



Università degli Studi di Ferrara

University OF Ferrara

BACHELOR of Science in COMPUTER SCIENCE

Implementation of a Progressive Web Application in VueJS for stock management

Rapporteur:

Prof. Giacomo Piva

Undergraduate:

Ilaria Zucchini

ACADEMIC YEAR 2023–2024

Index

	Page
Introduction	4
1 What is a PWA?	7
1.1 Description	7
1.2 Aspects	8
2 Technologies used	11
2.1 Front-end design and development	12
2.1.1 Figma	12
2.1.2 HTML	12
2.1.3 CSS	13
2.1.4 Bootstrap 5	13
2.1.5 JavaScript	14
2.2 Manifest and Service Worker	14
2.2.1 Manifest	14
2.2.2 Service Worker	16
2.3 API	16
2.4 Testing	17
2.4.1 Running on localhost	17
2.4.2 Chrome DevTools	17

3	PWA Implementation	19
3.1	App.vue	19
3.2	routers.js	20
3.3	Login	22
3.4	Customers	24
3.5	Menu and profile card	27
3.5.1	Menu	27
3.5.2	Profile card.....	28
3.6	Home and open orders.....	29
3.7	Creating and Accepting an Order.....	32
3.8	Modal	35
4	Results and Conclusion	39

Introduction

Nowadays, access to the Internet and its use are indispensable for each of us, in every sphere and at all times.

In the private sphere, for example, the Internet is used for entertainment (video games, music, etc.), to keep in touch with people, to pay for services, to stay informed, to buy products and so on.

In the business environment, the Internet is used to stay in touch with suppliers and customers. As we move more and more towards a digitalised world, it is necessary for companies to be able to communicate with customers and suppliers immediately and automatically. For this reason, many companies rely on software that, for example, sends an automatic e-mail when an order is placed, informing the customer that their order has been sent.

In addition, the Internet is now also indispensable in order to give employees the possibility of accessing their information on a device directly from their workstations, without having to resort to binders or printed sheets located in other rooms, perhaps far away from them. Access to this information in an immediate manner certainly plays a crucial role in optimising time within companies.

But how can employees be given access to this information?

A solution for companies can be the use of a Progressive Web App (PWA), which allows it to be used on computers and mobile devices, adapting to the screen on which it is opened so that it is always neat and intuitive; it can be accessed from the browser although there is also the possibility of installing it on the device.

The following text will deal precisely with the development of a Progressive Web App (PWA) for use by authorised technicians for the creation and acceptance of orders.

What is a PWA?

Before delving into the practical aspects, it is important to clarify what a PWA is. PWAs (Progressive Web Apps in full) are applications built using web technologies. These applications, while residing on browsers, manage to offer a user experience that is very similar to that of a native application installed on a device.

In addition, thanks to their versatility, they can run on various types of devices and on different browsers from a single code base, allowing them to be used by a large audience.

1.1 Description

PWAs, unlike the traditional native applications we find installed on devices, are a hybrid between these and traditional web pages. When implementing this type of application, an attempt is made to make the most of the potential offered by browsers, combining it with the convenience and advantages of native applications for use on mobile devices.

The term '*Progressive*' implies that this type of application can be run on any type of device and on any browser.

PWAs are installable applications; in fact, the browser can propose the user to install the PWA on the device.

Once installed, it will be displayed on the main page of the device via its icon. In this way, the user will perceive the newly installed PWA just like a native application.

1.2 Aspects

PWAs also have several aspects to take into account:

- Progressive: as explained above, PWAs work for every user on every browser and on every device.
- Responsive: they adapt to the screen size of the device in which they are used (smartphone, tablet, desktop, etc.), which allows optimal visualisation of the application on both small and large screens.
- Secure: they run on HTTPS¹ protocol, which prevents information or content from being altered.
- Installable: installation can be done either from the app store (in case the manufacturer publishes it) or directly from the web, where a pop-up notification will appear when opening the application from the browser asking the user if they want to install it.

Once installed, its behaviour is similar to a native application, displaying it in full screen while the address bar is hidden.

- Updated: these applications, residing on the web, can be updated in the background, allowing users to always have access to the latest version.

¹ HyperText Transfer Protocol over Secure Socket Layer

- Integrated: they can be integrated with operating systems, which implies the possibility of functionality through the use of API², e.g. the PWA will be able to access the device's camera, GPS or microphone, which can also receive push notifications.
- Offline: they also work offline or in the background thanks to the presence of Service Workers; this ensures continuity for the user.
- Linkable: they are easily shared via URL³. [7]

² Application programming interface

³Uniform Resource Locator

Technologies used

Developing this *Progressive Web App* was possible by integrating several selected technologies to ensure an intuitive interface and a smooth user experience. This chapter describes the main technologies adopted and their role in the project.

The interface design was carried out with Figma, while the development was carried out with *HTML*, *CSS*, *Bootstrap* and *JavaScript*, using the *Vue.js* framework. For communication with the API, *Axios* was used, while the implementation of the PWA functionality was made possible thanks to *Manifest* and *Service Worker*. Finally, *Chrome DevTools* was used for testing.

Vue.js is a framework, i.e. a structure used to facilitate software development. Instead of the classic DOM¹, *Vue.js* uses a *Virtual DOM* so that only the necessary parts of a page can be updated without having to reload everything.

¹ Document Object Model

2.1 Front-end design and development

2.1.1 Figma

Figma is a UI/UX design software, used to add blocks and colours, defining the structure and visual appearance of different pages. Although no advanced features were used, Figma provided a useful environment to visualise and organise design ideas before moving on to actual development.

At the end of the project, the PWA looks completely different from the first draft, as some parts have been modified to be lighter and more pleasant in the eyes of users.

