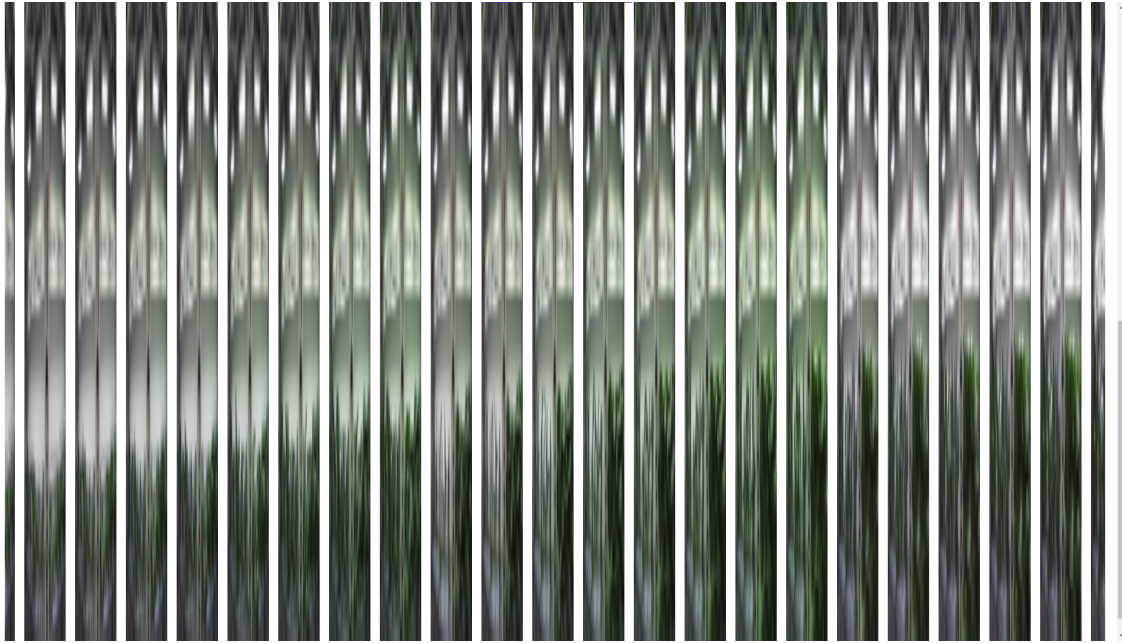


Figure 4: Slideshow Stock Component: Scrolled Down

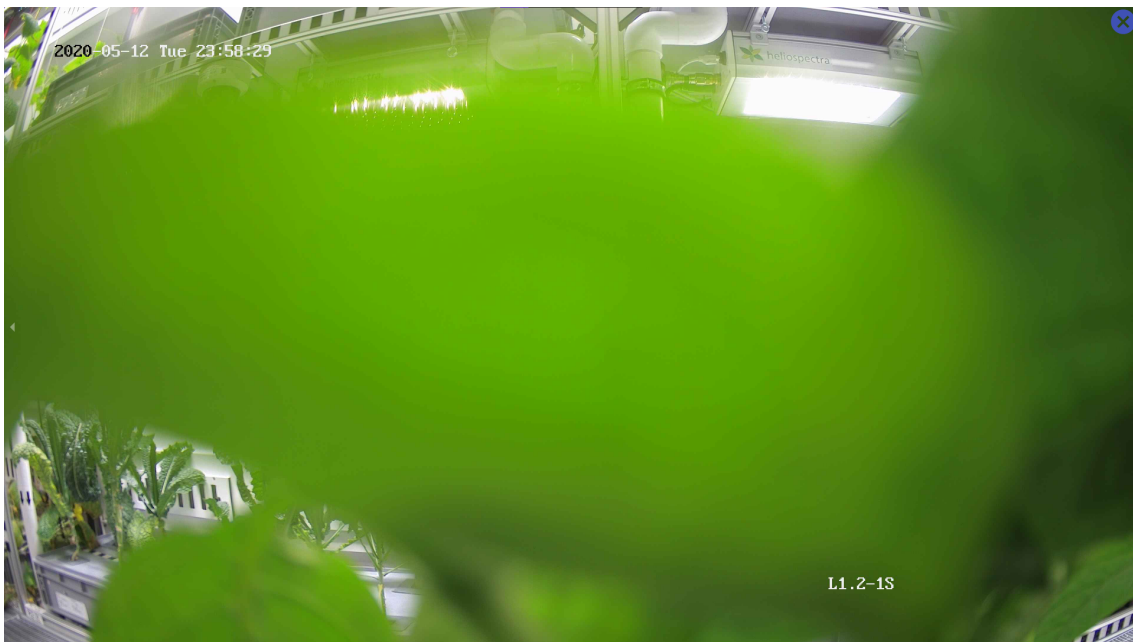


Figure 5: Slideshow Final Component

4 Implementation

Implementing a project, in a programming language that I have never worked before and from scratch, is not easy. There was no other already-built application close or similar to mine which I could use as a basis and further develop it. Every other application was very

specific to their own requirements as this application was very specific to the DLR EDEN ISS team. In this section I am going to explain how the logic in my application works and how everything is connected together.

4.1 Image Retrieval

One of the biggest challenges of this project was to come up with a way to find the latest images needed for display. Having to do with a complex folder structure and hundreds of pictures inside them whilst still having a decent performance, complicated things. There are many ways to do this, but it all comes down to being efficient and coming up with a good solution that is relatively fast for this kind of task.

It is certain that one of the slowest ways to get all the single right pictures in this kind of structure would be with the use of loops. A recursive way of accessing the folders and the pictures would outperform the loop way in terms of performance any day of the week. In our case, we use a recursive way with the help of an inside *forEach()* loop function.

First of all, we decided to use *express*[48], since it is the most commonly used framework for developing Node.js and helps in fast-tracking development of server-based applications[63]. Since the customer's FTP server is on local storage and we are working with pictures, which get automatically updated also in local storage, we can make use of a middleware[57] with the function *static()*[49], part of *express*, to provide them[56].

This middleware only executes second in the middleware stack. We have yet another one which executes for every request that comes. This middleware handles all the possible missing headers to make the React client connect to the server from any port and make different requests. With each request to the server, we attach a header to the response by using *res.header()*, which is an alias of *res.set()*[50]. After this middleware is executed, we make sure to execute the second middleware by providing the *next()* function at the end of it and the order of the functions in the code matter, too[58].

We bind and listen for connections on a specific host and port, which in our case is 5000, and we do that using *listen()*[51] function and print a sentence letting the user know about which port is occupied with *console.log()*[47].

We have 2 endpoints in our server, one to the default route, i.e. `"/`, and the other one is a route to path `"/folder`". The first endpoint's functionality is to provide a JavaScript Object Notation (JSON)[16] object with all the latest images and their specific folder. The second endpoint's functionality is to provide all the images of a certain folder input.

Since all the images located in the local FTP server have a certain name pattern, concretely `"HDCAM_{name of camera}_{date of creation}_{time of arrival}"`. The name of the camera is also the name of each folder. In order to be able to find the latest picture we set a variable that holds `"{year}{month}{day}"`. The variables are set using the object *Date()*[77] provided by JavaScript. We always get the date one day before the current day, as our application restarts at 06:00:00 and usually the pictures are uploaded to the server at 23:58. After getting the right date, we use a modification of a function provided open-source called *walkSync()*[80].

The function starts at the top of the directory, provided at its input, and checks all the contents of that directory with a *forEach()* function. The directory is read using *readdirSync()*[53]. Every content is checked whether it is a directory or whether it is not using *isDirectory()*[54]. In case the content is a directory, then we call the *walkSync()* function again, with

the new updated path to "{top directory} + / + {content that is directory}". Automatically it scans for every content in this new directory. For every content that is not a directory, we check if it contains our specified date using `includes()`[79] function, and the file ends with ".jpeg", with the `endsWith()`[78] function. If a match has been found, then we push an object using `push()`[14] containing this image and the folder it was found from: "image : {image file name}" and "folder : {current directory}". After, we remove a certain number of characters depending on the path length using `substr()`[15] to `filelist`, which will be our `return` value at the end of the function. At the end of the GET request of our root path endpoint, we return a response using `walkSync({path to images})` that we format using `json()`[52] function.

The other endpoint to path `/folder` uses a more simplified version of the `walkSync()`. We do not make use of the `date` variable here, and we don't use the function recursively. In this case, for the directory specified in the request, we scan every content if it is a file, using `isFile()`[55] function, and if that content ends with ".jpeg". If a content like that has been found, we push it, after removing a certain number of characters depending on the path length, to our array `filelist` that we return at the end of the function. We send the array back with a request by sorting and formatting it to JSON notation.

4.2 Layout Custom Components

As mentioned before, the layouts were not based on any ready-to-use components provided by Material-UI or Bootstrap. They are created by making use of the different components defined as:

- `< Container / >`
- `< Column / >`
- `< Row / >`
- `< Cell / >`

They are located in separate files and always ready to use. I will explain each of them down below.

4.2.1 Container

The use of the `< Container / >` component was to transform the entire screen (without including the top bar) into a usable `< div / >`[11]. The point in doing this was to make sure that the screen could be worked on by the other components used inside it. This page-created `< div / >` contained a simple style comprised by a `height` attribute consisting of the value `100vh` (100% of `< div / >` vertical height), and a `display` attribute consisting of the value `flex`. As an input, the `< Container / >` component has the entire application `props`. Inside the `< div / >`, we insert `props.children` as they can be passed down and used by the `< Cell / >` component where the images are made sure to be rendered. The most important part of the layout creation is this particular `flex` parameter that allowed us for an easier way to work with the other components. It enables a `flex`-context for all its direct children[10]. The `< Container / >` is an arrow function that returns the `< div / >`, which we export to use in other files as a component.

4.2.2 Column

After the parent, `< Container/ >`, made possible to inherit the `flex` attribute and also made the screen into a giant `< div/ >`, we always use the `< Column/ >` component as the next child. The `< Column/ >` component also consists of a simple style where we store 3 attributes: a `flex` variable, `display : "flex"`, and lastly `flexDirection : "column"`. The last attribute is the most important attribute, from which we have named our component from, and makes sure that the multiple `< div/ >` created by this component follow the column pattern style, i.e. top to bottom. As input it takes an object comprised by an integer, which accounts for the flex attribute inside the style, and the `children` of the `props`, passed down by the `< Container/ >`. The `< Column/ >` component is also an arrow function that returns the `< div/ >`, which we later use in other files.

4.2.3 Row

The `< Row/ >` component is almost identical to the `< Column/ >` component, but with a small difference. It also creates a simple `< div/ >` with a simple style. The style contains 3 variables: a `flex` attribute which is set dynamically from the input, `display : "flex"`, and `flexDirection : "row"`. The last variable is the one that defines the name of the component and is the most crucial one for it. It enables the different `< Row/ >` components to follow a pattern with the logical direction of a row, i.e. items are arranged in a left-to-right manner. As input, it takes `flex`, a variable which is used to determine the `style` attribute as mentioned above, and `children`, a variable which comes from a parent and is stored inside the `< div/ >` created. The `< Row/ >` component, just like the other components so far, is also an arrow function that returns a `< div/ >` usable in other files.

4.2.4 Cell

The `< Cell/ >` component takes the name from its particular functionality as well and is fundamentally different from all three above components. Its purpose is a container for image storing and slide-show opener. It has 3 inputs: `url`, `onClick`, and `folder`. As an output it creates an even simpler `< div/ >` with nothing inside it. It only has a `style` that defines the `< div/ >` and the React event handler called `onClick()`. First, we initialize the `style` with `flex : 1` and `border : "1px solid white"` attributes. The reason we always attach a static value to `flex` is that the `< Cell/ >` is the very last component, inside other nested ones (containing `< Container/ >`, `< Column/ >`, and `< Row/ >` used for the layout creations), and it is always used alone with no other components alongside it. The reason we attach a `border` attribute is to differentiate the layout images from each other and create that white-grid when appeared in a layout. To make the pictures "appear" in the cell and be rendered in the front-end part, we set the value of the `backgroundImage` style attribute to the particular picture. This attribute uses the `url` variable from the input, which is the path of the image that comes from the back-end (our HTTP server). After setting `backgroundImage`, we also set the `backgroundSize` to "100% 100%", i.e. we force the image to fill 100% of the `< div/ >` container top to bottom, and also 100% of the `< div/ >` from left to right. With this last one, we are done with the style. The `onClick()` event handler is an arrow function that takes as input the `folder` variable that comes from the `< Cell/ >` input, too, and makes sure that when each image is clicked in the front-end it opens a slide-show of images of that particular folder. This component is also exported for uses outside the file.

4.3 Layout Creation

In the previous subsection we introduced our custom-made components consisting of `< Container/ >`, `< Column/ >`, `< Row/ >`, and `< Cell/ >`. In this subsection we are going to show an example on how to create a layout using those components and their properties. Our example is going to consist on how our default layout is created.

Our application's default layout arranges the pictures in a 4×4 manner. Before everything else, we make sure to wrap the available working space under the top bar menu in a `< div/ >` created by the `< Container/ >` component.

