MOBBEEL PROJECT DOCUMENTATION

1. Introduction

This PDF has been designed to explain in more detail all the development carried out for the technical test conducted for Mobbeel. In this document, we will explain how each requested requirement was implemented, and a 10-minute video demonstrating the web application to showcase its functionality will also be attached.

2. Creation of the project in Angular

We chose the Angular framework because it provides structure, scalability, and efficiency to the project, facilitating the integration of complex functionalities such as image capture, document detection, interaction with external APIs, and other features that will be discussed later. Additionally, its component-based system and development tools make the code easier to maintain and improve in the long term.

The first step is to create our Angular project. However, before doing so, we need to ensure that the Angular CLI is installed, which can be done with the command *npm i -g @angular/CLI*. If it's already installed, we proceed to create the project using the command ng new project-name, where we can choose the desired project name. During this process, we will be asked to configure several project options, and once completed, it will be ready to start working.

It is important to be mindful of the Angular version we install, as compatibility issues may arise. I encountered this issue when installing the most recent version (18), which omitted several key Angular configuration files, such as *app.module or environments*. To avoid this, it is recommended to disable the standalone option when creating the project by using the command: *ng new project-name --standalone=false*

3. Implementation of the requested requirements

Next, we will provide a general explanation of how we implemented the requested requirements. The results can be seen in the attached video. Before proceeding, let's review the structure of our project in Illustration 1:

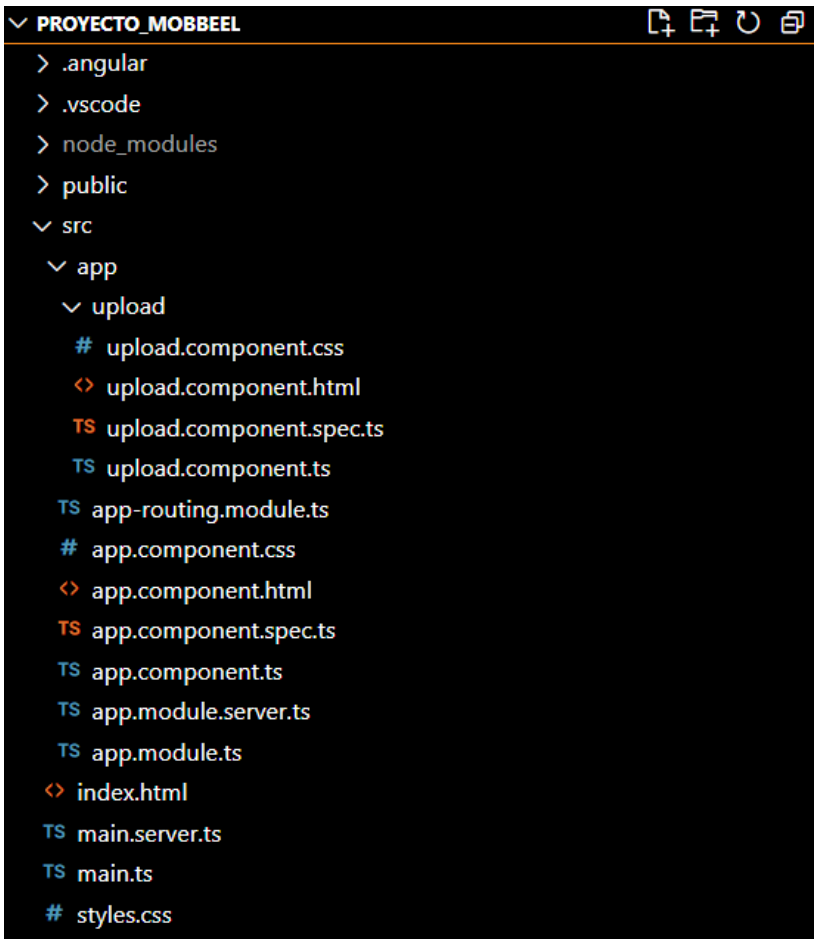

Illustration 1: Project structure

As we can see, the project contains a single component called 'upload,' where the files created by Angular are in the 'upload' folder. This folder includes a *.ts* file with TypeScript code, an *.html* file for adding elements to our page, and a *.css* file to give it a good visual appearance.