2.1.2 HTML

HyperText Markup Language is the most basic component of the Web, defining the meaning and structure of web pages. Other technologies besides HTML are generally used to describe the appearance (CSS) or functionality/behaviour (JavaScript) of a web page. "Hypertext" refers to links connecting web pages to each other, either within a single website or between websites.

HTML uses 'markup' to annotate text, images and other content to be displayed in a web browser. HTML markup includes 'elements' such as `<head>`, `<title>`, `<body>`, `<header>`, `<footer>`, `<article>`, `<section>`, `<p>`, `<div>`, and many others.

An HTML element is separated from other text in a document by 'tags', which consist of the name of the element surrounded by `<` and `>`. The name of an element within a tag is not case-sensitive. That is, it may be written in upper case, lower case or a combination. However, the convention and recommended practice is to write tags in lower case. [5]

2.1.3 CSS

Cascading Style Sheets is a language that describes how HTML elements are to be displayed (colours, layouts, fonts, etc.). For example:


```
1 p{  
2   colour : blue ;  
3   font - size : 16  
4   px;  
}
```

This will make the text in the `<p>` (paragraph) tags blue and 16 pixels in size. *Cascading* refers to the fact that language rules follow a hierarchy. CSS is one of the main languages of the web and is standardised according to the W3C² specification.^[4]

2.1.4 Bootstrap 5

Bootstrap was born in 2010. It is an open source front-end framework and is a collection of graphical, stylistic and layout tools that can be used for the development of responsive web applications. To use it, all you have to do is insert the default class you need within the HTML tags. An example:

```
1 <div class =" bg - dark rounded ">  
2   <p class =" text - white fw - bold "> Lorem  
   ipsum dolor sit amet . </ p>  
3 </ div>
```



Lorem ipsum dolor sit amet.

Figure 2.1: Result of class utilisation

² W3C: World Wide Web Consortium

In the <div> tag, the classes added:

- *bg-dark* : sets the background black.
- *rounded* : set the rounding of edge corners.

Classes have been added to the <p> tag for:

- *text-white*: set text colour.
- *fw-bold* : increase text thickness.

2.1.5 JavaScript

JavaScript is a lightweight, interpreted (*just-in-time- compiled*) programming language. It is mainly known for use in webpage programming, but can also be used in non-browser environments.

JavaScript, moreover, is a language that supports different types of programming, one example being object-oriented programming.[6]

2.2 Manifest and Service Worker

PWAs rely on two fundamental files that guarantee their operation: the manifest.json and the serviceWorker.js.

2.2.1 Manifest

The manifest.json is a JSON file containing basic information about the PWA (name, start url, background colour, etc.) and how it should behave when installed, it also contains the icons (in various sizes) and (not necessarily) screenshots of the application.

Below is the manifest.json of the PWA, the name present is not the original name, but a fictitious name.

```
1 {  
2   "name ":    " Example PWA  
3   ", " short_name ": "  
   Example ",
```

```

4  "theme_colour ": "# 022970". ,
5  "icons ': [
6  {
7      "src ": "./ img / icons / favicon 48 x 48
8      . ico ", " sizes ": "48 x48 ",
9      "type ": " image / x- icon ".
10 },
11 {
12     "src ": "./ img / icons / favicon 96 x 96
13     . png ", " sizes ": "96 x96 ",
14     "type ": " image / png ".
15 },
16 {
17     "src ": "./ img / icons / favicon 192 x 192
18     . png ", " sizes ": "192 x192 ",
19     "type ": " image / png ".
20 },
21 {
22     "src ": " favicon . ico ",
23     "sizes ": "256 x256
24     ", " type ": " image /
25     x- icon ".
26 },
27 {
28     "src ": "./ img / icons / favicon 512 x 512
29     . png ", " sizes ": "512 x512 ",
30     "type ": " image / png ".
31 }
32 ],
33 "start_url ": "/",
34 " display ": " standalone
35 ",
36 "background_colour ": "#
37 022970". , " screenshots ": [
38 {
39     "src ": "./ img / screenshots / Desktop .
40     png ", " sizes ": " 3061 x1759 ",
41     "type ": " image / png
42     ", " platform ": "
43     wide ".
44 }
45 ]
46 }

```

As can be seen, there are icons in all the necessary sizes, there is the start URL which will be the first URL opened when the application starts, there is the colour of the theme and finally there is a screenshot of the application.

2.2.2 Service Worker

The `serviceWorker.js` is a JavaScript file, acts as a remote server and is located between the PWA, the browser and the network. Its task is to intercept network requests, make decisions based on network availability and update the resources residing on the server. Service workers are asynchronous events and are executed in a separate manner from the main application thread.

Finally, it allows background synchronisation of the API³ and access to the push notifications. [8]

2.3 API

Application Programming Interfaces, or simply APIs, are a set of procedures that enable communication between products and services without necessarily having to know the implementation of the product or service with which they are communicating, thus saving both time and money.

By defining the mode of interaction between applications, APIs enable the sending and receiving of data.[2]

Within the project, API calls were implemented through the use of Axios, a JavaScript library that enables connection with the backend API and handling requests via the HTTP protocol.

Being promise-based, Axios supports the implementation of asynchronous code, so the programme will be able to send the API request without having to interrupt the execution of other operations.

³ API: Application Programming Interface

2.4 Testing

Two tools were used to test the functioning of the PWA: the execution on localhost and Chrome DevTools.

2.4.1 Execution on localhost

For this type of execution, a local development server is started up via the `npm run serve` command (on Vue.js), which allows the application and the changes made to be viewed in real time .

2.4.2 Chrome DevTools

Chrome DevTools is a set of tools for web developers directly within Google Chrome.