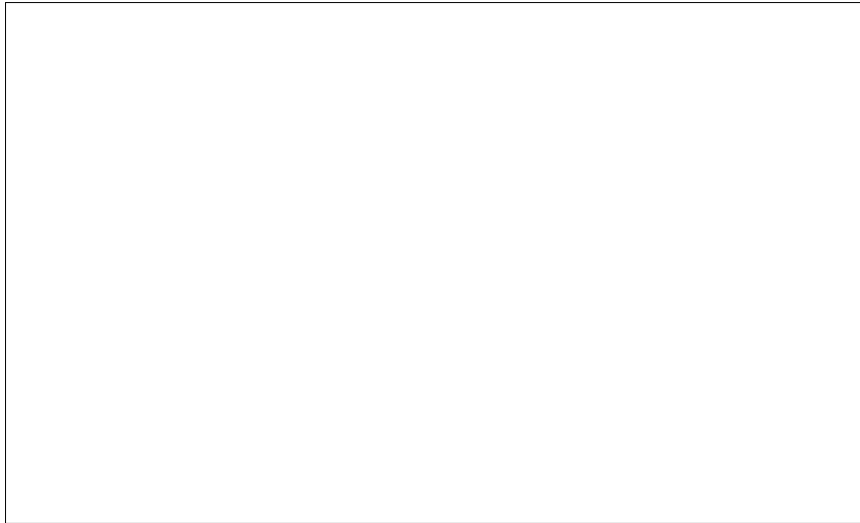


Figure 6: Container (and also Column) Wrapping

This makes sure that every other `< div/ >` being created inside this workable space inherits the `flex` property from their parent (i.e. `< Container/ >`).

Then, we work our way through using the properties of the `< Column/ >` and `< Row/ >`. As stated before, we always use `< Column/ >` component into the space wrapped by `< Container/ >` (the picture representation would look the same as Fig.6, because they overlap each other and take the same amount of space). This allows us to make sure that the other `< div/ >` created inside it get organized in a column-way, i.e. top-to-bottom.

Knowing this, we build 4 different `< div/ >` rows with the same size using the `< Row/ >` component and that size is adjusted by the different `flex` values that we give it.

Figure 7: 4 Rows Created Inside One Column / Container

In our particular case all the rows created here have a *flex* value of 1, i.e. they all take the same space in regard to one another. Until this point, we have divided our screen into 4 equal rows of `< div/ >` and what is missing is to divide these rows with other equal `< div/ >` to be able to achieve an equal distribution of the 4x4 layout.

To do so, we make use of the `< Column/ >` component once again. Inside each row created, we start to divide the space into 4 different blocks (`< div/ >`). Our `< Row/ >` component made sure its children `< div/ >` nested inside them follow the left-to-right manner rule. Using this property, we create 4 different `< div/ >` provided by the `< Column/ >` component.

Figure 8: 4 Columns Created Inside Each Single Row (each block having a Cell inside)

These also have to make sure that are spread equally inside each row, so we once again use the *flex* attribute to our advantage, with each `< Column/ >` having a *flex* value of 1.

For every `< Row/ >` and `< Column/ >` that we have created, since all their *flex* attribute values are always 1 and there are 4 instances of each (rows and columns components), $1 + 1 + 1 + 1 = 4$, so that means each of them takes $1 / 4$ of the space[29] and ultimately we have the perfect 4×4 grid layout. Now what is left is filling the missing content with the pictures that come from the back-end. For that we use our `< Cell/ >` component, made specifically for this purpose. Inside each grid block we create a `< div/ >` of `< Cell/ >`, meaning a total of 16 `< div/ >` and our layout is complete and ready for observation. With the `< Cell/ >` insertion the layout would look the same as in Fig.8.

Below is a representation of how flexbox divides the spaces of the div:

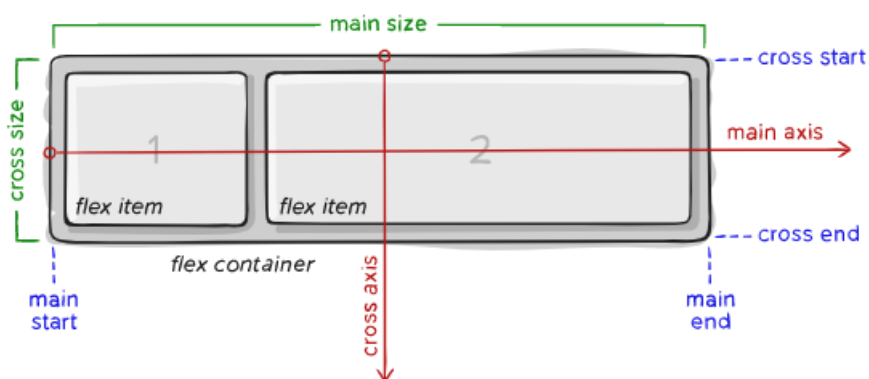


Figure 9: FlexBox Layout Specification[10]

4.4 Image Movement

One of the most important parts of the application is the picture movement. In order to make that work, we decided the program would be fastest using simple, but very powerful, array manipulation. I.e. using the JSON notation, which we used for picture information storing and organization, together with an array variable called *newData*, which contains a fixed number of elements different from layout to layout. In the beginning of every layout file we use a separate function to store the images from the inner *props* to the current *state*[42] of our application. This allows to store the pictures that need to be displayed and also be accessed from other functions inside the files. The number of pictures saved in the *state* differ from layout to layout, depending on how many can fit in each one. Then, I bind the timer from the GUI to the React life-cycle method called *componentDidMount()*[39]. In our case, this life-cycle method is called repeatedly and ensures that the pictures that will get displayed in the layout, change, and get shifted before they get rendered. In order to do this, every iteration of *componentDidMount()*, we use the *newData* variable to store 1 more picture than the maximum allowed number of pictures in each layout from the *props*. Then I shift *newData*, which removes the first element, and leaves it with the accepted number of images, and ultimately update the *state* of the application with this newly-shifted array using *setState()*[42]. However, this needs to create a loop that only uses the pictures from the *props*, and be dynamic, so that if another folder is added, it should still work as designed. We control this with another variable called *endIndex* which only increments as high as the number of images stored

in *props*. And when it does go until the limit of the *props*, we reset it to 0, i.e. the start index of *props* again. This process gets repeated indefinitely and creates a loop in itself without the need of a normal loop, but by making use of the application life-cycle methods of React.

However, *componentDidMount()* cannot, and does not handle everything. Another life-cycle function called *componentDidUpdate()*[40] is needed for the job. With this built-in function we handle the changes in our application. Meaning, every time the end user increments or decrements the timer, or somehow the pictures have retrieved from the folders have changed, this function comes to play. What it does is that it takes as an input the previous *props* and the previous *state* of the application and compares them to the current ones. It gets called repeatedly, like *componentDidMount()*, but in case there are no changes, it does nothing. And when there changes, it handles and updates them accordingly.

We also make use of another React life-cycle method called *componentWillUnmount()* [41] that handles things when the application is stopped. Our block of code that is bound to an interval will keep executing until it is stopped. We make sure that when stopping/exiting the application we clear the interval to make it stop, otherwise it will run in the background until the memory is consumed. It is really important that we shut down things correctly when the program is supposed to be turned off so we have no leftovers.

Below is a diagram of how React rendering and life cycles work:

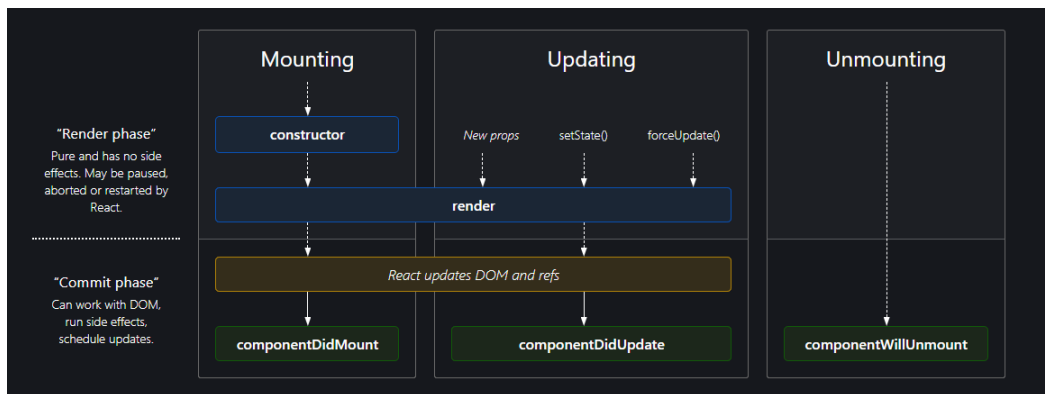


Figure 10: React life-cycle methods diagram[70]

4.5 Image Slideshow

For the creation of our image slideshow we used a combination two powerful components: `< Dialog / >`[23] and `< Carousel / >`[69]. Out of these two, the most important one is the latter. It is what the user sees and interacts with when image slideshow is opened. In the website design section it was said that we required the cleanest possible UI (user interface), so that the user could see the images without obstacles on the way. The component itself comes with a lot of options already provided and a lot more to modify and to insert. In our case, we needed to remove these already-provided options, modify the animations, and add some functionality parameters to `< Carousel / >`.

4.5.1 What Was Removed

We removed, what I would say was the most obstructing parameter of all of them, the indicators. The indicators are a dynamic parameter that overlap the image itself trying to give you a visualization with little transparent dots of how many images the slideshow has and which one you are currently seeing. In very different cases they can just take a small horizontal line of space to occupying the whole screen. In our case, they take a significantly big block of image space nearly half of the screen, considering how many pictures are inside each folder. So, to remove them, we used a component built-in command: `showIndicators = {false}`.

Another thing that was removed is a little text, shown on the top-right of the application, that used to display "`{current image number}` of `{total number of images}`" of the current opened image slideshow. It is a significantly small text, but because in that top-right corner we have located our blue *X* button, it was best decided to remove it, since it had no real functionality after the button overlap. In order to achieve this, we did: `showStatus = {false}`.

The other thing that was removed are the thumbnails. These were small presentations of the following images that were located on the far bottom side of page. They were regarded as inconvenient and a waste of space for the following reasons:

- Considering our images' resolution, they would be far too small and hard to tell, only having the color as a differentiation factor.
- They are supposed to be used in a case with not a lot of images in the slideshow to make it easier to fit them in a straight horizontal line, otherwise they would be hidden from sight.
- From their stock behavior, we noticed that the thumbnails that could be displayed was dependent on the screen width, thus making the use of them inconsistent.
- Since our image slideshow is supposed to cover the whole screen for a better observation of each image, and the thumbnails appeared in the bottom side, a scroll bar appeared whenever we enabled them, making the image in word slightly smaller, so not the best observation efficiency.

Thus, the image thumbnails under the slideshow were removed by using the following command: `showThumbs = {false}`.

The following modification takes part in the animation area. The developer made sure that his slideshow could have different transitions times made for everyone. By default, the images have a significant transition time that in our case appears as a black screen being filled by the image coming from left or right direction. This makes it hard to focus and tires the mind easily with the moving images. So, we completely removed it, so as not to have a transition time between them and each image just appears on top of the last one after the user switches. This also creates a sort of a time-lapse illusion if you switch them fast enough and you could see even the small changes that were made to every plant or any other area of the picture. This was made possible by modifying the transition time: `transitionTime = {0}`.

4.5.2 What Was Added

As discussed in the website design, the physical keyboard arrows are significantly faster than the virtual ones, so the first and last thing ever added to the `< Carousel / >` image slideshow component was the keyboard functionality. It makes possible to interact with the component with the keyboard keys and the command for what is `useKeyboardArrows = {true}`.

No other functionality- or animation-related parameter was added to `< Carousel / >` as discussed in the different paragraphs above. However, other functionalities were added to the image slideshow custom component. Most of them handle the opening of the slideshow, the closing, the rendering, and the error handling. They are called `renderSlider()`, `renderLoading()`, `renderClose()`, and `renderError()` accordingly. What we have talked about up until in this section has had everything to do with the function `renderSlider()`.

`renderError()` displays a text that notifies the user that something went wrong, concretely *"There is an error please try again later!"*. This error message is displayed when the image slideshow is opened and suddenly an unpredictable error occurs. All of the times, the error is related to the images coming from the back-end, so also in all cases, the user should restart the server. The message was rendered using `< Typography / >`[\[45\]](#) component from Material-UI.

`renderLoading()` displays a white screen with a big circular progress animation until the image slideshow is ready to open. It was decided to also be blue for consistency and it is created using `< CircularProgress / >`[\[38\]](#) component from Material-UI.

`renderClose()` is the function that displays the circular *X* button on the top-right side of the screen. Certain style modifications were made to assure that the button is located properly where it is. They are part of a `< div / >` that holds the style stated below:

- `position : "absolute"`
- `top : 0`
- `right : 0`
- `zIndex : 10`
- `width : "50px"`
- `height : "50px"`
- `borderRadius : "50px"`

Regarding the button itself, Material-UI provided `< CancelIcon / >`[\[34\]](#) with the proper functionality for it. Since the `< div / >` acts as its parent, it is forced to take the same style, and we made sure it fills the `< div / >` with `width : "100%"`, `height : "100%"`, and `color : primary` (i.e. blue). Apart from that, we have to make sure that it actually has a functionality, so we added an event handler `onClick()` that closes the image slideshow and thus take the user to the *HOME* screen.

Now we can explain the `< Dialog / >` component that we mentioned in the beginning of the section. It has a simple use and that is works as a parent `< div / >` for everything that we have talked about until now. It has as a parameter two booleans called `fullscreen`, and `open` that control the size of the dialog and whether it is going to show or not. It

also has an event handler called *onClose()* that handles the closing of the slideshow and updating the state in a correct way.

open boolean, *loading* boolean, *error* boolean, *onClose()* event handler, and all the pictures variables that are used in "SlideShow.js" are taken from the application *props* and are controlled by "App.js".