Lastly, there are additional files, as seen in the image, such as *index.html*, *main.ts*, or *styles.css*, which are added when the project is created. These files were not modified for this test.

3.1 Upload a document from the device

The first requested feature was that the user could upload an official document to a webpage, such as an ID card, driver's license, health card, etc. Once the document is uploaded, it should be displayed somewhere on the webpage.

For this, we were provided with an API capable of detecting whether an image is a document or not. For example, as shown in Illustration 2, if we upload an image that is a document, a correct response is received with various fields representing parts of the document itself, such as its position, description, image, etc.

ID document detected:
HTTP code: 200
JSON:


```
{
   "code": "OK",
   "description": "DETECT_DOCUMENT_SUCCESS",
   "scanId": "{SCAN_ID}",
   "imageDocumentDetected":     {BASE_64_ENCODED_JPG_IMAGE},
   "imageMRZCoordinates": {},
   "imageFaceCoordinates": {}
   "imageDocumentCoordinates": {
       "p1": {
           "x": 79,
           "y": 13
       },
       "p2": {
           "x": 1093,
           "y": 16
       },
       "p3": {
           "x": 1078,
           "y": 636
       },
       "p4": {
           "x": 93,
           "y": 643
       }
   },
   "imageQuality": "VALID",
   "imageQualityScore": 0.571
}
```


Illustration 2. Correct API response

On the other hand, if the uploaded image is not a document, we will receive an error response from the API, as shown in Figure 3:

ID document NOT detected:
HTTP code: 200
JSON:


```
{
 "code": "ERROR",
 "description": "ERROR_DOCUMENT_NOT_DETECTED"
}
```


Illustration 3. Incorrect API response

Therefore, instead of displaying the API response on the webpage, the document will be shown if it is an official document, and if not, an alert will be displayed indicating that no document was detected. This could be due to the image not being a document or because the image quality is poor.

Once we understand how the API works, the implementation of the first requirement is explained below. The 'Upload file' button allows the user to upload an image file from their device to be processed afterward. The following outlines the process and the implementation of the button:

```
<input
  type="file"
  (change)="onFileSelected($event)"
  style="display: none;"
  #fileInput
/>
<button class="upload-button" (click)="fileInput.click()">Upload
file</button>
```

The <*input*> element of type file is responsible for opening the file selector when the user clicks the button. However, this input is hidden *(style="display: none;")* since a custom button is used to trigger its function. On the other hand, the button has the event *(click)="fileInput.click()"*, which simulates a click on the hidden <*input*> when the button is pressed, opening the dialog to select a file.

Meanwhile, in the TypeScript file, we have the method onFileSelected:

```
onFileSelected(event: Event) {
  const target = event.target as HTMLInputElement;
  if (target.files) {
      this.selectedFile = target.files[0];
      this.selectedFileName = this.selectedFile.name;
      this.captureTime = null;
      this.croppedImage = null;
      this.result = null;
      this.showCaptureButton = false;
      this.isAutoDetected = false;
      this.cd.detectChanges();
  }
}
```

The onFileSelected function is executed every time the user selects a file. The selected file is obtained through the event and stored in the *selectedFile* variable. Additionally, the name of the file is saved in *selectedFileName* so that the name of the uploaded file can be displayed.

Moreover, other variables such as *croppedImage* and result, which are related to previous captures or past detection results, are reset. This ensures that every time a new file is selected, the previous document and its name are cleared up and updated with the new name and document. There are also other variables used for additional requirements that will be explained later.

Once the functionality of the first button has been explained, we need to display the document. For this purpose, a new button called *'Check your document'* has been created:

```
<button class="upload-button_act" (click)="onUpload()"
[disabled]="!selectedFile">Check your document</button>
```