Through this tool, it is possible to instantly modify parts of the page you are currently viewing. The changes are not made permanently, and when the page is reloaded, the changes are cancelled if they have not previously been reflected in the code in the editor being used (in this case Visual Studio Code).

You can open Chrome DevTools by right-clicking and selecting 'inspect' or with the key sequence: `Ctrl+Shift+C`, if you are on a Mac just press `Option` instead of the `Shift` key.^[1]

Implementation of the PWA

The PWA that is the subject of this thesis consists of several files, each of which consists of two or three parts: `<template></template>` for the HTML code, `<script></script>` for JavaScript code and `<style></style>` (not always present) for CSS code.

3.1 App.vue

App.vue is the base file of the *Progressive Web App*, each component (in Vue, what is usually known as a page is actually a Vue component) to be displayed, is loaded by passing through this file.

A very short piece of code is given:

```
1 <template>
2   <router-view />
3 </template>
```

`<router-view/>` is the component of Vue.js that allows the display of the component's content based on the current URL.

For example, if the user visits the Home page, the Home.vue component will be redirected by Vue Router within the tag, which in turn will load the page and display it.

App.vue presents an API within the <script> tag, its call starts automatically and performs a check on the server, ensuring that it is online.

Finally, the <style> tag contains the CSS code for almost the entire application including the font for the entire application.

3.2 routers.js

Within the routers.js file, routes are created to reach the application components.

Each component is imported at the beginning of the file and each of them is given a name and a path, which makes it possible to realise the various necessary redirects between pages.

The path will be displayed in the address bar if the PWA is opened in a browser.

An instance is also created that manages browser history and URL changes, which allows the page not to be completely reloaded, but only the dynamic elements to be reloaded. The method used for creating the instance is createWebHistory(). The code of the *routers.js* file is shown:

```
1 import Home from "../components / Home . vue ";
2 import Login from "../components / Login . vue
3 "; import Order from "../components / Order .
4 vue "; import Accept from "../components /
5 Accept . vue "; import Client from "../
6 components / Client . vue ";
7 import Open Order from "../components / Open Order . vue ";
8 import { create Router , create Web History } from ' vue - router
9 ' const routes = [
10 {
11   path : '/',
    redirect : '/ login
    '.
```

```

12 },
13 {
14   name : 'Login',
15   component : Login ,
16   path : '/ login',
17 },
18 {
19   name : 'Home',
20   component : Home ,
21   path : '/ home',
22 },
23 {
24   name : " Order ",
25   component : Order ,
26   path : "/ order ",
27 },
28 {
29   name : " Accept ",
30   component : Accept ,
31   path : "/ accept ",
32 },
33 {
34   name : " Client ",
35   component : Client ,
36   path : "/ client ",
37 },
38 {
39   name : " Open Order ",
40   component : Open Order ,
41   path : "/ open_order ".
42 }
43 ];
44
45 const router= create Router ({
46   history : create Web History () ,
47   routes
48 });
49
50 export default router ;

```

Within routes you can see that the first element is:

```
{ path: '/', redirect: '/login' }
```

This is due to the fact that when opening, the *Progressive Web App* will try to open '/', but finding no possible redirection will redirect to '/login'.

3.3 Login

The login page is visually very clean, presenting a form containing input for entering email and password.

Once you have entered the required data, you can check that you have entered your password correctly via the dedicated checkbox.

The checkbox via the 'showPassword()' function changes the tag type from 'password' to 'text' and vice versa when selected or deselected.

```
1 show Password () {  
2   var pw= document . get Element By Id (' password ');  
3   if ( pw. type === "password") {  
4     pw. type = "text ";  
5   } else {  
6     pw. type = "password";  
7   }  
8 }
```

After that, pressing enter on the keyboard or clicking the login button calls the API, which, in the event of a positive response, returns the user's data and creates a token that allows access to the application and redirects the user to the customer view page.

In this step, the returned value is checked, which must be equal to 200, and it is also checked that the returned array is not empty. The check is performed via the method:

`Object.keys().length`

Object.keys() is used because the data initially returned is of type object, but an array is required to perform this check using .length.

This method then transforms the data into an array.

Finally, the length of the latter is checked with, precisely, .length.

The user data is stored in the local browser memory via the JavaScript method JSON.stringify(), which converts JavaScript data into JSON data.

Converting them is necessary to save them and consequently to allow the PWA to use them to maintain access and to call subsequent APIs.

Below is the code for the controls and data storage:

```
1  if ( response . status== 200 && Object . keys ( response . data ) .
    length
2    > 0) {
    local Storage . set Item ( ' token ', JSON . stringify ( response .
3      data . token ));
    local Storage . set Item ( ' id', JSON . stringify ( response .
4      data . user . id));
    local Storage . set Item ( ' company_name ', JSON . stringify (
5      response . data . user . company_name ));
    local Storage . set Item ( " referent_name ", JSON . stringify (
6      response
        .      data .      user .      referrer_name      ));
7    local Storage . set Item ( ' email ', JSON . stringify (
    response . data .
8      user . email ));
9    local Storage . set Item ( " logo ", JSON . stringify ( response .
10   data . user
11     . logo ));
    this . $router . push ({ name : ' Client ' });
  } else {
```

In the event of a negative response, an error message is returned which the user will display in red between the checkbox and the login button. The error message will read 'Incorrect e-mail or password'.

Should the user try to log in without entering one of the two fields or only partially type in their e-mail, no error message will be returned, but a tooltip will appear informing the user of the need to fill in the field. The tooltip is automatically generated by inserting (in the HTML code) 'required' at the end of the `<input>` tag.