4.6 Top Bar Menu

Another important aspect of the application is the blue side bar located at the top. Its purpose is a container to hold other small components needed for the functionality of the application. The top bar in itself is a ready-to-use component `< AppBar / >` taken from Material-UI. This component however is not used as-is, but of course has certain modifications necessary to be integrated in our system. The most important modification is its position regarding the other components. Since the application's purpose is fulfilled best when you only see the images with no other distraction, it was thought that a possible solution would be to hide and show the top bar when needed to. The default behavior of the top bar as provided by Material-UI is to always show it when you scroll up and down. That is a really nice effect when you want easy access to the side bar and you don't need to scroll all the way to the top to access it. In our context, since there is not much scrolling to be made we had to modify its behavior. And to do this we change the value of *position* attribute from *fixed* to *static*. The color of the top bar is blue, by default as well. It did not need to change since when integrated with a white text from the other components, it makes sure that they are visible, and is easier to focus on it.

The next component is also taken directly from Material-UI and is used as-is, which is called `< Toolbar / >`[\[44\]](#). This is also spawned throughout the whole top bar as another `< div / >`, child of `< AppBar / >`. The reason for this is that, according to their documentation, it is easier to use other components inside it like `< Button / >`[\[21\]](#), `< IconButton / >`[\[28\]](#), and `< Typography / >`[\[44\]](#). We needed to create three buttons: *HOME*, *FOLDERS SELECTED*, and *TIMELAPSE*, using `< Button / >` component from Material-UI. These buttons have the color inherited from their parent (`< AppBar / >`) and are outlined for an easier and visible clickable area. For design uniformity it was thought that the buttons all have the same style.

The top bar is comprised of:

- *HOME*
- *Layout* : `{layoutname}` (Layout Menu)
- *Timer* : `{timerinput}` (Timer Text-Field)
- `{nr. of selected folders} / {nr. of total folders discovered} FOLDERS SELECTED` (Filters Menu)
- *TIMELAPSE*

We will be explaining each of them in detail in their own section down below.

4.6.1 HOME

HOME is just a button with a pretty straight-forward function and that is taking you to the image observation section. This button does nothing when you already are in the

home screen of the application, but its functionality comes to life when the user is in the Time-Lapse section. It changes the state of our application from *timelapse* to *home* with the command `setState({page : "home"})`.

4.6.2 Layout Menu

Layout : `{layoutname}` is a custom component called `< LayoutMenu / >` that enables you to switch between different layouts. It is a drop-down menu consisting of 9 different types of grid layouts to choose from. It utilizes ready-to-use components from Material-UI, nested together to create the drop-down. Our type of menu is what is called a "selected menu", i.e. a menu that is used for item selection and displays that item in the text area when the menu is closed [35].

The file containing the layout menu is an *const* arrow function. The input of it is an object consisting of *options*, *selectedIndex*, *setSelectedIndex*, and *showBorder*. *options* is the list of all the layout names ready to display in the hidden menu. *selectedIndex* is an integer that controls which layout is currently selected, *setSelectedIndex* is an event handler which sets the new index, i.e. sets the new selected layout to be displayed, and *showBorder* is a variable used to create a *1px solid blue* border by our request, which we use in section 4.7.

The "Layout: {layout name}" that the user sees is a `< div / >` that has the functionality of a button, but it is not a button or declared as a button. What I mean by that is that it is created by using the `< ListItem / >`[30] component from Material-UI, which itself has a property called *button*. We enable set this attribute to *true* for its particular type of behavior. The text that is shown inside this `< ListItem / >` is also a component from Material-UI called `< ListItemText / >`[31]. We dynamically store the appropriate layout text inside this component's props from `options[selectedIndex]`, and allows us to show the selected name in the GUI.

We then have a menu, that by default is hidden from sight, until you click the `< ListItem / >` mentioned above. It is created with `< Menu / >`[32] component from Material-UI where we have listed the layout options coming from the input. We make use of the `map()`[33] function from JavaScript to map every layout option with an index, which is individually different, and create the list of layouts to display dynamically. Every layout option besides the according index, also has a specific key that React requires to each of their names, and an event handler that traces back to the *setSelectedIndex* one.

In this arrow function we also have separate functions that handle the the opening of the hidden menu, the click of a certain item in the menu, and the close of the menu. We export it as `< LayoutMenu / >` in the end to use it as a component.

4.6.3 Timer Text-Field

Timer : `{timer input}` is a text-field that regulates the speed of the image shifting. For it, we use `< TextField / >`[43] from Material-UI. There were a lot of different text inputs that they provided and our specific one is the *filled-number* input, which we have to specify to the component's *id* prop. However, with this prop modification, we are still not sure than the user will not input any other random character from the keyboard. To be sure that that does not happen, we force the text field to only take a number as an input, giving the vale *number* to the *type* prop.

After making sure that it is the right text input type, we have to make sure it is properly integrated in our application style. As default, it comes with an underline under the input (i.e. the number), and since the main concept of design is minimalism, we decided to remove it with the props command `InputProps = {{disableUnderline : true}}`. We also modify its style to take a fixed position after the "Timer: " text, with `marginLeft : "5px"` and `padding : "0"`.

We always make sure that the number value that is displayed in the text input field comes from the *state* of the application, concretely with the component's props command `value = {'${this.state.timer}'}`. This timer has a very important key role, because it controls the flow of the images and that is why we have control over it using the *state*, where all the layout files access it from for the image movement.

The timer has also an event handler called `onChange()`, which is an arrow function. This arrow function has as input an *event*, where we take the newly changed value from the user's input. Inside this function we have some checks that make sure that the number provided by the user is appropriate, otherwise it would lead to an error or a bug. We have handled the case of the user's input of other keyboard characters with `type : "number"` as explained above, as that would make the timer unusable in the image movement shift speed. Another case that makes our timer unusable is when there is no timer at all. So if this ever happens, we make sure to give it a static value, which is also the default one, which is 5 (seconds). However, there is yet another case that would make the timer unusable and that is if the user tries to input a number smaller or equal to zero. In this case, we handle the timer to have a value of 1. Otherwise, in any other case, we override the current value of the timer with the one that the user desires. The range of numbers it is allowed to take is from 1 to 1e+21. Since the timer is located inside the *state* of React, we always override it using the built-in function `setState()`.

This timer field has its own function called `renderTextInput()`. It is located inside "App.js" and it only gets properly rendered if we are in our *HOME* page, which is also controlled by the *state* of the application.

4.6.4 Filters Menu

`{nr. of folders selected} / {nr. of total folders discovered} FOLDERS SELECTED` that the user sees in the GUI is created from a component located in a file named "Filters-Menu.js". The component is custom-made, but it uses other small components provided by Material-UI. It is a hidden modal, like the `< SlideShow / >`, and opens up a screen when the button that shows the folders is clicked.

In section 4.5, we saw the use of `< Dialog / >` component from Material-UI, and it is the same in `< FiltersMenu / >`. It also has the same *props* options as `< Dialog / >` in `< SlideShow / >`: `fullScreen`, `open`, and `onClose()`. Again, `fullScreen` is a boolean that enables the `< FiltersMenu / >` to occupy the entire screen and overlap the screen that you see before clicking the *Folders Selected* button, `open` is also a boolean which comes from the inner application *props* and checks if `< FiltersMenu / >` should be opened or not, and `onClose()` is an event handler that also comes from application *props* and handles the closing of the component. We have also rendered the exact same blue *X* button when the modal is open, with the same style as the one in `< SlideShow / >`.

What is new, is the dynamically displayed folder names that have a checkbox on the left side. They come from the application *props* and by default, all the checkboxes are

checked, thus displaying the latest picture of every folder discovered by the HTTP server. This part of the GUI was provided by `< Checkbox / >`[22] from Material-UI. This component provides built-in *props* that allow you to input a name associated with a checkbox, and it is called *name*, which we assigned dynamically to the name of each folder accordingly. It also has a *props* called *checked*: a boolean, where it gets the functionality from, which we also assign dynamically for every checkbox. For that we have a JSON called *filters* inside the *props*, which contains `{name of folder} : {true}`. By default, it is *true* for all of them, it changes from the event handler called *onChange*, and it is assigned to each `< Checkbox / >`.

`< Checkbox / >` only enables the animation and the functionality of a simple checkbox though, excluding the name from this process. However, since we assume that there are a lot of folders discovered by the back-end, we wanted to make the clicking process easier by making the name clickable, too. We solved this by wrapping the checkboxes and the names into a `< FormControlLabel / >`[25]. Since this component needs to have the functionality of `< CheckBox / >`, we enable that by *control* = `{< CheckBox / >}`, and inside it has everything we have talked about the `< CheckBox / >` so far. To make the whole name clickable we give *label* = `{foldername}` and we are done.

What we have explained until now, it is only true for one folder name with one checkbox and we don't know how many will be there in the future. To account for this, we dynamically create an array (called *row*) of arrays (called *column*) while pushing the folder names to create the rows and the columns of the `< FiltersMenu / >` component and assign their needed functionality. It is the same logic as when building a grid layout, but in this case we have a dynamic number of *row* items, with one *column* each. The *column* only contains a fixed number of items inside: 4. This way we display a grid with each row containing only four columns items of checkboxes and names at a time. In a bigger picture we have a `{dynamic nr. of rows} x 4` (row x column) layout. Then, with the help of *map()*, we map each of the row items that we have to a `< FormGroup / >`[26] that groups them together. Each `< FormControlLabel / >` component created has a *flex* value of 1 which enables an equal distribution of the items, as they are created inside the `< FormGroup / >`, one by one in a column-direction principle. The `< FormGroup / >` components that are created also have a *flex* value of 1, which enables the same equal distribution but in row-direction, and it is also reinforced by the *row* built-in boolean attribute. All of this is located inside a big `< div / >` wrapped up by `< FormControl / >`[24] that creates this type *fieldset*[12] we have talked about. We did not provide a *legend*[13] for our *fieldset* in word, because the button provides that intuitive purpose and minimalism. The component is exported to be used in "App.js".

4.6.5 TIMELAPSE

TIMELAPSE is another button that has the same exact functionality of *HOME* button. It takes you to the time-lapse section of the application whenever you are in the home screen. Same as *HOME* button, it enables this specific functionality while changing the state of the application with `setState({page : "timelapse"})`. When the user is in the time-lapse section, the button has no functionality.

4.7 Time-Lapse

The time lapse section of the application opens up when you click *TIMELAPSE* button located on the blue top bar. When this section first opens up, has a very simple and empty layout, with 3 options below the top bar: *Folder* : {*folder name*}, *Timer* : {*timer value*}, *Images* : {*images number*}, and below mainly a blank white screen.

`< Timelapse/ >` has a state object that get initialized with these default items and is used to control the functionality of the component:

- *folder* : "" (i.e. empty)
- *imageNumber* : 50
- *timer* : 0.1
- *selectedIndex* : 0
- *loading* : *false*
- *error* : *null*
- *src* : *null*

Folder : {*folder name*} is a drop-down menu created with the use of our custom `< LayoutMenu/ >` component that we have explained in section 4.6.2. It has almost the same design as the one in the application top bar, but with a *1px solid blue* border created by giving it the *showBorder* input. We assign a default selected index to the menu: the first folder that is found by our HTTP server, i.e. *selectedIndex = state.selectedIndex*. We also pass a function that is called *setSelectedIndex()* to the *props* to allow the application to get the newly selected folder from the user. Finally, we use *options* to pass down our items that are going to be displayed in the drop-down. With the help of the *map()* function we create an array only with the folder names. The folders are retrieved from our application *props*, i.e. the JSON containing all the images and their appropriate folder, gotten from the back-end. This drop-down menu has also a wrapping `< div/ >` with a certain style to make it in line with the other divs.

Next in line is the timer text field. It has been created by `< TextField/ >` component from Material-UI, the same way as explained in section 4.6.3, and it has the same style and *props* properties as the timer text field in the top bar. The only difference here is that the input is a float and not an integer. This means that the user is not as limited, to some extent. It can take numbers ranging from 0.1 to 1e+21 and it can hold up to 17 visible digits after 0, the space of the input field created by `< TextField/ >`. The default value is 0.1, always coming from the *state*, which we also set when the value of the user is negative, empty, or a text by using *setState()*. In other cases we update it using the same function, too. All of this is done by an in-line event handler called *onChange()*.

The last is the images number. This input field is created using `< TextField/ >`, too. It has exactly the same style and properties as the timer, but it has certain limitations. The input is integer and the default value is set to 2. Here, we also use an in-line event handler declaration for *onChange()*, which sets the values of the image numbers by changing the state of the component using *setState()*. The range of numbers here is from 2 to 80, because 80 is the maximum number of images it can take before the request times out, and we are going to explain later on why. So, for any value above 80, we automatically force the input to stay at 80. If the user's number is negative, empty, or a text, then we

force the input to be 2. In all other cases, we update it in the state with the user's input using `setState()`.

The blank space that the user initially sees is a `< div / >` that is dynamically adjusted to the window's size. This empty space is where the time-lapse will be displayed, i.e. rendered. The render function is called `renderGIF()` and it has 3 different functionalities: to render the loading component using `< CircularProgress / >`, to render the error message using `< Typography / >`, and lastly to render the GIF that is taken from this component's state.