The event *(click)="onUpload()"* calls the *onUpload* function when the button is pressed. Additionally, the attribute *[disabled]="!selectedFile"* ensures that the button is only enabled if a file is selected. This prevents the user from submitting the request if they have not previously selected or captured a file. Now, we can observe the functionality in the TypeScript file for the *onUpload* method:

```
onUpload() {
  if (this.croppedImage) {
    this.cd.detectChanges();
  } else if (this.selectedFile) {
    const formData = new FormData();
    formData.append('documentSide', 'front');
    formData.append('documentType', 'TD1');
    formData.append('image', this.selectedFile);
    formData.append('licenseId', 'mobbscan-challenge');
    formData.append('returnCroppedImage', 'true');

    this.http.post(this.apiUrl, formData).subscribe(response => {
      this.result = response;

      if (this.result && this.result.imageDocumentDetected) {
        this.croppedImage = this.result.imageDocumentDetected;
        this.cd.detectChanges();

        if (this.isAutoDetected) {
          this.updateFileInfo();
        }
      } else {
        console.log('No document was detected in the new request');
        this.croppedImage = null;
        alert('No document was detected.');
      }

      this.stopCamera();
      this.resetFileInput();
      this.cd.detectChanges();
    }, error => {
      console.error('Error:', error);
      this.croppedImage = null;
      alert('An error occurred while processing the image.');
      this.stopCamera();
      this.cd.detectChanges();
    });
  } else {
    console.log('No file selected or document detected');
    alert('No document has been uploaded or detected.');
  }
}
```

The *onUpload* function is executed when the user presses the *'Check your document'* button. It begins by checking if a cropped image already exists (from a previous detection), in which case, the view is simply updated. If no cropped image exists but a file has been selected, a *FormData* object is created containing the selected file and the necessary parameters for the request.

Subsequently, an HTTP request is sent to the API using HttpClient. If the API returns a detected document image, it is stored in croppedImage to be displayed in the interface. If no document is detected, the user is notified through an alert indicating that no document was found.

Finally, there are additional calls that will be explained later, which pertain to new requirements.

In summary, at this point, we have completed the explanation of the first requirement, which allows the user to upload any image file from their computer. The name of the uploaded file will then be displayed, and after clicking the button to show the document, the document will be displayed if it is a valid one; otherwise, an alert will be shown.

3.2 Upload the document using a manual photo

This new requirement aims to perform the same function as the first one, except that instead of uploading a file from the computer, a photo will be taken using the device's camera. Therefore, the methods involved in this requirement will be explained, some of which will be the same, such as *onUpload*, since this method is always called to display the document, whether it is from a file uploaded from the device or from a captured image.

To implement this requirement, there is a button called *'Open camera.'* This button is key to activating the device's camera and allowing manual image capture. When pressed, it calls the *openCamera()* method, which is responsible for accessing the device's camera and displaying the live video feed:

```
<div *ngIf="isCameraActive">
      <video #video autoplay></video>
      <button class="capture-button" *ngIf="showCaptureButton"
(click)="captureImage()">Capture image</button>
    </div>
```

This button only appears if the camera is not active *(!isCameraActive)*. This ensures that the *'Open camera'* button is not visible when the camera is already in use. When this button is clicked, it invokes the method that uses the *navigator.mediaDevices.getUserMedia* API to access the camera. If permission is granted, the video is displayed in a *<video>* element linked to the component.

Therefore, the relevant code for the *openCamera* method, located in the TypeScript file, is shown below:

```
openCamera() {
  if (this.isCameraActive) {
    console.warn('The camera is already open.');
    return;
  }

  this.isCameraActive = true;
  this.showCaptureButton = true;
  this.cd.detectChanges();

  navigator.mediaDevices.getUserMedia({ video: true })
    .then(stream => {
      this.videoElement.nativeElement.srcObject = stream;
      this.videoElement.nativeElement.play();
      this.cd.detectChanges();
    })
    .catch(error => {
      console.error('Error accessing the camera:', error);
      alert('Could not access the camera.');
      this.isCameraActive = false;
      this.cd.detectChanges();
    });
}
```

This method ensures that the camera is not already active by checking *isCameraActive*. If it is not active, it attempts to open it and displays the capture button by setting *showCaptureButton = true*. If access to the camera cannot be granted, an error message is shown, and the camera state is disabled.