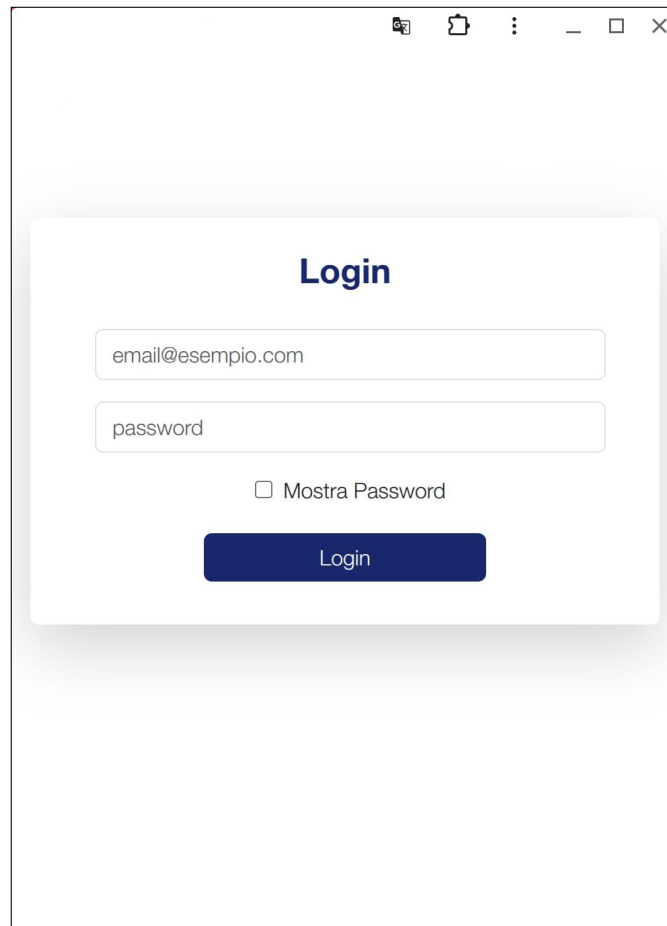


Figure 3.1: Login page

3.4 Customers

Once on this page, the user will be able to make the choice of the customer for whom he wants to perform a transaction. When the page loads, an API will be called automatically to retrieve the user's customer data

just logged in.

Upon receiving a positive response, the only information that will be displayed by the user will be the name of the customers.

The names will be displayed in a list, generated with the Vue.js directive, `v-for`.

This directive works in a very similar way to a for loop, but with the advantage of being reactive: each time the data changes, Vue automatically updates the DOM without the need for manual intervention.

In this way, the interface always remains synchronised with the available data. Below is some of the HTML code:

```
1 <li v- for =" customer in customers " : key =" customer . id" class ="
  client
2   - list - group - item ">
    <a @ click =" choose ( customer )" class =" text - decoration - none
3     "> {{ customer . company_name }}
4     <i class =" bi - chevron - right " ></ i>
5   </ a>
</ li>
```

The function called on clicking on the customer's name is `choose(customer)` (`@click`, in Vue.js equivalent to 'on click') and handles the selection of the user by performing two main operations.

First, it saves both the ID and the name of the selected client in the local browser memory, ensuring that this information can be retrieved and used later.

Immediately afterwards, it redirects to the home page.

The use of:

`<i class="bi-chevron-right"></i>`

allows for an arrow pointing to the right immediately after the customer's name, which improves the visual clarity of the interface and intuitively communicates that clicking on the name leads to a specific action. Finally, the use of an arrow pointing to the right is common in user interfaces, making the experience more

fluid and familiar to the user.

So to summarise: the user chooses the customer for whom he wants to perform an action, selects it and is redirected to the home page, during this loading the name and the customer identification code are saved in the browser's memory.

The user interface that will be displayed:

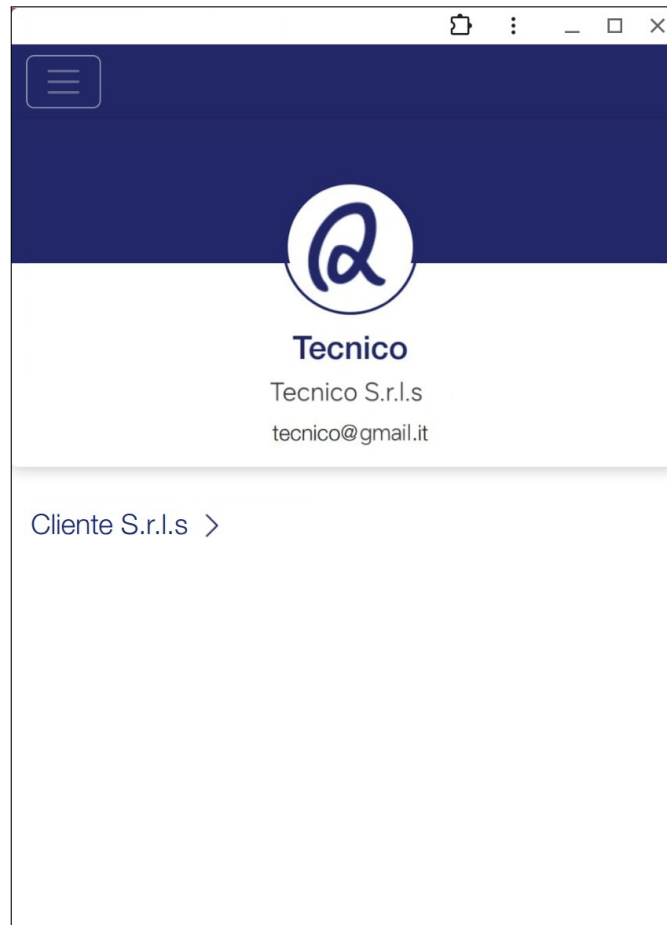


Figure 3.2: Customer choice

3.5 Menu and profile card

The header of the PWA consists solely of the menu and profile card of the user implemented via HTML and Bootstrap 5.