Finally, after the user has chosen his preferred folder, the speed of the time-lapse, and the number of images, he/she can hit the `Create` button. This button is made using `< Button / >`, the same as `HOME` and `TIMELAPSE` buttons. When the user clicks this button, the "magic" happens, making use of the event handler `onClick()`, which refers to the custom function called `createGIF()`. Whenever this function is called we first set the state of the application to `loading`, so the user knows that something is being done in the background. We then make a `GET` request from our server with the folder selected by the user. Then the response is taken and worked with inside an arrow function. We create an empty array that gets filled with the images coming from that specific folder. The number of images is set to choose between the smallest value of image number selected from the user and the actual number of images located inside the folder, in case there are less in the folder than the number selected. After our array is ready and full with the appropriate number of images, we use `createGIF()` from gifshot[90] to create the time-lapse. I has a lot of `props` that one can use to create the GIF, but for us only 4 of them are needed: `images`, `interval`, `gifWidth`, and `gifHeight`. We set the images to the array that we just filled, for the interval we set it to the timer located inside the `state` object, and for the width and height we give it 1600 and 900 accordingly, which is also referred as our time-lapse resolution. We then call a function inside `createGIF()` that sets the finished time-lapsed images to the state of the application as `src`. However, since this function is located on the cloud, there are some limitations to using `createGIF()`:

- The user needs to have an active internet connection
- We cannot calculate the time it takes to process the images and create the GIF
- We have to take account to the request timeout, which is usually 30 seconds
- The higher the resolution the smaller the image number
- The smaller the image number the worse the presentation

Usually for the GIFs all over the internet, the resolution is not that great, and it shows, because they also want to make the file size smaller and the upload / download faster. In our case, since this is referred as a time-lapse, we needed the image to be as clear as possible for presentation purposes. So we tested around with the resolutions and the timeouts of the request and found out that a sweet spot was 1600x900 to also take the image number to an acceptable one.

4.8 Control File

Our control file, like most other React client projects, is called "App.js". Everything that has been explain until this point is almost in all cases related to this file with variables being passed down from it. Firstly, this file has the biggest `state` of them all initializing:

- *page* : "home"
- *selectedIndex* : 0
- *loading* : true
- *error* : null
- *images* : []
- *timer* : 5
- *slideshowOpen* : false
- *sliderLoading* : false
- *sliderError* : null
- *sliderImages* : []
- *filtersMenuOpen* : false
- *imagesFilters* : {}

The names are pretty much intuitive and self-explanatory on what each of the state items' function is inside the application. In "App.js" we mostly take care of the rendering and the put together of every other component while using them and provide their inputs. We have a life-cycle method for handling what happens when our application is given "birth"[64]. It does a *GET* request to the server and retrieves all the information needed for the application to start to work, i.e. it fills the *images* and *imageFilters* in our *state* with all the latest images and every folder using *setState()* function. It also sets *loading* to *false* to stop the circular progress animation and *error* to *null*. It is only called once before the rendering of the application and never again.

We have a function called *renderLayout()* that from the name it makes sure that it renders each layout properly. It has a *switch* statement that controls which layout to display using *selectedIndex* that comes from the *state*. When it has found a layout, it returns it by calling the specific layout component out of the total nine that we have created. It makes sure to pass down all the filtered *state* images, using an arrow function, the *state* timer, and a function as an event handler of *onCellClick()*. The function is called *openDialog()* and is used to open the slide-show modal whenever the user clicks on a picture displayed in the layout. It does that by using a *GET* request from the server where it makes sure to get the specific folder needed. With the response data, i.e. all images inside that folder, coming from the request, we make sure to assign it to our *sliderImages* to the *state* by using *setState()*.

renderLayout() is used inside a function called *renderContent()*, which is responsible for rendering the content of the error message using *< Typograhpy / >*, the circular progress using *< CircularProgress / >*, the time-lapse content, and the homepage content. The time-lapse content is rendered using *< TimeLapse / >* custom component passing down *loading*, *error*, and *images* from the *state* of the application. Regarding the homepage content, it makes sure to render the layout using *renderLayout()*. Then, we render the slideshow using *< Slideshow / >* custom component by passing down *slideShowOpen*, *sliderLoading*, *sliderError*, and *sliderImages* inputs from the *state*, and lastly an event handler called *onClose()*, which refers to another function called *closeDialog()*. This

function makes sure that the slide-show is closed by updating the *state* of the application using *setState()*. After that, we render the filters menu using `< FilterMenu / >` custom component while passing down *filtersMenuOpen*, *images*, and *imageFilters* inputs coming from the state, too. We also have to handle the closing of this component with the function *closeFiltersMenu()*, which updates the *state* using *setState()*. There is one more *props* that we have to handle and that is *setFilter* using the function with the same name, which updates the *state* with the appropriate checked filters using *setState()* as well.

We have two more important functions called *renderFilters()* and *renderTextInput()*. *renderFilters()* handles the `< div / >` in the GUI by using the *state* to find out how many of the folders are checked to display. The whole function is built from `< Button / >` and that allows us to open the filters menu using the event handler function *onClick()*, which refers to *openFiltersMenu* that updates the *state* to open the custom component. Whereas, *renderTextInput()* function is what we have talked thoroughly in section 4.6.3.

Lastly, we use the *render()* method to combine everything together

4.9 Automation

To automate the system, in the beginning we had to figure out a way to externally control the opening and the closing of the React client and the HTTP server. In order to do that, we had to work with the programs on a processing level, i.e. try to find both of them on task manager, since the environment was Windows 10, and try to control them. We decided it was best to use Python, for the easiness of the syntax and the availability of many libraries that provided us with a possible solution.

The logic that was come up with was:

- Keep the python script running at all times
- Only stop the script in case of an error
- The script would restart the applications at 06:00:00 sharp
- In all other times, it would remain on standby and have no functionality
- The script would have to wait between different openings and closing of the applications to take into account the time it took for each of them
- The script would display messages so the user can see what is happening in the meantime while waiting for the restart and confirming what was done

4.9.1 React Client Windows Application

But, the idea of this entire thesis was to also use many instances of the application in different monitor setups, so we could not control them all, the way we wanted to. Normally, one would start the React client with the command: "npm start" and then you can open multiple instances of *localhost : {port nr.}* on a browser and have them to display in multiple monitors. We could try to work with the localhost and cmd line, but React automatically would open a new tab of localhost on the browser. This would be an inconvenience, since the user would want to find the applications as they left it while they restart. Even if we found a way to properly restart and not automatically open the localhost, every tab would go back to their default layout and the different layouts selected by

the user would be reset, and that was also an inconvenience. Since we wanted to provide an optimal solution for the user not to touch or work with the application, and have everything automated, this idea was brought down quickly and we needed to provide another way of handling this.

After further evaluations, since React was only there to only display and render the latest images retrieved by the HTTP server, we did not need to restart the client, but only the server. This would be good, because the React client would not change properties provided by the user, since it got the new pictures automatically once the server provided them again. If the restart would be fast enough, the restart would be nearly unnoticeable from the user even if he/she would be staring at it.

However, to make things easier for every different kinds of users, in case of any problems, we needed to stop using a manual script and try to automate the client, as well. A way to do that was to build a proper Windows application from it and the user would open an instance of it like any other normal application. That would be possible using *electron-packager*[62]. This package allowed us to bundle the React client to a single executable application. But, it would take a significantly bigger space and time to build than normal, because it would also install an entire browser inside. And now, the application would open in a separate window, not inside e.x Chrome/Safari and cannot be compiled anymore using a cmd line script. It can go into full-screen however, and provide the same functionality as before.

4.9.2 HTTP Server Windows Application

For the same reasons as explained, we had to come up with a way to automate the server as well, but we could not use a packager, because the server would need to be compiled every-time to be able to retrieve the latest images being updated every-day. We solved this by making a python script to re-compile the server every-time. Then, we made an executable of this python script called *server.exe*, using *pyinstaller*[87]. Now, every-time the user clicks the python executable, the server will re-compile automatically.

To do that, we modified a function provided by another author called *process_exists()*[20]. This function checks all currently open applications in the task manager. You can check if a process is running or not by giving as input its name, and concretely our HTTP server project is called "*node.exe*". We use this function to check this every-time the user decides to re-compile the server, so that we don't open a new instance with the same port number, because that would refuse to open to the same port number or have some unexpected errors. If the instance of the application is already running and function finds it, we kill it using the command *os.system("taskkill /f /im node.exe")* from *os*[59]. After we kill it, we wait for 1 second for windows to close the application and we open it again with the command *Popen('{path to node project} && node index.js', shell=True)* using *Popen()* function from *subprocess*[60]. Then we wait for 5 seconds to let the application open properly. If there does not exist any instance of the application before-hand, we simply open it and wait for 5 seconds.

To properly terminate the application in case it is not, when one closes the cmd window, we have implemented a functionality that can. If the user inputs the signal *CTRL + C*, then the application will kill itself with the same command that kills other instances, and will make a proper *sys.exit(0)*.

This Windows application was created in case the user wants to manually start (compile)

the server in case of an unexpected error to the main automation python script explained in the subsection below.

4.9.3 Python Automation Script

The functionality of this python script and the server python script does not have too many differences. This script also uses the function called *process_exists()* to check if a process is running or not. It also only checks for our HTTP server instance and only re-compiles that. It uses the same python library functions to kill the server and compile it. It also has a proper shutdown with the same properties as *server.exe* when the user inputs the signal *CTRL + C*. It also build into a proper windows application using *pyinstaller*, and it is called *automation.exe*. The only difference is that it runs all the time and it only provides the functionality we have talked about at 06:00:00.

5 All Layouts

5.1 Layout: 4x4



Figure 11: Layout 4x4

5.2 Layout: 3x3



Figure 12: Layout 3x3

5.3 Layout: 2x2



Figure 13: Layout 2x2

5.4 Layout: 1x1



Figure 14: Layout 1x1

5.5 Layout: 1+12



Figure 15: Layout 1+12

5.6 Layout: 2+8



Figure 16: Layout 2+8

5.7 Layout: 1+7



Figure 17: Layout 1+7

5.8 Layout: 1+11



Figure 18: Layout 1+11

5.9 Layout: 18+1



Figure 19: Layout 18+1

6 JavaScript + React.JS + Node.JS

6.1 What is JavaScript and its Uses

First of all, for our web-site related programming project, we decided to use JavaScript, because it has had the biggest hand in developing dynamic and interactive web pages since 1995[72]. Over the past 20 years or so since when JavaScript was released, it has become the most popular programming language for web development today.[71] Being one of the most powerful client-side programming language, it has not just kept its position maintained as a top programming language after Java, but also been empowering 96.2% of the websites on the internet.[27] In addition, it has been used by 10 million developers today and it is the only interpreted language that has been universally accepted by all web applications and browsers.[27]

That's why it stands up second in the list and software companies are widely using it in various areas as mentioned below:

- **Web Development:** JavaScript is a client scripting language that helps in creating dynamic web pages with special effects on pages and also supports external applications including PDF documents, running widgets, and more on the website.[27]
- **Presentations:** JavaScript is not only a powerful programming language but also provides a wide choice of libraries that help you develop a web-based slide back. Basically, reveal and BespokeJS are the two most common libraries that enable you to create the most beautiful and interactive decks using HTML.[27]
- **Web Servers:** Using Node.JS, developers can easily create a web server. The biggest advantage of using Node.JS is, it does not wait for the responses of the previous call as it leverages the event to get a notification. The servers built on Node JS are very fast and never transfer the chunks of data.[27]
- **Mobile Application:** When it comes to mobile applications, there are majorly two popular operating systems iOS and Android which require two different languages to build apps for both the platforms. However, with JavaScript's most potential framework "PhoneGap" make it is possible to write once and use it on both platforms.[27]

Apart from all these uses, JavaScript and its frameworks are popularly used for creating games, drawing graphics, flying robots, and contributing in other web app development solutions, with the combination of JavaScript and HTML.[27] According to the Stack Overflow survey report of 2019[65], it is revealed that JavaScript is the most popular technology for programming, scripting, and markup languages. And currently, 69.7% of professional developers are using JS for modern developments.

Most Popular Technologies

Programming, Scripting, and Markup Languages

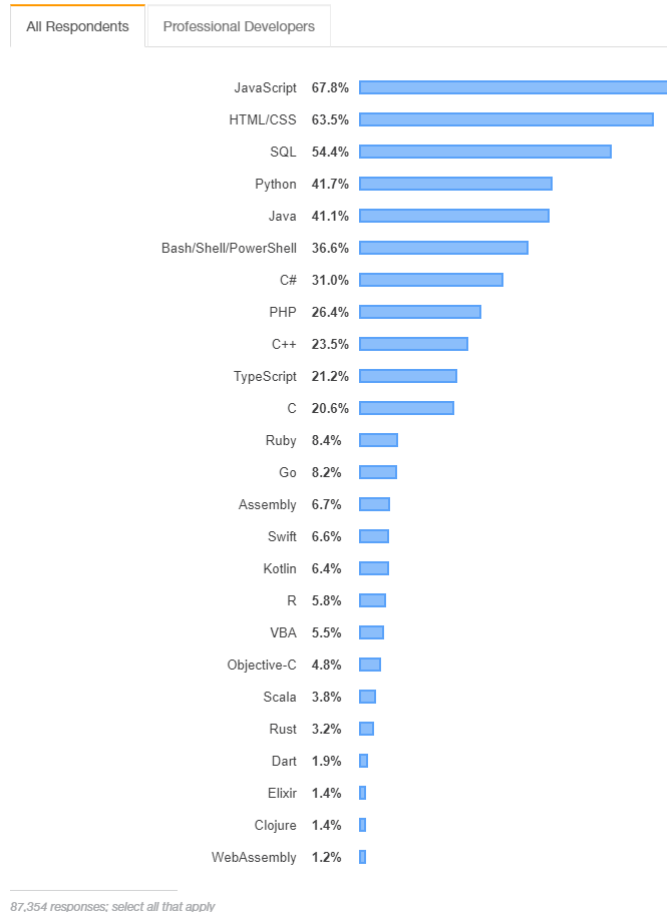


Figure 20: JavaScript Usage[65]

6.2 Reasons to Use JavaScript for Modern Web App Development in 2020

With the help of its frameworks, developers can easily develop hybrid applications for different platforms including iOS, Android, and Windows. And one of the prime reasons for using JavaScript for app development is it allows pages to be very engaging, presenting effective graphics, verify data, create cookies, responsive to events, and more.[27]