Once the camera is active, the *'Capture image'* button appears. This button allows capturing an image from the video feed and converting it into a file that can be processed. The capture button only appears when the video is active and disappears once the image is captured:

```
<button class="capture-button" *ngIf="showCaptureButton"
(click)="captureImage()">Capture image</button>


<canvas #canvas style="display: none;"></canvas>
```

As mentioned earlier, this button only appears when the *showCaptureButton* variable is activated, which happens when the camera is opened in the *openCamera()* method. When this button is pressed, the video is captured in a *<canvas>*, converted into a blob, and then into an image file, which is stored in *selectedFile*. Therefore, the code for the *captureImage* method is shown below:

```
captureImage() {
    const canvas = this.canvasElement.nativeElement;
    const context = canvas.getContext('2d');

    if (context) {
        const video = this.videoElement.nativeElement;

        canvas.width = video.videoWidth;
        canvas.height = video.videoHeight;

        context.drawImage(video, 0, 0, canvas.width, canvas.height);

        canvas.toBlob(blob => {
            if (blob) {
                this.selectedFile = new File([blob], 'captured-
image.jpg', { type: 'image/jpeg' });
                this.selectedFileName = this.selectedFile.name;

                const currentDateTime = new Date();
                this.captureTime =
`${currentDateTime.getHours()}:${currentDateTime.getMinutes()}:${cur
rentDateTime.getSeconds()}`;

                this.croppedImage = null;
                this.isAutoDetected = false;
                this.cd.detectChanges();

                // Stop the camera and hide the capture button
immediately
                this.stopCamera();
                this.resetFileInput();
                this.showCaptureButton = false;
                this.cd.detectChanges();
            }
        }, 'image/jpeg');
    } else {
        console.error('Could not obtain the canvas context.');
    }
}
```

This method uses the drawImage function to capture the content of the video and draw it onto the canvas. After drawing the image on the canvas, it is converted into a JPEG image blob and finally into a File type. The capture time is stored in captureTime, and the capture button and camera are disabled after the image is captured.

After capturing the image with *captureImage()*, the resulting file is stored in *selectedFile*, with a static name along with the capture time to differentiate it from other captures. This file can then be uploaded and processed by pressing the *'Check your document'* button, invoking the *onUpload()* method. The change in this flow is that the file is now created manually after the image capture, which is essential for its subsequent validation. This set of interactions allows taking a photo, creating an image file, and processing it as if it were a document uploaded from the user's hard drive.

3.3 Upload the document using real-time detection

This new requirement is an additional request, where the difference from the previous one is that the document can be uploaded without having to press a capture button. In other words, once the document is detected, it will be displayed automatically on the webpage.

Therefore, a new button is needed, called *'Open video*,' which initiates the automatic detection of documents in the live video feed. Unlike the *'Open camera*' button, which allows manual image capture, this button activates a continuous process that attempts to detect a document every second, without user intervention.

```
<button class="video-button" type="button"
(click)="startVideoDetection()">Open video</button>
```

When this button is pressed, the automatic document detection process in the video is initiated. This calls the *startVideoDetection()* method, which activates the camera and repeatedly executes the document detection process. The method is shown below:

```
startVideoDetection() {
    this.showCaptureButton = false;
    this.openCamera();
    this.videoInterval = setInterval(() => {
      this.detectDocument();
    }, 1000);
    this.cd.detectChanges();
  }
```

Since the detection is automatic, there is no need to press the 'Capture image' button. Additionally, the *openCamera()* method is reused to open the camera and obtain the video feed. Finally, *setInterval()* is used to call the *detectDocument()* method every second, attempting to detect a document in each frame of the video. Therefore, the code for *detectDocument* is shown below:

```
detectDocument() {
    const video = this.videoElement.nativeElement;
    const canvas = this.canvasElement.nativeElement;
    const context = canvas.getContext('2d');

    if (!video || !context) {
      console.error('Video context or element unavailable');
      return;
    }

    canvas.width = video.videoWidth;
    canvas.height = video.videoHeight;
    context.drawImage(video, 0, 0, canvas.width, canvas.height);

    canvas.toBlob(blob => {
      if (blob) {
        const file = new File([blob], 'frame.jpg', { type:
'image/jpeg' });
```

```
        const formData = new FormData();
        formData.append('documentSide', 'front');
        formData.append('documentType', 'TD1');
        formData.append('image', file);
        formData.append('licenseId', 'mobbscan-challenge');
        formData.append('returnCroppedImage', 'true');

        this.http.post(this.apiUrl, formData).subscribe(response =>
{
          this.result = response;
          if (this.result && this.result.imageDocumentDetected) {
            this.isAutoDetected = true;
            this.updateFileInfo();
            this.stopCamera();
            this.showCaptureButton = false;
            this.resetFileInput();
            this.cd.detectChanges();
          }
        }, error => {
          console.error('Error processing the frame:', error);
          this.cd.detectChanges();
        });
      }
    }, 'image/jpeg');
  }
```