3.5.1 Menu

The menu contains links to the Customer page, open orders (within the Home page) and logout.

As for logging out, once the user has clicked in the menu, the appropriate API will be called, which will check the user's token, downloading it from the browser's local memory, and upon receiving a positive response, the data will be deleted from the browser's memory by the method:

`localStorage.clear()`

and the user will be redirected to the login page.

The menu has the particularity of being responsive to the size of the screen. In fact, if viewed from the desktop, i.e., from a large screen, the components will be shown one after the other, whereas if viewed from a smaller screen (such as a smartphone or tablet) or by simply resizing the screen from the desktop, the menu will be enclosed in a drop-down menu.

In technical jargon, the class `navbar-expand-lg` indicates that the menu on large screens will be expanded while it will be collapsed on smaller screens, ensuring an optimal user experience.

In the menu there is also an icon before each string, which has been inserted as the bootstrap class of the `<a>` tag, this tag (located within `list`) will allow redirection to the chosen page.

3.5.2 Profile card

The profile card was implemented with the dedicated Bootstrap classes: card, profile-card, profile-header and profile-body. The code is given below:

```
1 <div class =" card profile - card ">
2 <div class =" profile - header " >&nbsp; ; </ div>
3 <div class =" profile - body ">
4 <div class =" image - area ">
5 <img : src =" logo " alt =" Logo " height =" 100 " width =" 100 " />
6 </ div>
7 <div class =" content - area ">
8 <h3 class =" my -2" >{{ referent_name }} </ h3>
9 <p class =" my -1" >{{ company_name }} </ p>.
10 <p >{{ email }} </ p>.
11 </ div>
12 </ div>
13 </ div>
```

Within `referrer_name`, `company_name` and `email`, data are saved by downloading them from the browser memory using the method:

`localStorage.getItem().replace(/"/g, '')`

`replace()` was used because on the browser's local memory, data is stored in inverted commas (") and with these, it is also downloaded from memory. The above function replaces the inverted commas with the value inside "", which in this case is empty, so it simply removes the inverted commas.

The letter 'g' indicates 'global', the inverted commas will then be searched for and removed throughout the string.

The best solution for the implementation of these two elements (menu and profile card) fell upon the creation of a Header component.

This choice was made with a view to convenience and order within the code.

This component, in fact, once created is simply called up in the other components via its own `<Header />` tag.

The expanded menu display on small screens and the profile card display:

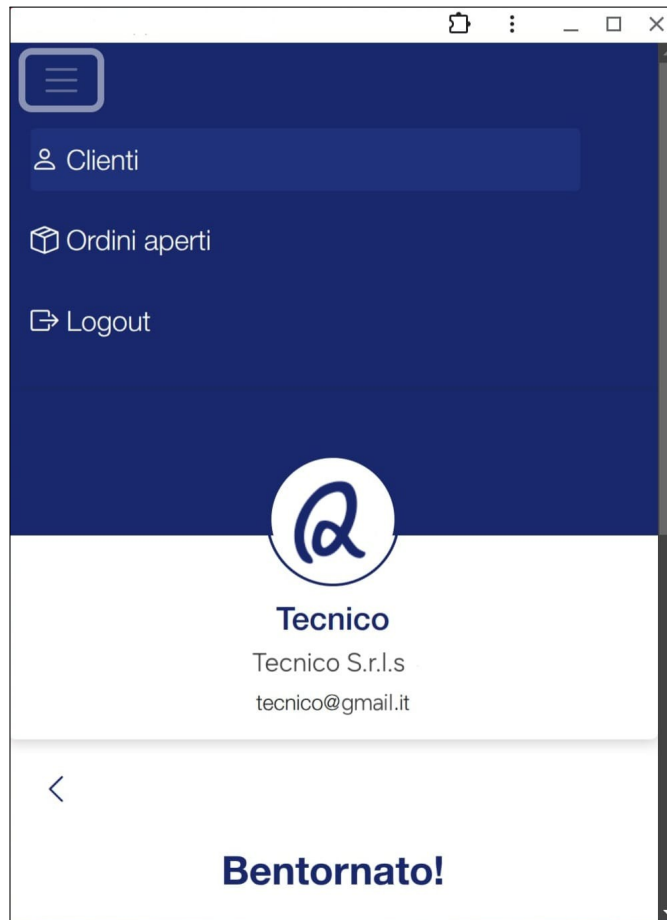


Figure 3.3: The expanded menu and profile card

As can be seen, the profile card shows the information of the logged in user: the company logo, the contact person's name, the company name and finally the email of the contact person.

3.6 Home and open orders

On the Home page, you can decide on the operation to be performed and view open orders by scrolling to the bottom of the page.

Here, you can also be sure that you have selected the correct customer, as there is a subheading <h2> showing the name of the selected customer.

The method used to display the customer's name is:

```
JSON.parse(localStorage.getItem('selectedCostumer'))
```

then, the name is taken from the local browser memory (where it was saved just before) and via JSON.parse() is transformed into a JavaScript value.

There are two operations the user can choose from:

- the creation of a new order
- acceptance of an order

The implementation of the buttons for accessing operations was done via the HTML tag <button> and clicking on them will redirect you to the creation or acceptance page, depending on which button you clicked.

To display the open orders, the token and the customer id are taken from the local memory of the browser and stored in an array, after which the API call is made, which needs this data to return the open orders of that particular customer.