6.2.1 Minimize the Complexity of Web App Development Process

By accessing the JavaScript library, a developer can easily create domain (DOM) shadow boundaries which in return minimize the complexity of app composition. Furthermore, in order to simplify the web app development process, the shadow DOM further segregates the components of each JavaScript library.[27]

6.2.2 Ease of Writing Server-side Code in JavaScript

One of the most common reasons for using JavaScript for web app development solutions is the ability to write server-side code in JS. By using a cross-platform run-time[85] engine

like Node JS, developers can further execute the JavaScript code efficiently and with the help of built-in libraries, programmers can make their app run smoothly without using any external web servers.[27]

6.2.3 MEAN Stack: 4 Major Components in a Single Pack

As we said above, developing anything without JavaScript seems impossible for the developers. MEAN stack is a powerful package that not only helps in developing modern web apps but also simplifies the development process. MEAN stack consist of MongoDB, Express.JS, AngularJS, and Node.JS. On the one hand, MongoDB is a modern and schema-less NoSQL database whereas Node JS is a popular cross-platform that provides a server-side run-time environment for the simple web app development structure. At the same time, Angular JS and Express JS are commonly used JavaScript Frameworks that have been designed with the features to simplify the app development process.[27]

6.2.4 Hassle-Free Integration of Multiple Transpilers

Compared to others, JavaScript is a lightweight programming language but it lacks some robust features that you may need in modern app development. However, that's not the biggest issue for the developers as they can easily extend the features of JavaScript by using multiple Transpilers like TypeScript, DukeScript, CoffeeScript, and Vaadin. Each transpiler empowers developers with more features and allows them to meet their growing requirements for developing large enterprise applications in a hassle-free manner.[27]

6.2.5 Ability To Develop Responsive Web Pages With JavaScript

Even Google says, Mobile First! It means whether you are developing a website or a web app, make sure its pages are accessible all across multiple browsers and devices. It is true that responsive web design enables developers to optimize a web page for both computers and mobile devices with a single code base, but to make it possible developers have to combine HTML, CSS3, and JavaScript. And while developing the web apps by leveraging JS libraries and frameworks, you don't further need to integrate any other coding to make your app responsive. JavaScript is already known for making dynamic web pages and based on HTML & CSS.[27]

6.2.6 A Broad Access of Libraries and Frameworks

JavaScript provides access to a wide choice of libraries and frameworks that allow developers to extend the functionality of JavaScript. There are various heavily feature-loaded libraries like AngularJS, Ember that allow web developers to add functionality to more complex web applications without any need of writing additional codes. On the other hand, JavaScript's lightweight libraries like React Js make it easier for developers to accomplish the specific task without any hassle. Furthermore, developers can choose to install open-source tools like NPM and manage all JS libraries for web app development much more efficiently.[27]

6.3 Popular JS Frameworks That You Can Use For Web App Development

JavaScript frameworks are basically the frameworks that have been written in JS. These frameworks are written in a way that allows developers to code the application in such a manner to make it run on multiple channels. JS frameworks are a kind of hardware that makes it working with JavaScript in a far simpler and smoother way.[27]

These are the few popular JavaScript frameworks that you can choose for the development:

- **Angular JS For Front-End Development:** Angular JS is an open-source framework backed by Google and utilized for developing single-page applications.[27]
- **React JS For Backend Development:** It is a powerful JS framework library owned by Facebook, that has been widely used for building dynamic and high performing web applications in a fast turnaround.[27]
- **Ember JS:** This framework is a composition of the Rails that provides more freedom and flexibility to write code and build interactive web design within the reduced time.[27]
- **Node JS:** It is event-driven that doesn't wait for the input and output responses. Rather, it initiates a function in a call-back queue and helps in building real-time applications that ensure cluster-based performance.[27]
- **Vue.JS:** An advanced version of AngularJS that ensures you a two-way data binding and virtual DOM for the development. Vue.js is the lightweight library and loved by hundreds of developers.[27]

6.4 What makes React so fast

6.4.1 Virtual DOM

The whole idea behind React is approaching web-sites in a "puzzle"-like manner. Meaning you create piece-by-piece every component that your web-site has and then bring them all together into the website like a puzzle. This approach makes the web-site more maintainable and assure quality since you can easily adjust and debug different components separately in case of an issue that may occur. Moreover, one of the strongest features of React, besides breaking things into components, is the use of its Virtual domain (VirtualDOM). DOM manipulation is the heart of the modern, interactive web.[7] Unfortunately, it is also a lot slower than most JavaScript operations.[7] This slowness is made worse by the fact that most JavaScript frameworks update the DOM much more than they have to.[7] E.x. say you need to update a small text in your big website. Only that small text differs and everything else is the same, but the whole web-site has to be rebuilt again, and that is more work than necessary. VirtualDOM fixes this performance issue. In React, for every DOM object, there is a corresponding "virtual DOM object".[7] A virtual DOM object is a representation of a DOM object, like a lightweight copy.[7] It sounds bad performance-wise, because it is a copy of the DOM and whenever you change something the whole Virtual-DOM is updated, but the way it is used by React is what makes it special. Manipulating the DOM is slow.[7] Manipulating the virtual DOM is much faster, because nothing gets drawn onscreen.[7] Think of manipulating the virtual DOM as editing a blueprint, as opposed to moving rooms in an actual house.[7] Once the Virtual-DOM has been updated, React compares the virtual DOM with a virtual DOM

snapshot that was taken right before the update. By doing this, React figures out exactly which object has changed and only updates that specific one on the real DOM. React also makes working with JavaScript easier, because it has all these functions that take care of your code and transform it to a JS code the web-service can understand, so you don't need to write in a pure JavaScript syntax.

6.4.2 Diffing Algorithm

Like the actual DOM, the virtual DOM is just a node tree[84] that lists elements and their attributes and content as objects and properties.[5] Their tree diffing algorithm is actually incredibly simple and based on the following two assumptions:

1. Two elements of different types will produce different trees
2. The developer can hint at which child elements may be stable across re-renders with a key prop

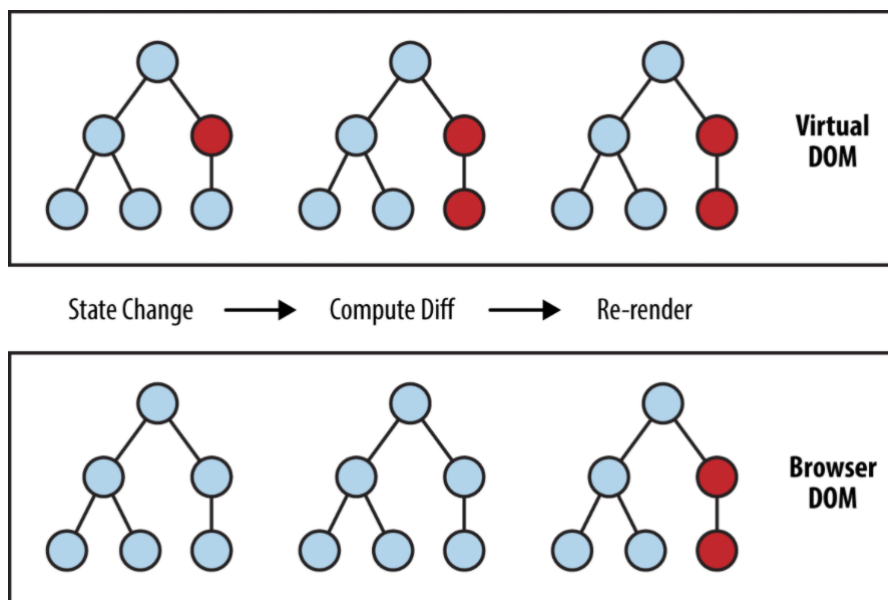


Figure 21: Performing calculations in the virtual DOM limits rendering[81]

If the root DOM element is different, it completely tears down the old tree and begins from the root of the new tree. On the contrary if the root DOM element is the same, then the algorithm compares each and every difference in attributes, keeps the same valued attributes and changes only the new/changed attributes.[5]

Next, when an element contains multiple child nodes (i.e. a series of $|l_i|$ elements), React will check for differences at the same time step by step. If the only differences in the child nodes are at the end, then that addition will be noted as the only update but if an element is added at the beginning all the subsequent children will also be flagged to update. In order to solve for this, React implemented a key attribute which it leverages to match up the children when running the comparison.[5]

The last piece that increases React's performance is that it actually compiles a list of all these changes necessary to the DOM and then batches them. This is done to avoid the

browser's DOM from triggering a re-painting of the UI, as this is the most expensive and inefficient part of DOM updates. Instead of sending updates for each single change in state, these are sent over to the DOM in one batch to ensure that re-painting event is only performed once.[5]

React has intentionally chosen to keep their algorithm this simple in order to optimize for time efficiency while still keeping performance at a nearly identical level to some of the state of the art algorithms. They note that while those algorithms could lead to small performance gains they have a complexity somewhere in the order of $O(n^3)$ vs. their simple algorithm's linear time — $O(n)$.[5]

6.4.3 Single-way Data Flow

In React, a set of immutable values are passed to the components rendered as properties in its HTML tags. The component cannot directly modify any properties but can pass a call back function with the help of which we can do modifications. This complete process is known as “properties flow down; actions flow up”.[83]

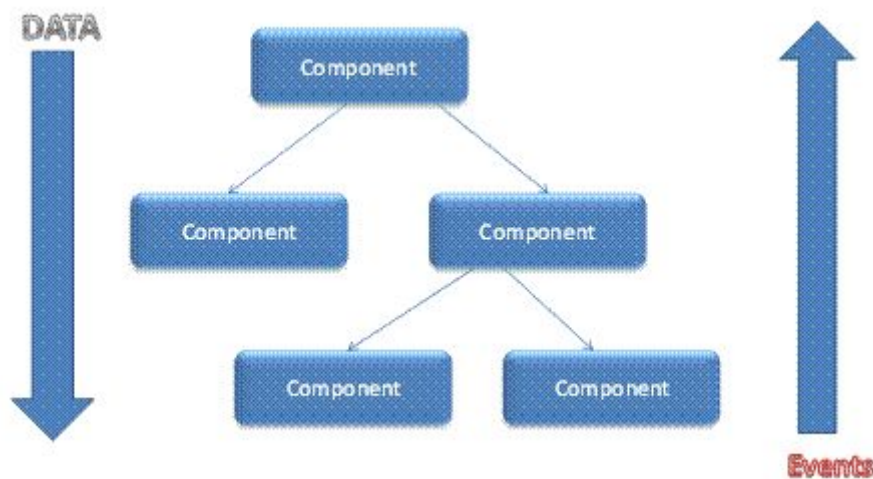


Figure 22: React Flow[83]

6.5 What makes Node so great

the most revolutionary thing about Node.js is that it is the first-ever environment supporting JavaScript both client-side and server-side.[46] What started small, is now an open-source with MIT licence, supported by a massive community and hundreds of add-ons.[61] According to Stack Overflow 2019 survey[46], Node.js is now the most popular tool in “Frameworks, Libraries, and Tools” category with 50.4% of professional developers using it.

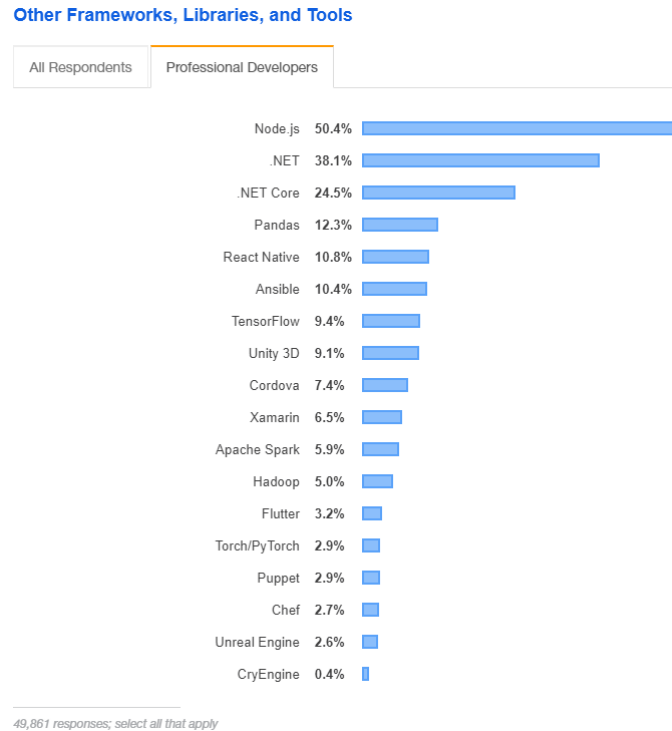


Figure 23: Node Use Survey[66]

6.5.1 Single programming language

Nowadays, the language of choice on a front-end is JavaScript. Because you can also use it on backed, it improves the app’s maintainability. You no longer have to hire two people for separate back-end and front-end positions. It’s a huge advantage to save your time and money.[61]

6.5.2 Large Community

Node.js, being an open-source project, encourages support and contribution aimed at the improvement and adoption of the platform. This is the mission of its Foundation intended for continuous development and enhancement of Node.js. Therefore, you can be sure that, on one hand, Node.js is always getting better and, on the other hand, there is already a lot of reusable resources.[46]

6.5.3 API

Nowadays, it’s the most popular type of application. Nearly all web applications need some kind of backend to communicate with databases or other external services. Thanks to the big community, you can easily find a library to create e.g. REST or GraphQL API. In the past, Node.js was recommended only for applications with a low number of CPU-intensive operations. Since release version 10.5.0 you can use worker threads to do it.[61]

6.5.4 Scalability

This is a true jewel of the Node.js development environment, as it allows building applications that can easily grow with your business. Node.js works great in systems using the microservices architecture or containerization where the scalability and flexibility can be achieved quickly and easily.[46]

6.5.5 Real-time web applications

Because Node.js is very good at handling lots of I/O operations, you can use it to build a real-time web application, for example, a chat room where people can talk to each other in real-time. Or maybe a collaboration tool, where co-workers will work on the same document at the same time. Done! Building a video conference web application also won't be a problem. It is possible thanks to Node's Events API and WebSockets.[61]

6.6 Conclusion

JavaScript has made web development a lot easier for everyone, while maintaining a top position with the increasing technologies and programming languages over time. The benefits mentioned above clearly define its power in the development community and how it is simplifying the web app development process.