This method is similar to the manual capture process, where a frame from the video is captured and drawn onto the canvas. It is then converted into a blob file, and subsequently, the frame is sent to the API for document detection. If a document is detected, the *updateFileInfo()* method is invoked to update the information of the captured file. Once a document is detected, the camera is automatically stopped by calling *stopCamera()*. Therefore, the functionality of the *updateFileInfo()* method is shown below:

```
updateFileInfo() {
    if (this.isAutoDetected) {
      const currentDateTime = new Date();
      this.captureTime =
`${currentDateTime.getHours()}:${currentDateTime.getMinutes()}:${cur
rentDateTime.getSeconds()}`;

      this.selectedFile = new
File([this.result.imageDocumentDetected], 'detected-document.jpg', {
type: 'image/jpeg' });
      this.selectedFileName = this.selectedFile.name;

      this.croppedImage = this.result.imageDocumentDetected;
      this.cd.detectChanges();
    }
  }
```

This method only updates the information if *isAutoDetected* is true, which occurs when a document is detected automatically. The file receives a name based on the capture time and the file name is updated to *'detected-document.jpg.'* This is to differentiate between a document taken with a manual photo (capture-image) or a document that has been detected automatically.

In the automatic detection flow, the *'Check your document'* button remains relevant for viewing the detected document. The difference here is that the file is created automatically during video detection and can be accessed without the need to manually upload a file. With this functionality, an efficient process for automatic document capture and detection in real time through the device's camera has been implemented.

3.4 Responsive design

Finally, the last requirement that was requested, which is also an extra, is that it should be responsive design. This term refers to applying techniques (relative units, flexbox, media queries, and scalable elements) to the webpage so that on devices with smaller screens, such as mobile phones, all content can be viewed just as well as on larger devices like computers.

In the code below, some techniques applied in the .css file for responsive design can be observed:

```
.container {
    justify-content: center;
    align-items: center;
    display: flex;
    flex-direction: column;
    min-height: 90vh;
}

.container_shadow {
    box-shadow: 0 4px 20px rgba(0, 0, 0, 0.2);
    justify-content: center;
    align-items: center;
    display: flex;
    flex-direction: column;
}

img {
    display: block;
    margin: 10px auto;
    max-width: 100%;
    height: auto;
    border: 1px solid #ddd;
    border-radius: 4px;
}

video {
    width: 100%;
    height: auto;
```

```
    border: 1px solid #ddd;
    border-radius: 4px;
    margin: 10px 0;
}

@media (max-width: 600px) {
    button {
        font-size: 16px;
    }
    h2 {
        font-size: 2em;
    }
}
```

The images and videos have a *max-width: 100% and width: 100%,* which means they will adjust to the size of the parent container, essential for responsive design. Additionally, the buttons' width is set to *width: 100%,* allowing them to occupy the full width of the container, adapting to the screen size.

The classes .container and .container_shadow use *display: flex* with alignment properties *(justify-content: center and align-items: center*), allowing the internal elements to automatically adjust to different screen sizes and be centered appropriately.

On the other hand, there is a media query specifically for screens with a maximum width of 600px. This query adjusts the font size of the buttons and the title (h2) on smaller screens, improving readability and user experience on mobile devices.

Finally, other elements like div and other containers have margins and padding defined with *max-width: 600px and margin: 0 auto*, meaning they will remain centered and appropriately sized on both small and large screens.

4. Issues Encountered

At this point, the main issues encountered during the development of the technical test will be explained, along with how they were resolved.

4.1 Camera Stopping

During the development of the technical test, a workflow was intended where every time a photo was taken manually or a document was detected automatically, the camera would close. However, we encountered several problems and bugs while trying to implement this flow, as the camera remained on and did not turn off after taking a photo or capturing an image.