The code that allows open orders to be displayed on the home page:

```
1  async open Orders () {
2    let token= local Storage . get Item (' token '). replace (/"/ g,
3    ''); try {
4      let response= await axios . get (' https :// api / ${
5        this . customer . id }/ orders ', {
6        headers : {
7          ' authorisation ': ' Bearer ${ token
8          }, ' accept ': ' application / json
9          },
10         });
11     console . log ( response );
12     this . orders= response . data . orders ;
13   } catch ( error ) {
14     console . warn (" Error : ",
15     error )
16   }
```

14
15

```
}  
}
```

On the home page, it will only be possible to see the ID of open orders. In order to view the content of an order, it will be necessary to click on the relevant ID in order to be redirected to the dedicated openOrder page, where it will be possible to view the order in full, thus displaying the order details, such as, for example, items missing from completion. If there are no open orders, the phrase 'No open orders' will be displayed.

As a conclusion to the description of this page, it is useful to inform you that in the top left-hand corner (below the profile card, which you are reminded is present on all pages except the login page) there is an arrow that allows you to return to the previous page.

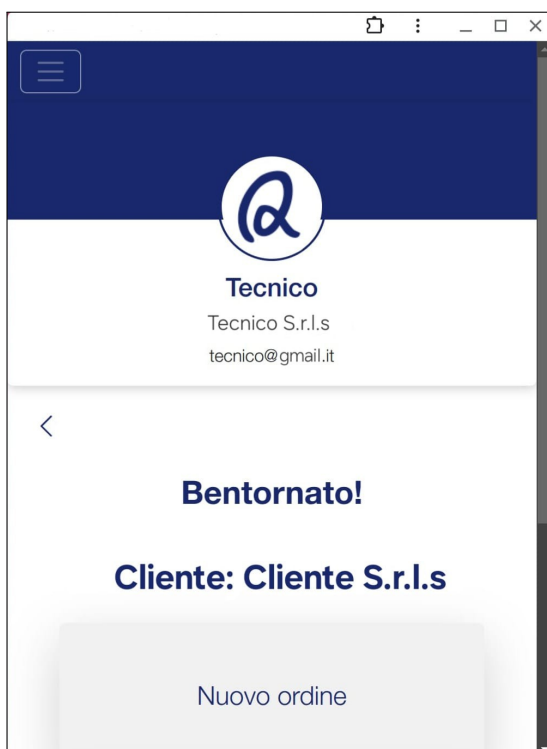


Figure 3.4: P agina Home 1

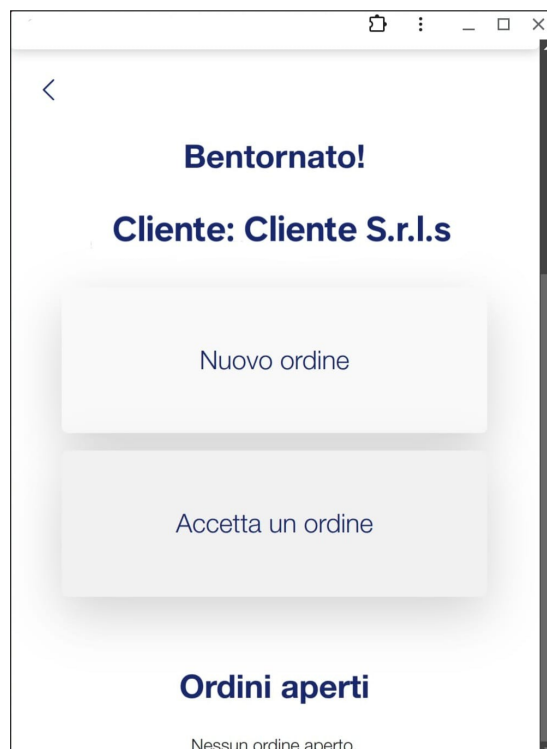


Figure 3.5: P agina Home 2

3.7 Creating and accepting an order

Once you have decided on the operation to be performed, you are redirected to one of two pages, the creation of a new order or the acceptance of an order.

Both pages have an identical user interface, so in both, there is: the page title 'New Order' or 'Accept Order', a bulleted list with an explanation of how to perform the chosen operation correctly, a text area in which to enter the product barcodes, a button to send or accept the order, and finally the button to delete the entire contents of the text area.

The entry of product codes to be ordered or accepted is possible via two routes: manual entry or entry via an external device:

- Manual input: input via the keyboard of the device on which the PWA is used (computer keyboard or the keyboard of a smart phone rather than that of a tablet).
- Insertion via an external device: this is done via the Bluetooth connection of a barcode reader to the device being used. Any code read with the reader will automatically be inserted into the text area.

Once all the codes have been entered and the order is ready to be sent or to be accepted, press the appropriate button, 'Send' or 'Accept'.

When the button is pressed, a file of type '.dat' will be created with the codes in the text area inside.

The code for creating the file:

```
1 let token= local Storage . get Item ( ' token ' ). replace ( '/' g , ' ' );  
2 const blob= new Blob ( [ this . text Content ] , { type : ' text / plain  
3 ' } ); const file= new File ( [ blob ] , ' test . dat ' , { type : ' text /  
4 plain ' } ); const form Data= new Form Data ( );  
5 form Data . append ( ' file ' , file );
```


The creation of this file was possible using the method:

```
new Blob([this.textContent],    type:  'text/plain' ).
```

This method creates a 'blob' with the data contained in the text area. Blob, literally meaning 'Binary Large Object', is thus an object similar to a raw, immutable data file.[\[3\]](#)

Inside this.textContent is the text written in the text area. type:'text/plain', specifies the type of MIME (Multipurpose Internet Mail Extensions), which is nothing more than a label that allows the browser or server to ascertain the type of file it is reading, so as to interpret it correctly.