React JS had a personal advantage over Angular JS, because it was a language that I had worked with before in my internship. I was familiar with the way it worked and how to use it efficiently.

Manipulation of the browser DOM is not inherently slow, it's their painting of the UI that is costly. React's virtual DOM helps to minimize these painting events by making sure only the elements that need to be changed in the DOM are manipulated, and that these updates are sent in batches. These batches of updates prevent unnecessary "slow" painting events and make React applications more efficient.[5]

Node.js is a powerful development framework showing excellent performance in many cases. It works very well with JavaScript and its libraries and frameworks.

All in all, you cannot go wrong when combining JavaScript, React.JS, and Node.JS. It is one of the most optimal and used solutions worldwide for web development.

7 Material-UI vs Bootstrap

Superior user experience is becoming increasingly important for businesses as it helps them to engage users and boost brand loyalty. Front-end website and app development platforms, namely Bootstrap vs Material Design empower developers to create websites with a robust structure and advanced functionality, thereby delivering outstanding business solutions and unbeatable user experience.[4]

7.1 Bootstrap

Bootstrap is an open-source, intuitive, and powerful framework used for responsive mobile-first solutions on the web. For several years, Bootstrap has helped developers create splendid mobile-ready front-end websites. In fact, Bootstrap is the most popular CSS

framework as it is easy to learn and offers a consistent design by using re-usable components.[4]

7.1.1 Pros

- **High Speed of Development**

If there is limited time for the website or app development , Bootstrap is an ideal choice. It offers ready-made blocks of code that can get the developer started within no time.[4]

Bootstrap also provides ready-made themes, templates, and other resources that can be downloaded and customized to suit the needs of the developer, allowing him/her to create a unique website as quickly as possible.[4]

- **Bootstrap is mobile first**

Since July 1, 2019, Google started using mobile-friendliness as a critical ranking factor for all websites. This is because users prefer using sites that are compatible with the screen size of the device they are using. In other words, they prefer accessing responsive sites.[4]

Bootstrap is an ideal choice for responsive sites as it has an excellent fluid grid system and responsive utility classes that make the task at hand easy and quick.[4]

- **Enjoys a strong community support**

Bootstrap has a huge number of resources available on its official website and enjoys immense support from the developers' community. Consequently, it helps all developers fix issues promptly.[4]

Furthermore, there are plenty of websites offering Bootstrap tutorials, a wide collections of themes, templates, plugins, and user interface kit that can be used for a project.[4]

7.1.2 Cons

- **All Bootstrap sites look the same**

The Twitter team introduced Bootstrap with the objective of helping developers use a standardized interface to create websites within a short time. However, one of the major drawbacks of this framework is that all websites created using this framework are highly recognizable as Bootstrap sites.[4]

- **Bootstrap sites can be heavy**

Bootstrap is notorious for adding a lot of unnecessary bloat to websites as the files generated are huge in size. This leads to longer loading time and battery draining issues. Furthermore, if you delete them manually, it defeats the whole purpose of using this framework.[4]

- **May not be suitable for simple websites**

Bootstrap may not be the right front-end framework for all types of websites, especially the ones that do not need a full-fledged framework. This is because, Bootstrap's theme packages are incredibly heavy with battery-draining scripts[4][1]. Moreover, Bootstrap has CSS weighing in at 126KB and 29KB of JavaScript that can increase the site's loading time.[4]

7.2 Material Design

Material-UI is a React UI framework that follows principles of the Material Design. It is based on Facebook's React framework and contains components that are made according to Material guidelines[82]. When compared to Bootstrap, Material Design is hard to customize and learn. However, this design language was introduced by Google in 2014 with the objective of enhancing Android app's design and user interface. The language is quite popular among the developers as it offers a quick and effective way for web development. It includes responsive transitions and animations, lighting and shadows effects, and grid-based layouts.[4]

Material-UI is strongly connected with Material Design, but one should not confuse the two. Material-UI is just a react component library without Material design, that is why we built our comparison from the perspective of the fact that Material guidelines go first, Material framework follows. Google uses Material Design in all of its products, Material-UI is used by Nasa, Amazon, Unity, JPMorgan, etc.[82]

7.2.1 Pros

- **Offers numerous components**

Material Design offers numerous components that provide a base design, guidelines, and templates. Developers can work on this to create a suitable website or application for the business. The Material-UI concept offers the necessary information on how to fully utilize each component.[4]

- **Is compatible across various browsers**

Both Bootstrap vs Material Design have a sound browser compatibility as they are compatible across most browsers. Material Design supports React Material User Interface. It also uses the SASS pre-processor[4], which is the most mature, stable, and powerful professional grade CSS extension language in the world.[6]

- **Does not require JavaScript frameworks**

Bootstrap completely depends on JavaScript frameworks. However, Material Design does not need any JavaScript frameworks or libraries to design websites or applications. In fact, the platform provides a material design framework that allows the developers to create innovative components such as cards and badges.[4]

7.2.2 Cons

- **The animations and vibrant colors can be distracting**

Material Design extensively uses animated transitions and vibrant colors and images that help bring the interface to life. However, these animations can adversely affect the human brain's ability to gather information.[4]

- **It is affiliated to Google**

Since Material Design is a Google-promoted framework, Android is its prominent adopter. Consequently, developers looking to create applications on a platform-independent UX may find it tough to work with Material Design.[4]

- **Carries performance overhead**

Material Design extensively uses animations that carry a lot of overhead. For instance, effects like drop shadow, color fill, and transform/translate transitions can be jerky and unpleasant for regular users.[4]

7.3 Conclusion

Bootstrap is great for responsive, simple, and professional websites. It enjoys immense support and documentation, making it easy for developers to work with it. So, if one is working on a project that needs to be completed within a short time, opt for Bootstrap. The framework is mainly focused on creating responsive, functional, and high-quality websites and applications that enhance the user experience.[4]

Material Design on the other hand, is specific as a design language and great for building websites that focus on appearance, innovative designs, and beautiful animations, e.x. portfolio websites. The framework is pretty detailed and straightforward to use and helps one create websites with striking effects.[4]

In conclusion, the most recommended and used framework is Bootstrap. However, these are some of the reasons why Material-UI was used in this project:

- The `< Grid/ >` component in Bootstrap did not feel right and caused bugs when we tried to modify it to the application's needs.
- The `< Grid/ >` component in Material-UI used a more intuitive way of implementation.
- The application is a little different from all the Bootstrap-build websites, and for me that was important.
- I wanted the application to look vibrant and "alive" with the colors provided by Material-UI.
- Material-UI had better UX design than Bootstrap.
- Bootstrap adds a lot of unneeded bloat.
- Since it adds so much bloat, it was not very suitable for our simple website.
- Material-UI has bloat as well, but we can handle while minimizing the bundle size[36] of the application.
- We did not need the application to be super responsive, because it is going to be used accessed from monitors.
- Since we managed to implement our own custom grid layouts, in the end, it did not matter much which one we used.

8 FTP Server vs HTTP Server

HTTP and FTP both are the file transfer protocols that are used to transfer data between client and server. HTTP functions similar to the combined functioning FTP and SMTP. FTP is a protocol that sorts the problem when a communicating client and server have different configuration.[8]

The basic point that distinguishes HTTP and FTP is that HTTP on request provides a web page from a web server to web browser. On the other side, FTP is used to upload or download between client and server.[8]

8.1 FTP

While transferring a file from one host to another the problems that may occur are, the communicating host may have different file name conventions, may have different directory structures, different way to represent data. FTP overcomes all these problems. FTP is used when two hosts with different configurations want to exchange data between them.[8]

FTP has no meta-data when using file transfer and only the raw binary.[9] In our case, we get the latest pictures from the file name of the pictures, so if no meta-data exists, we cannot access the pictures with this method. We would need to come up with an image processing algorithm to read written information inside each picture. This is also a cons and pro of the FTP, which makes it faster than HTTP in this context because, when sending small files, the headers can be a significant part of the amount of actual data transferred.[9]

8.1.1 Pros

- **Secure Data**

FTP is mostly used to store secure data and information, unless shared by the admin. For secure transmission that protects the username and password, and encrypts the content, FTP is often secured with SSL/Transport Layer Security (TLS), also known as FTPS. Or, it is replaced with Secure Shell (SSH) File Transfer Protocol (SFTP).[67]

- **Directory Listing**

One area in which FTP stands out somewhat, is that it is a protocol that is directly on file level. It means that FTP has for example commands for listing directory contents of the remote server, while HTTP has no such concept.[76]

However, the FTP spec authors lived in a different age so the commands for listing directory contents (LIST and NLST) do not have a specified output format, so it is very difficult to write programs to parse the output. Latter specs (RFC3659) have addressed this with new commands like MLSD, but they are are not widely implemented or supported by neither servers nor clients.[9]

- **Packet Verification**

One of the top qualities of transferring files with FTP is the packet verification. What this means is that FTP is somewhat slower for a given size file, but the file is more likely to get through in one piece. That is because for a binary application file, for example, one lost packet is as good as losing the whole thing.[76]

- **Two-Way System**

FTP is a two-way system. This means that any file can be transferred from the server to the client and vice-versa, from client to the server.[76]

- **Good for smaller transfers**

Compared to HTTP, FTP puts less load on the network for smaller transfers. For transfers of large amounts of data, both are comparable.[76]

- **FXP**

FTP supports "third party transfers", often called "FXP". It allows a client to ask a server to send data to a third host, a host that isn't the same as the client. This is often disabled in modern FTP servers though due to the security implications.[9]

- **IPv6**

HTTP and FTP both support IPv6 fine, but the original FTP spec had no such support and still today many FTP servers don't have support for the necessary commands that would enable it. This also goes for the firewalls in between that need to understand FTP.[9]

8.1.2 Cons

- **State-full Connection**

FTP is built on a client-server architecture and establishes two separate TCP connections[67]:

- Control Connection (command port; port 21) to authenticate the user.
- Data Connection (data port; port 20) to transfer the files.

FTP has a stateful control connection, therefore the FTP server will maintain state information like a user's current directory for a session. This can constrain the total number of sessions FTP can maintain simultaneously. FTP also requires client authentication in order to transfer or serve information successfully.[67]

- **FTP Command / Response**

FTP involved the client sending commands to which server responds. A single transfer can involve quite a series of commands. This of course has a negative impact since there's a round-trip delay for each command. Retrieving a single FTP file can easily get up to 10 round-trips.[9]

- **Two Connections**

One of the biggest hurdles about FTP in real life is its use of two connections. It uses a first primary connection to send control commands on, and when it sends or receives data, it opens a second TCP stream for that purpose.[9]

- **Firewalls and NATs**

FTP's use of the two connections, where the second one use dynamic port numbers and can go in either direction, gives the firewall admins grief and firewalls really have to "understand" FTP at the application protocol layer to work really well. This also means that if both parties are behind Network Address Translation (NAT), you cannot use FTP! Additionally, as NATs often are setup to kill idle connections and the nature of FTP makes the control channel remain quiet during long and slow FTP transfers, we often end up with the control channel getting cut off by the NAT due to idleness.[9]

- **Active and Passive**

FTP is a text based protocol that operates on two channels, namely the command channel and the data channel. For the data channel, we have to differentiate between two data transfer modes, Active and Passive.[89]

- **Passive** mode is where the client asks the server for IP and port number and then opens a separate data channel to the specified destination[89]
- **Active** mode is the older one of the two and is rarely used nowadays. The client asks the server where it can be found and then have the server initiate the data connection. The main reason why this mode is not really used anymore, is because of Firewalls and NATs that generally block all incoming connections. These protective technologies were not used back in the day and therefore they caused no issues with a mode like this. Regardless, it is yet to be found a case where **Active** mode makes sense and why it has been introduced in the first place.[89]

The following chart represents a conversation between a client and a server, with the client intending to upload a picture into the folder "/foo/bar":

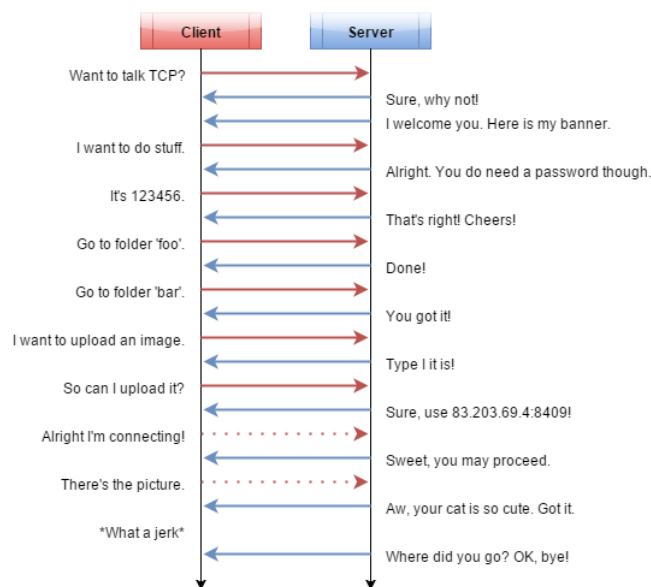


Figure 24: FTP client-server conversation[89]

As made obvious, this is a ridiculous number of back-and-forth messaging that leads to a huge waste of time due to round-trip delay and control overhead. The delays occur not only on FTP level, but also on TCP level since the transfer of every single file requires a new TCP connection to be opened, introducing yet another TCP handshake (connection).[89] Depending on the use case, this might be acceptable, though in our case we are serving a large number of files (pictures), so performance is a must here.