Therefore, a method was created outside of the main methods called *'stopCamera*,' which is shown below:

```
stopCamera() {
    const videoStream = this.videoElement.nativeElement.srcObject as
MediaStream;
    if (videoStream) {
      const tracks = videoStream.getTracks();
      tracks.forEach(track => track.stop());
    }
    this.videoElement.nativeElement.srcObject = null;
    this.isCameraActive = false;
    clearInterval(this.videoInterval);
    this.cd.detectChanges();
    console.log('Camera stopped.');
  }
```

The method *stopCamera()* is responsible for stopping the camera and releasing the associated resources once they are no longer needed. First, it retrieves the video stream from the video element, and if it exists, it stops all audio and video tracks to free up the hardware. Then, it unbinds the video stream from the element, updates the state of the variable indicating whether the camera is active, and stops any ongoing automatic detection processes. Finally, it ensures that the view is updated to reflect these changes and logs a message to the console for debugging purposes. This method is crucial for ensuring efficient use of system resources and maintaining user privacy.

4.2 Cleaning the File Input Field

There was an issue when uploading files from the device. It is obvious that if you upload a file and then try to upload the same once again, you cannot do so because it is already uploaded. The problem was that after uploading a file from the device and then taking a manual or automatic photo, if you decided to upload the first file again, it would not load. This was incorrect because the file had already changed due to the manual or automatic photo. That's why the method *resetFileInput()* was created, as shown below:

```
resetFileInput() {
    const fileInput = document.querySelector('input[type="file"]')
as HTMLInputElement;
    if (fileInput) {
        fileInput.value = '';
    }
}
```

The *resetFileInput()* method is used to clear the file input field after a file upload or image capture has been processed. It searches for the file input element in the DOM, and if found, sets its value to an empty string. This method is important to ensure that the user can re-select the same file or a new image without issues, as some browsers may not allow a user to reselect the same file unless the input field is reset. Therefore, it helps improve the user experience by making the interface more intuitive and flexible.

4.3 Updating the View

During the development of the project, there were visualization issues, as Angular did not reflect these changes in the main view. For example, when taking an automatic or manual photo, the camera did not disappear as it was supposed to. This is why *ChangeDetectorRef* was imported *from @angular/core*.

By using the statement *this.cd.detectChanges()*, the change detection system is notified to check the state of the components and update them based on changes in the data. In this project, it is mainly employed after making modifications to the component variables, such as when selecting a file, capturing an image, or updating the information of the detected document

4.4 Separating the Logic for Manual and Automatic Document Capture

There were several issues encountered in getting the document capture functionality to work independently for both manual and automatic methods. This was due to the intention to reuse some methods utilized for detecting documents from manual photos in the implementation of automatic detection. Specifically, the method *openCamera* was reused, but when attempting to do so, using the openCamera function also triggered automatic document detection, which was not the desired behavior, as selecting that option was meant for manual document capture.

To avoid this problem, a new method, previously explained, called *startVideoDetection* was created. This method is responsible for opening the camera (by calling *openCamera()*) and, most importantly, initiating an interval that calls the detectDocument() function every second. This is what enables the automatic detection of documents while the video feed is being displayed in real time.

5. Possible Improvements for the API

When considering a possible improvement for the API, it could be the implementation of an endpoint that allows users to consult the history of processed documents, including details about detection results and processing dates.

In terms of how it should be implemented, a table should first be created in the database to store information about each document, including the file name, detection results, and processing date. Then, an endpoint would be implemented on the server to retrieve and return this history in JSON format. Additionally, each detection process should be automatically logged in this table when a document is processed, allowing users to view their upload history along with the corresponding dates.

6. Conclusions

The development of this technical test has demonstrated an effective integration of technologies by implementing a document detection system using Angular and an API, allowing for a smooth and responsive user experience. In summary, there are three options that the user can utilize, providing constant updates and visibility of a document, whether uploaded from their device, taken manually as a photo, or captured automatically, always clearing the previous one and displaying the most recently uploaded or captured document.

I hope this documentation, along with the video and code, will be uploaded to GitHub, and I hope you appreciate and enjoy it as much as I have enjoyed creating it this week.