After creating a blob of data, the .dat file is created using the method:

```
new File([blob], 'test.dat',    type:  'text/plain' )
```

blob is passed as an array because File() as the first input parameter requires an array of data parts (you can also pass strings or ArrayBuffer), the second input parameter is the name you want to assign to the created file (in this case, 'test.dat') and finally the additional options are passed, in which case the MIME type is repeated (type: 'text/plain').

Finally, formData() creates an object used to send data by post, which in this case is the method used for the API call, and adds the file by renaming it 'file'.

At this point, the file has been successfully created and you can call the API who will take care of sending or accepting the order.

For the API to work, it requires the customer's ID, the user's token and of course, the newly created formData containing the order data.

Positive reply In case of a positive answer, the order is sent or accepted, the text area is cleared.

Negative response In case of a negative answer, the order is neither sent nor accepted, the text area is not cleared.

In both cases, a modal appears; its operation will be explained in the next section.

The screenshot shows a mobile application interface for creating a new order. At the top, there is a back arrow icon. The title 'Nuovo ordine' is centered in a bold, dark blue font. Below the title, it says 'Crea un nuovo ordine per: Cliente S.r.l.s'. A list of four instructions is provided: connecting a barcode reader via Bluetooth, positioning the reader in the text box, reading the barcode, and clicking 'Invia' to submit. Below the instructions is a large, empty text input box with the placeholder text 'Scrivi qui...'. At the bottom, there are two buttons: a dark blue 'Invia' button and a grey 'Cancella tutto' button.

Figure 3.6: The page for placing a new order

Concluding the explanation of this part of the PWA, it is necessary to remind that the inserted image is only one because the acceptance page of an order presents a user interface identical to the one shown for the creation of a new order, this allows continuity and consistency to be maintained throughout the application.

3.8 Modal

The modal that is displayed at the end of either operation has been implemented via *Bootstrap*.

For its creation, a component, *Modal.vue*, was created. Inside it is the *JavaScript* code and the *HTML* code:

```
1 <div class =" modal fade " id =" modal Id " data - bs - backdrop ="
  static " data - bs - keyboard =" false " tabindex =" -1" aria -
  hidden =" true " ref =" modal Element ">
2 <div class =" modal - dialogue ">
3   <div class =" modal - content ">
4     <div class =" modal - header ">
5       <h5 class =" modal - title " >{{ title }} </ h5>
6       <button type =" button " class =" btn - close " data - bs -
        dismiss
7         =" modal " air - label =" Close " ></ button>
8     </ div>
9     <div class =" modal - body ">
10      <slot> </ slot > // allows the parent component to insert
        content
11    </ div>
12    <div class =" modal - footer ">
13      <button type =" button " class =" btn btn - secondary " data -
        bs - dismiss =" modal "> Close </ button>
14    </ div>
15  </ div>
16 </ div>
```

Modal.vue then creates a modal using *Bootstrap 5*'s modal system. The purpose is to display customised content within a window, with a title and a close button.

By creating a separate component, it is possible to allow other application components to use the modal without having to rewrite the structure each time. Two special attributes have been added to the modal:

`data-bs-backdrop="static"`

prevents the modal from closing by clicking outside the window that is created, whereas

`data-bs-keyboard="false"`

prevents closing using the 'Esc' key on computer keyboards.

Returning briefly to the discussion in the previous section, should the response to the sending or acceptance of an order be positive, the modal containing a message of success in the operation would be opened.

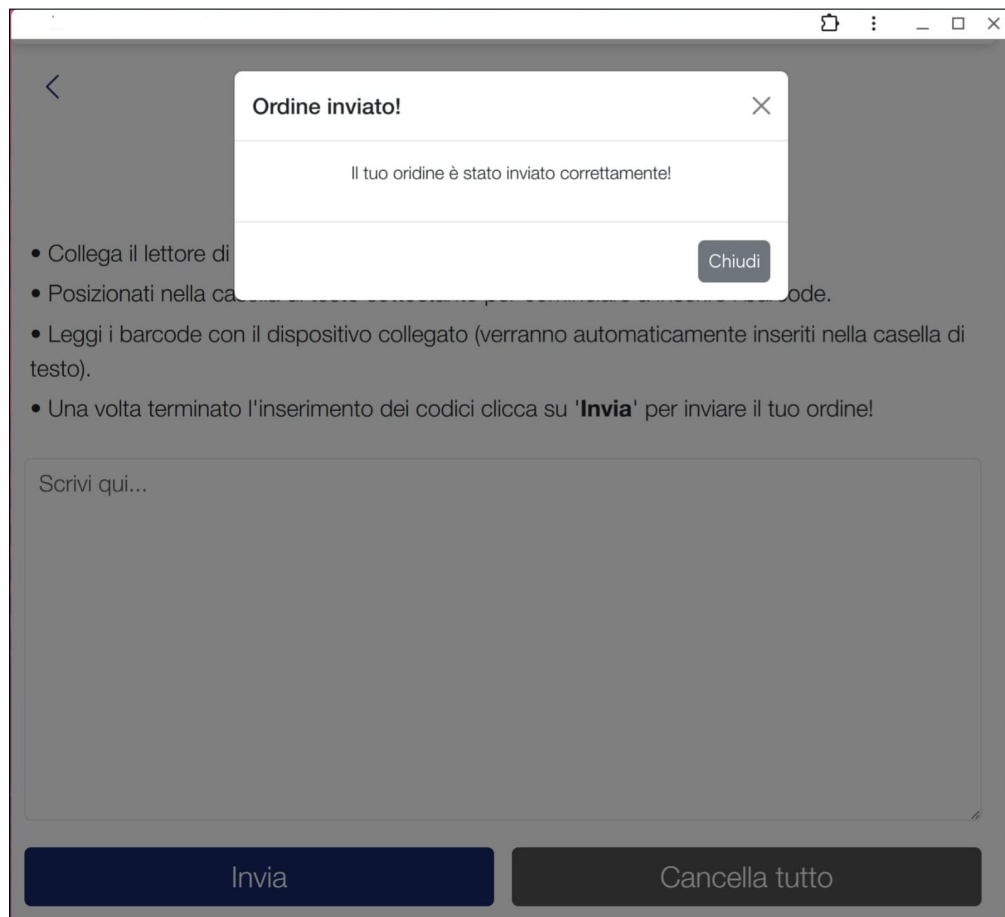


Figure 3.7: Success mode

Otherwise, i.e. in the event that the operation was unfortunately not successful, but rather failed, the modal will open and display an error message.