Using FTP poses another problem, there is no verification whether a transfer has been completed successfully or not. The server is even unaware of the file size. As much as FTP is concerned, once a stream ends, that is the end of the file. It does not matter if the connection was interrupted and half the file is missing. At least the transfers can usually be resumed from where they have been stopped, that is, if you notice or know somehow that the file is corrupted in the first place.[89]

- **Download**

In order to serve the pictures from the server to the client, we need to download them first from the FTP server[8]. Since we have a recursive way of getting the latest images from the folders, we are not really sure of how many pictures and folders there are going to be in total. Furthermore, we need to download a lot of pictures. If we have command line access to the server, we need to log in automatically, and bundle everything into one file with zip, gzip, bzip or tar (whichever is available, and can be reversed on the destination) with a third-party program. This reduces the number of files and bytes we have to download; and the number of the connections that have to be opened every time.[2]

- **Persistent Connections**

FTP must create a new connection to the client for each new data transfer.[9] For example, if you have 10 files to transfer, the FTP server opens 10 connections in total (a new connection every time). Repeatedly doing new connections are bad for performance due to having to do new connections all the time and redoing the TCP slow start period and more.[9] The protocol overhead might be negligible for large files but it completely destroys performance when numerous tiny files must be transferred.[89]

- **Ranges / Resume**

FTP supports resumed transfers in both directions, but not as many byte ranges as HTTP. Resumed transfers for FTP that start beyond 2GB position has been known to cause trouble in the past but should be better these days.[9]

- **Compression**

FTP offers an official "build-in" run length encoding that compresses the amount of data to send, but not by a great deal on ordinary binary data. It has also traditionally been done for FTP using various "hackish" approaches that were never in any FTP spec.[9]

- **Proxy Support**

FTP has always been used over proxies, but that was never standardized and was always done in a lots of different ad-hoc approaches.[9]

8.2 HTTP

HTTP stands for Hyper Text Transfer Protocol. It is the foundation of data communication of the World Wide Web (WWW) - as in, the whole WWW runs on it. HTTP is the backbone of the WWW and it defines the format of messages through which web browsers (Chrome, Firefox, Edge, etc.) and web servers communicate. It also defines how a web browser should respond to a specific web request.[67]

8.2.1 Pros

- **Stateless Connection**

HTTP also uses Transmission Control Protocol (TCP) as an underlying transport and typically runs on port 80. It is a stateless protocol since each command is executed independently, without any knowledge of the commands that came prior. A stateless

protocol is a communications protocol in which no session information is retained by the receiver, typically a server.[67]

- **Transfer Speed**

What makes HTTP faster:

- reusing existing persistent connections make better TCP performance.
- pipelining makes asking for multiple files from the same server faster.
- (automatic) compression makes less data get sent.
- no command/response flow minimizes extra round-trips.

Ultimately, the net outcome of course differs depending on specific details, but for single-shot static files, you will not be able to measure a difference. For a single-shot small file, FTP might be faster (unless the server is at a long round-trip distance). When getting multiple files, HTTP is the faster one.[9]

- **Headers**

Transfers with HTTP always also include a set of headers that send meta data. Its headers contain info about things such as last modified date, character encoding, server name and version, etc.[9]

- **Pipelining**

HTTP supports pipelining. It means that a client can ask for the next transfer already before the previous one has ended, which thus allows multiple documents to get sent without a round-trip delay between the documents. TCP packets are thus optimized for transfer speed.[9]

- **Authentication**

FTP and HTTPS have a different set of authentication methods documented. While both protocols offer basically plain-text user and password by default (which is not really secure), there are several commonly used authentication methods for HTTP that is not sending the password as a plain text, but there are not as many (non-kerberos) options available for FTP.[9]

Something related, although not similar, is FTP's support for requesting multiple files to get transferred in parallel using the same control connection. That's of course using new TCP connections for each transfer so it'll get different performance metrics. Also, this requires that the server supports doing this sort of operation (i.e. accepting new commands while there is a transfer in progress), which many servers will not.[9]

- **Download**

HTTP does not need to download files in order to serve them to a client. This is one of the strongest benefits of using this transfer protocol over FTP. The iteration of the folders and the finding of the latest pictures will have the same process and speed, but HTTP can immediately use them, whereas FTP will need to download each one.[2]

- **Ranges / Resume**

Both FTP and HTTP support resumed transfers in both directions, but HTTP supports more advanced byte ranges.[9]

- **HTTP Chunked Encoding**

To avoid having to close down the data connection in order to signal the end of a transfer - when the size of the transfer was not known when the transfer started, chunked encoding was introduced in HTTP.[9]

During a "chunked encoding" transfer, the sending party sends a stream of [size-of-data][data] blocks over the wire until there is no more data to send and then it sends a zero-size chunk to signal the end of it.[2]

Another obvious benefit (apart from having to re-open the connection again for next transfer) with chunked encoding compared to plain closing of the connection is the ability to detect premature connection shutdowns.[9]

- **Compression**

HTTP provides a way for the client and server to negotiate and choose among several compression algorithms. The gzip algorithm being the perhaps most widely used one, with brotli being a recent addition that often compresses data even better.[9]

- **Name-based virtual hosting**

Using HTTP 1.1, you can easily host many sites on the same server and they are all differentiated by their names.[9]

- **Proxy Support**

One of the biggest selling points for HTTP over FTP is its support for proxies, already build-in into the protocol from day 1. The support is so successful and well used that lots of other protocols can be sent over HTTP these days just for its ability to go through proxies.[9]

8.2.2 Cons

- **Directory Listing**

Directory listing is not usually possible with Web connections. The information is usually hidden from the user.[76]

Directory listings over HTTP are usually done either by serving HTML showing the directory contents or by the use of WebDAV, which is an additional protocol run "over" or in addition to HTTP.[9]

- **Little Packet Verification**

HTTP was designed to be fast, but not solid. While transferring files from the server to the client's browser, some files might have a few packets lost on the way.[76]

- **One-Way System**

HTTP is a one-way system. That means that with an HTTP server, one can only transfer files in one direction, i.e. from the server to the client's browser.[76]

- **Easy target for hackers**

HTTP does not need a mandatory authentication method, unlike FTP. Furthermore, our application does not have an authentication method for simplicity and fastness. This

makes the HTTP server vulnerable to potential hackers.[76] However, we need not worry about it, because the system is only going to be used locally.

8.3 Conclusion

HTTP is a far more used for web applications and easier to server and connect to our React project. FTP is older and is being replaced with new protocols whereas, the HTTP will be there in the future.[8]

8.4 Is There a Better Solution that Outperforms Both?

Managed File Transfer (MFT) is a secure solution that encompasses all aspects of inbound and outbound data transfers while using industry-standard protocols (SFTP and FTPS) and encryption technologies (Open PGP).[67]

A managed file transfer solution, like GoAnywhere MFT can be used by organizations of all sizes for transfer needs ranging from a few dozen a week to thousands a day. It replaces the need for time-consuming manual processes and allows the ability to automate, simplify, and streamline all aspects of file transfers.[67]

9 Problems during Development

During the development of the application, there were problems, of course.

9.1 Grid Component

During the creation of the different image layouts, the first problem that was encountered was related to the `< Grid/ >` component from Material-UI. The `< Grid/ >` component was working as designed with most layouts, but with certain layouts it would break. The component in its source code divides the screen into 12 identical columns[27] to work with, so one can add components inside the grid and re-size them to their needs. However, when in the middle of the creation of one of the custom, non-standard, and non-symmetric layouts, something strange was noticed: the screen was being left blank and not automatically filled by an image. When one tries to increase the columns of 1 image e.x. 10/12 columns and all other images take 2/12, then the blank space is created. The problem occurs only with images, because they have to fit a certain size of the `< Grid/ >`. When you play with the size of the `< div/ >`, it behaves in different ways storing images vs storing text inside. The image also has to stretch vertically as it is stretched horizontally and that creates the blank space. After further inspecting the issue, it was found that the pictures could fit in this blank space area even if, pixel-wise, it was a perfect fit. However, the component behaves in a different programmable way and the use of this component from Material-UI it is not the best for the application. So, it was decided to restart the development process in another hard-coded approach so that the application works as we want it to and we can control everything.

9.2 Python Application

The python executable Windows applications created using pyinstaller, were regarded as Trojan[68] virus by Windows Defender[74]. After the creation of the applications, they

were immediately deleted at run-time by it. These are regarded as a false positives[86] and it is when the antivirus finds a legitimate file and evaluates that it is a harmful, when really it is not. A lot of people have had this issue with pyinstaller and there are many ways to fix it, one of which was reporting[75] the false positive, but that would take some time. So the only long-term solution would be excluding[73] the project folder to the Windows Defender scanning.

10 Conclusions

This web application's purpose is adapted to only be used for the observation of artificially-grown crops for the DLR Bremen EDEN ISS team. It has access to each camera, capturing their growth on a daily-basis, and uploading the images to their server. The server is located in a local machine at DLR HQ, so we were able to use that to our advantage. Since the server was an FTP server, we could not work with it efficiently and feed the pictures to our client application. However, being a local folder helped us a lot, because we made we then created an HTTP server to retrieve all the images and to serve them. In the end, we managed to find a decent solution for this problem using a recursive function approach. HTTP was the perfect server type for our web application.

The HTTP server was implemented using Node.JS. The syntax language of Node, being JavaScript, made it easier to get the hang of it quickly. The combination of the implementation of the front-end being written in JavaScript, as well as Node in the back-end, also simplified the process of serving the images to the client. Furthermore, Node is required a lot when using real-time applications. In our case it helps in the automation process when restarting the application because of how fast it feeds the images to the GUI.

The client application was implemented in JavaScript, making use of React library and its character. By using React, we were able to use its attribute and divide the application into smaller pieces. This way, it was faster and easier to work with each component and then integrate them with the GUI. The different components were controlled by one file and the application's state. Actually, the entire front-end area was implemented in a way so that it focuses on making use of the React state attribute. The state is an object, that controls the application itself and its different components. By initializing the state and changing it using the designated functions, we were also able to reflect those changes in the GUI in real-time for the users, e.x. when the timer changes. The last React attribute useful for our project was the virtual DOM. React makes use of having a copy of the actual DOM virtually and makes the application slightly faster improving its performance.

The GUI was built using Material-UI design components. The main principles were minimalism and simplicity of the application. These design principles were strictly followed in application and its components and it shows everywhere. The colorfulness of the components was one of the reasons why Material-UI was chosen in comparison to Bootstrap, but most of all it was the idea that Bootstrap websites almost look identical and we needed something different.

The layouts in the GUI were created using completely custom components. The `< Grid / >` components from either Material-UI or Bootstrap were not working properly with our application, because of their own rules and implementation. The custom components were each very simple, but a proper combination of them all can create various complex and uniform layout grids. Essentially, all they do is play around with the application's `< div / >`.

Picture shifts in the front-end were made possible using array manipulation. After the array was manipulated and then shifted, we updated the state of the application to make sure that our changes in the GUI reflected the changes that were made to be done. The timer, which was part of the application's state, was heavily integrated in the file controller and each of the 9 layouts files, to make sure that it was updated accordingly when the user made a change.

Image slideshow was a part of the GUI that also needed to have very good performance, because it holds all the images of a folder dynamically. It creates an effect of a reversed time-lapse when looking the images, since the latest picture gets shown first. This effect was achieved by removing the transition time and the transition animation that were there before. Furthermore, it is cleared from its bothersome props to have a minimalist look that makes it easier for the user to see all the details of each image.

The time-lapse section of the application also needed to have decent performance and a quick presentation of what was created. What is created is a concatenation of a certain number of images selected from the GUI. In essence it is a GIF, whose speed can also be controlled. The quality of the GIF is standard and cannot be changed, unless modified in the implementation. The reason for that is to make it stable and to not result in an error or a request timeout. The request can timeout because it uses an open-source function to create the time-lapse and return it to the client.

The client application was transformed into a proper Windows application. It opens a new independent window every time the user clicks on it. Each window can also be configured separately and they have no connections with each-other. The only common connection that they have is the HTTP server. The server was not transformed into a Windows application, because it needed to re-compile when restarted. We have a python script, which is a Windows application, that re-compiles the server. This script makes it look like it is the server that is a proper application. Furthermore, there is another python application which runs all the time and restarts the server at 6:00:00 AM.

To conclude, the application is not perfect and has a lot of room for improvement. The ideas on how to improve it can be limitless and will grow in number over time while new programming languages are released and better solutions are provided.