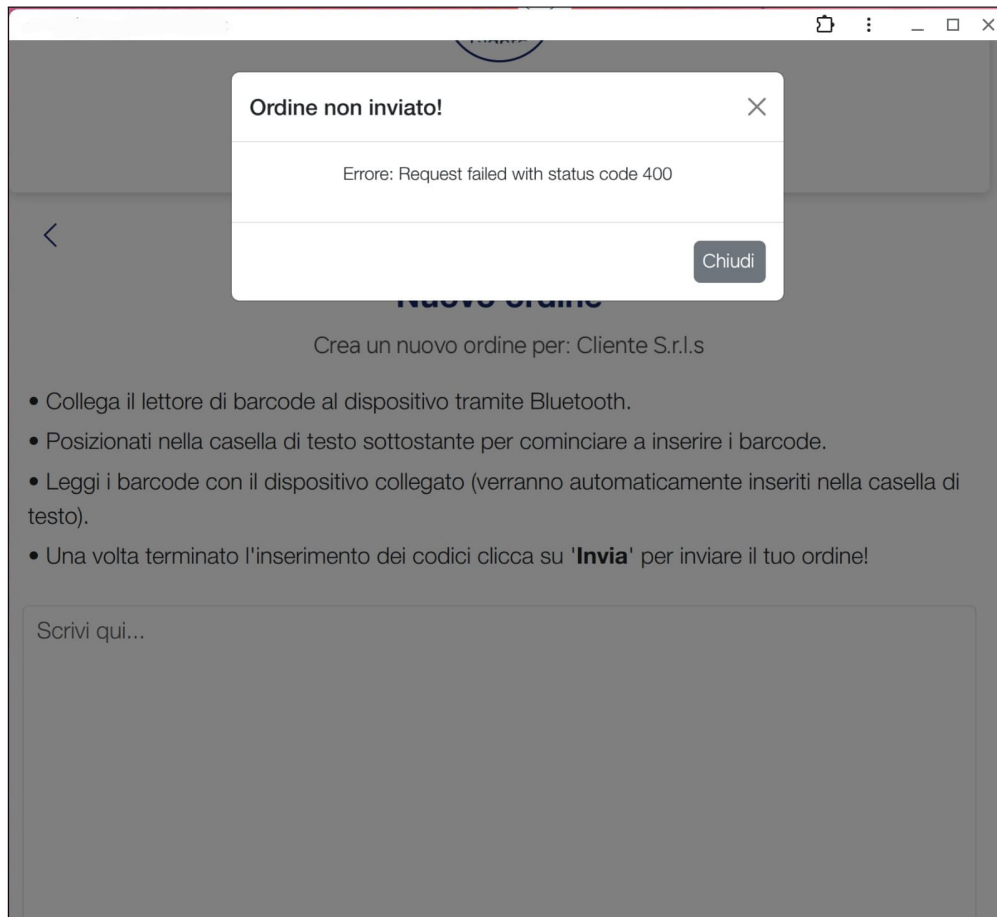


Figure 3.8: Error mode

Results and conclusion

At the end of the project, it is essential to conclude by analysing the results obtained. Once completed, the *Progressive Web App* presents itself as an intuitive, fast, responsive, working, ready-to-use application and, once installed, completely similar to a native application.

In the future, the PWA can be easily adapted to different languages, without the need to restructure the entire system, as the system allows the use of translation systems.

It will be sufficient to modify the texts visible to users, making expansion to new languages quick and easy.

The code written in English and the quick editing of the texts in Italian will pave the way for a global application, easily accessible by users of different nationalities.

Sitography

- [1] Google Chrome. *Chrome DevTools*. 16 February 2025. URL: <https://developer.chrome.com/docs/devtools/overview?hl=en>.
- [2] Red Hat. *What are Application Programming Interfaces (APIs)*. Feb 16, 2025. URL: <https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces#:~:text=The%20term%20API%20C%20acronym%20of,the%20integration%20of%20applications%20software..>
- [3] MDN. *Blob*. 18 February 2025. URL: <https://developer.mozilla.org/en-US/docs/Web/API/Blob>.
- [4] MDN. *CSS: Cascading Style Sheets*. 02 February 2025. URL: <https://developer.mozilla.org/en-US/docs/Web/CSS>.
- [5] MDN. *HTML: HyperText Markup Language*. 02 February 2025. URL: <https://developer.mozilla.org/en-US/docs/Web/HTML>.
- [6] MDN. *JavaScript*. 16 February 2025. url: <https://developer.mozilla.org/en-US/docs/Web/JavaScript>.
- [7] MDN. *Progressive Web Apps*. 26 January 2025. URL: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps.
- [8] MDN. *ServiceWorker*. 26 January 2025. URL: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API.

List of figures

2.1	Classroom utilisation result	13
3.1	Login Page.....	24
3.2	The customer's choice	26
3.3	The expanded menu and profile card	29
3.4	Home Page 1	31
3.5	Home Page 2	31
3.6	The page for placing a new order	34
3.7	Successful mode	36
3.8	Error mode.....	37