References

- [1] JetRuby Agency. *Pros and Cons of Bootstrap Themes*. URL: <https://expertise.jetruby.com/pros-and-cons-of-bootstrap-themes-4274c5608d3f>.
- [2] alexis. *CLI: quickly download large amount of small files over ftp*. URL: <https://superuser.com/questions/579259/cli-quickly-download-large-amount-of-small-files-over-ftp>.
- [3] D. Alimov. *Collage maker*. 2016. URL: https://github.com/delimitry/collage_maker.
- [4] G. Belani. *Should you use Bootstrap or Material Design for your web or app development project?* URL: <https://hub.packtpub.com/bootstrap-vs-material-design-for-your-next-web-or-app-development-project/>.
- [5] A. Buhler. *What Makes React so Fast?* 2020. URL: <https://medium.com/swlh/what-makes-react-so-fast-2f2ed27afb68>.
- [6] H. Catlin. *Syntactically awesome style sheets (Sass)*. URL: <https://sass-lang.com/>.
- [7] Codecademy. *React: The Virtual DOM*. 2020. URL: <https://www.codecademy.com/articles/react-virtual-dom>.
- [8] M. Cowan et al. *Difference Between HTTP and FTP*. URL: <https://techdifferences.com/difference-between-http-and-ftp.html#ComparisonChart>.
- [9] M. Cowan et al. *FTP vs HTTP*. URL: <https://daniel.haxx.se/docs/ftp-vs-http.html>.
- [10] C. Coyier. *A Complete Guide to Flexbox*. 2020. URL: <https://css-tricks.com/snippets/css/a-guide-to-flexbox/>.
- [11] Refsnes Data. *HTML <div> Tag*. 2020. URL: https://www.w3schools.com/tags/tag_div.ASP.
- [12] Refsnes Data. *HTML <fieldset> Tag*. 2020. URL: https://www.w3schools.com/tags/tag_fieldset.asp.
- [13] Refsnes Data. *HTML <legend> Tag*. 2020. URL: https://www.w3schools.com/tags/tag_legend.asp.
- [14] Refsnes Data. *JavaScript Array push() Method*. 2020. URL: https://www.w3schools.com/jsref/jsref_push.asp.
- [15] Refsnes Data. *JavaScript String substring() Method*. 2020. URL: https://www.w3schools.com/jsref/jsref_substring.asp.
- [16] Refsnes Data. *What is JSON?* 2020. URL: https://www.w3schools.com/whatis/whatis_json.asp#:~:text=JSON%5C%20stands%5C%20for%5C%20JavaScript%5C%200bject,describing%5C%22%5C%20and%5C%20easy%5C%20to%5C%20understand.
- [17] DLR. *EDEN Initiative*. 2020. URL: <https://www.dlr.de/irs/en/desktopdefault.aspx/tabid-11286/#gallery/35706>.
- [18] DLR. *Ground Demonstration of Plant Cultivation Technologies for Safe Food Production in Space*. 2020. URL: <https://eden-iss.net/>.
- [19] H. Ekhtiyar, M. Sheida, and M. Amintoosi. "Picture Collage with Genetic Algorithm and Stereo vision". In: *Computer Science Issues, Vol. 8* (2011).

- [20] ewerybody. *Python check if a process is running or not*. 2015. URL: <https://stackoverflow.com/questions/7787120/python-check-if-a-process-is-running-or-not>.
- [21] Inc. Facebook. *Button*. 2020. URL: <https://material-ui.com/components/buttons/#contained-buttons>.
- [22] Inc. Facebook. *Checkbox*. 2020. URL: <https://material-ui.com/components/checkboxes/#checkboxes-with-formgroup>.
- [23] Inc. Facebook. *Dialog*. 2020. URL: <https://material-ui.com/components/dialogs/>.
- [24] Inc. Facebook. *FormControl API*. 2020. URL: <https://material-ui.com/api/form-control/#formcontrol-api>.
- [25] Inc. Facebook. *FormControlLabel API*. 2020. URL: <https://material-ui.com/api/form-control-label/#formcontrollabel-api>.
- [26] Inc. Facebook. *FormGroup API*. 2020. URL: <https://material-ui.com/api/form-group/#formgroup-api>.
- [27] Inc. Facebook. *Grid*. 2020. URL: <https://material-ui.com/components/grid/#grid>.
- [28] Inc. Facebook. *IconButton API*. 2020. URL: <https://material-ui.com/api/icon-button/#iconbutton-api>.
- [29] Inc. Facebook. *Layout with Flexbox*. 2020. URL: <https://reactnative.dev/docs/flexbox/>.
- [30] Inc. Facebook. *ListItem API*. 2020. URL: <https://material-ui.com/api/list-item/#listitem-api>.
- [31] Inc. Facebook. *ListItem API*. 2020. URL: <https://material-ui.com/api/list-item-text/#listitemtext-api>.
- [32] Inc. Facebook. *ListItem API*. 2020. URL: <https://material-ui.com/api/menu/>.
- [33] Inc. Facebook. *ListItem API*. 2020. URL: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Array/map.
- [34] Inc. Facebook. *Material Icons*. 2020. URL: <https://material-ui.com/components/material-icons/#material-icons>.
- [35] Inc. Facebook. *Menus*. 2020. URL: <https://material-ui.com/components/menus/#selected-menus>.
- [36] Inc. Facebook. *Minimizing Bundle Size*. 2020. URL: <https://material-ui.com/guides/minimizing-bundle-size/#minimizing-bundle-size>.
- [37] Inc. Facebook. *Modal*. 2020. URL: <https://material-ui.com/components/modal/>.
- [38] Inc. Facebook. *Progress*. 2020. URL: <https://material-ui.com/components/progress/#circular-indeterminate>.
- [39] Inc. Facebook. *React.Component*. 2020. URL: <https://reactjs.org/docs/react-component.html#componentdidmount>.
- [40] Inc. Facebook. *React.Component*. 2020. URL: <https://reactjs.org/docs/react-component.html#componentdidupdate>.

- [41] Inc. Facebook. *React.Component*. 2020. URL: <https://reactjs.org/docs/react-component.html#componentwillunmount>.
- [42] Inc. Facebook. *State and Lifecycle*. 2020. URL: <https://reactjs.org/docs/state-and-lifecycle.html>.
- [43] Inc. Facebook. *Text Field*. 2020. URL: <https://material-ui.com/components/text-fields/#form-props>.
- [44] Inc. Facebook. *Toolbar API*. 2020. URL: <https://material-ui.com/api/toolbar/>.
- [45] Inc. Facebook. *Typography*. 2020. URL: <https://material-ui.com/components/typography/#component>.
- [46] I. Feoktistov. *Why and when to use Node.js?* 2020. URL: <https://relevant.software/blog/why-and-when-to-use-node-js/>.
- [47] OpenJS Foundation. *Console*. 2020. URL: https://nodejs.org/api/console.html#console_console_log_data_args.
- [48] OpenJS Foundation. *express - Fast, open, uncomplicated web framework for Node.js*. 2020. URL: <https://expressjs.com/>.
- [49] OpenJS Foundation. *express()*. 2020. URL: <http://expressjs.com/api.html#express.static>.
- [50] OpenJS Foundation. *express()*. 2020. URL: <http://expressjs.com/en/api.html#res.set>.
- [51] OpenJS Foundation. *express()*. 2020. URL: http://expressjs.com/en/5x/api.html#app.listen_path_callback.
- [52] OpenJS Foundation. *express()*. 2020. URL: <http://expressjs.com/en/5x/api.html#res.json>.
- [53] OpenJS Foundation. *File system*. 2020. URL: https://nodejs.org/api/fs.html#fs_fs_readdirsync_path_options.
- [54] OpenJS Foundation. *File system*. 2020. URL: https://nodejs.org/api/fs.html#fs_dirent_isdirectory.
- [55] OpenJS Foundation. *File system*. 2020. URL: https://nodejs.org/api/fs.html#fs_dirent_isfile.
- [56] OpenJS Foundation. *Provide static files in Express*. 2020. URL: <https://expressjs.com/de/starter/static-files.html>.
- [57] OpenJS Foundation. *Use middleware*. 2020. URL: <http://expressjs.com/de/guide/using-middleware.html#middleware-verwenden>.
- [58] OpenJS Foundation. *Writing middleware for use in Express apps*. 2020. URL: <http://expressjs.com/en/guide/writing-middleware.html#writing-middleware-for-use-in-express-apps>.
- [59] Python Software Foundation. *os — Miscellaneous operating system interfaces*. 2020. URL: <https://docs.python.org/3/library/os.html#module-os>.
- [60] Python Software Foundation. *subprocess — Subprocess management*. 2020. URL: <https://docs.python.org/3/library/subprocess.html>.
- [61] M. Gajda. *Why use Node.js web development? Scalability, performance and other benefits of Node based on famous web applications*. 2020. URL: <https://tsh.io/blog/why-use-nodejs/>.

- [62] Inc. GitHub. *Electron Packager*. 2020. URL: <https://github.com/electron/electron-packager>.
- [63] Guru99. *express - Fast, open, uncomplicated web framework for Node.js*. 2020. URL: <https://www.guru99.com/node-js-express.html#:~:text=The%5C%20express%5C%20framework%5C%20is%5C%20the,for%5C%20developing%5C%20Node%5C%20js%5C%20applications.&text=js%5C%20framework%5C%20and%5C%20helps%5C%20in,based%5C%20on%5C%20the%5C%20request%5C%20made..>
- [64] M. Hamedani. *React Lifecycle Methods – A Deep Dive*. 2015. URL: <https://programmingwithmosh.com/javascript/react-lifecycle-methods/#:~:text=What%5C%20are%5C%20React%5C%20lifecycle%5C%20methods,birth%5C%2C%5C%20growth%5C%2C%5C%20and%5C%20death..>
- [65] Stack Exchange Inc. *Developer Survey Results*. 2019. URL: <https://insights.stackoverflow.com/survey/2019#technology--programming-scripting-and-markup-languages>.
- [66] Stack Exchange Inc. *Developer Survey Results*. 2020. URL: <https://insights.stackoverflow.com/survey/2019#technology--other-frameworks-libraries-and-tools>.
- [67] H. Kath. *Comparing Transfer Methods: HTTP vs. FTP*. URL: <https://www.goanywhere.com/blog/comparing-transfer-methods-http-vs-ftp>.
- [68] AO Kaspersky Lab. *What is a Trojan Virus?* 2020. URL: <https://www.kaspersky.com/resource-center/threats/trojans>.
- [69] L. Leandro. *React Responsive Carousel*. 2020. URL: <https://github.com/leandrowd/react-responsive-carousel>.
- [70] W. Maj. *React lifecycle methods diagram*. 2020. URL: <https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/>.
- [71] S. Martin. *Grid*. 2018. URL: <https://torquemag.io/2018/06/why-millions-of-developers-use-javascript-for-web-application-development/>.
- [72] S. Martin. *Grid*. 2020. URL: <https://medium.com/javarevisited/why-is-javascript-still-important-in-developing-modern-web-apps-67fcd30d7ad6>.
- [73] Microsoft. *Add an exclusion to Windows Security*. 2020. URL: <https://support.microsoft.com/en-us/windows/add-an-exclusion-to-windows-security-811816c0-4dfd-af4a-47e4-c301afe13b26>.
- [74] Microsoft. *Stay protected with Windows Security*. 2020. URL: <https://support.microsoft.com/en-us/windows/stay-protected-with-windows-security-2ae0363d-0ada-c064-8b56-6a39afb6a963>.
- [75] Microsoft. *Submit a file for malware analysis*. 2020. URL: <https://www.microsoft.com/en-us/wdsi/filesubmission>.
- [76] Mirjana. *FTP vs HTTP*. 2011. URL: <http://www.worldofintegration.com/content/ftp-vs-http>.
- [77] Mozilla and individual contributors. *Date*. 2020. URL: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Date.
- [78] Mozilla and individual contributors. *String.prototype.endsWith()*. 2020. URL: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/String/endsWith.

- [79] Mozilla and individual contributors. *String.prototype.includes()*. 2020. URL: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/String/includes.
- [80] E. Newport. *List all files in a directory in Node.js recursively in a synchronous fashion*. 2013. URL: <https://gist.github.com/kethinov/6658166>.
- [81] Inc. O'Reilly Media. *Chapter 2. Working with React Native*. 2020. URL: <https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch02.html>.
- [82] N. Ovchinnikova. *Bootstrap Vs. Material-UI. Which One To Use For The Next Web App?* URL: <https://flatlogic.com/blog/bootstrap-vs-material-ui-which-one-to-use-for-the-next-web-app/#:~:text=Bootstrap%5C%20is%5C%20a%5C%20powerful%5C%20CSS,both%5C%20for%5C%20mobile%5C%20and%5C%20desktop.&text=Material%5C%20UI%5C%20is%5C%20a%5C%20React,principles%5C%20of%5C%20the%5C%20Material%5C%20design..>
- [83] N. Pandit. *What and Why React.js*. 2020. URL: <https://www.c-sharpcorner.com/article/what-and-why-reactjs/#:~:text=js%5C%3F-,React.,specifically%5C%20for%5C%20single%5C%20page%5C%20applications.&text=React%5C%20allows%5C%20developers%5C%20to%5C%20create,fast%5C%2C%5C%20scalable%5C%2C%5C%20and%5C%20simple..>
- [84] Tutorials Point. *Data Structure and Algorithms - Tree*. 2020. URL: https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm.
- [85] Sharpened Productions. *Runtime*. 2020. URL: <https://techterms.com/definition/runtime#:~:text=Runtime%5C%20is%5C%20the%5C%20period%5C%20of,most%5C%20often%5C%20in%5C%20software%5C%20development..>
- [86] Panda Security. *False positives – What are they?* 2020. URL: <https://www.pandasecurity.com/en/mediacenter/security/false-positives-what-are-they/#:~:text=A%5C%20false%5C%20positive%5C%20occurs%5C%20when,scanning%5C%20or%5C%20analysis%5C%20of%5C%20behavior..>
- [87] PyInstaller Development Team. *PyInstaller*. 2020. URL: <https://www.pyinstaller.org/>.
- [88] ttppp. *Mountain Tapir Collage Maker*. 2016. URL: <https://mountain-tapir.readthedocs.io/en/latest/readme.html>.
- [89] WDIS. *Why does FTP Suck?* URL: <https://whydoesitsuck.com/why-does-ftp-suck/>.
- [90] C. West and G. Franko. *GifShot*. 2020. URL: <https://yahoo.github.io/gifshot/>.