TÉCNICO LISBOA

Local Web Application Development

Framework Proposal and Demonstration on Two Case Studies

Diogo Miguel de Mello Caldeira Reis Almiro

Thesis to obtain the Bologna Master Degree in

Information Systems and Computer Engineering

Supervisor(s): Prof. Helena Sofia Andrade Nunes Pereira Pinto
Prof. José Luís Brinquete Borbinha

Examination Committee

Chairperson: Prof. David Manuel Martins de Matos
Supervisor: Prof. José Luís Brinquete Borbinha
Member of the Committee: Prof. João Fernando Peixoto Ferreira

November 2021

Aos meus avós.

Acknowledgments

In the first place, I would like to thank my supervisors, Sofia Pinto and José Borbinha, for the opportunity of working on this project in INESC-ID and for their invaluable guidance and availability throughout the project. Thank you for the weekly meetings that encourage me to keep working and for the always insightful feedback.

I want to thank my university colleagues and now friends for life, Diogo Tito, Pedro Granja and Susana Gamito, for their company, care and encouragement in the last years. It made these years have a smoother and more enjoyable path.

Lastly, I want to thank my family for their support and care since ever. In particular, to my mother and Mariana, who took time to understand, help and review this document.

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020 (INESC-ID).

Resumo

A tecnologia *web* tem atingido uma grandeza que a torna relevante não só para aplicações *web* mas também para o que costumava ser o domínio das aplicações *desktop*. O desafio deste projeto é propor uma arquitetura de referência que combine características das aplicações *desktop* e das tecnologias *web*, recomendando uma solução que use tecnologia madura. Para demonstrar e validar a solução proposta usam-se dois casos de estudo aplicados em contexto real. Ambos necessitam de gerir documentos e aplicar reconhecimento óptico de caracteres. O primeiro caso de estudo está relacionado com o trabalho de arquivar documentos históricos por parte da Secção de Gestão de Arquivos e Registos das Nações Unidas. O segundo caso de estudo está relacionado com o processo de decisão dos juízes do Supremo Tribunal de Justiça. Este projeto começa com a introdução de conceitos chave e dos requisitos de cada caso de estudo. Depois, analisa o estado da arte em relação às tecnologias do lado do servidor de uma aplicação. Em vez de se criar uma tecnologia nova, este projeto introduz uma extensão a uma tecnologia do estado da arte. A tecnologia escolhida foi o *Express*, e este projeto descreve a sua expansão. O projeto acaba com a demonstração da solução proposta através de pequenas aplicações criadas para os casos de estudo.

Palavras-chave: Tecnologia Web, Aplicações Hibridas, Framework, Express, NodeJS, OCR

Abstract

Web technology has reached a penetration that makes it relevant not only for web applications but also for what used to be the classic domain of desktop applications. The challenge of this project is to propose a reference architecture that combines attributes of desktop applications and attributes of web technology, recommending a solution using mature technology. There are two real-life case studies to demonstrate and validate this technology. Both of them require the management of documents and the execution of optical character recognition. The first case study relates to the workflow when archiving historical documents on the Archives and Records Management Section of the United Nations. The second case study relates to the workflow of making decisions by judges of the ”Supremo Tribunal de Justiça”. This project starts by introducing some core concepts and requirements for the case studies. It then analyses the state of the art of technologies related to the server-side of an application. Instead of introducing completely new technology, this project introduces an expansion from a particular technology from the analysis. The chosen technology is Express, and the project describes this expansion. It also demonstrates the usage of the proposed solution by implementing small applications for the case studies.

Keywords: Web Technology, Hybrid Application, Framework, Express, NodeJS, OCR

Contents

List of Tables

List of Figures

List of Listings

Glossary

ARMS	Archives and Records Management Section.
E2E	End-to-End.
IRIS	Information, Rationalization, Integration and Summarization.
JSON	JavaScript Object Notation.
NPM	Node Package Manager.
OCR	Optical Character Recognition.
SSR	Server-Side Rendering.
STJ	*Supremo Tribunal de Justiça*.
TDD	Test-Driven Development.

Chapter 1

Introduction

The so-called “web technology” has reached a penetration that makes it relevant not only for real web applications but also for what used to be the classic domain of “desktop applications”, meaning applications intended to execute in a single personal computer by an individual user. Therefore, the challenge of this project is to propose a reference architecture for this kind of scenario, recommending mature technology and demonstrating it. The motivation comes from two other projects with mutual requirements concerning these issues, making this problem relevant. These projects are related to the workflow for digitising historical documents by an archivist at the United Nations Archives and the workflow for analysing legal documents by a judge in a top national court.

This section introduces the motivation for this work, as the two case studies mentioned, which are referred to as ARMS and IRIS. Then, it enumerates in more detail the project objectives. It describes the methodology and time spent for each step of the project. It indicates software contributions throughout the project and where to find them. Finally, it describes the structure of the document.

1.1 Motivation

A significant motivation for this work relates to the Archives and Records Management Section (ARMS) of the United Nations. They have the job to preserve, scan and publish several documents. During this process, they also recognise text on the document with optical character recognition (OCR). Currently, ARMS archivists use Adobe’s paid tool to manually convert each original scan, a TIFF file, to a PDF and make the OCR of the PDF in a two-step process. A previous project created an open-source command-line based solution that can process multiple TIFF files directly or within a folder. The solution will allow the ARMS archivists, who do not use nor know how to use the command-line interface, to use this new technology with ease.

A second motivation concerns a need for the “Supremo Tribunal de Justiça” (STJ), the top court of Portugal’s hierarchy of justice courts. Their judges make decisions over documents with thousands of pages. This case study is still in very early development. The current idea is to help judges in decision-making, allowing them to annotate and search on the documents. Some pages might already have

digitalised text others might require OCR as they can also be scanned pages. In other words, give an application to a judge that can OCR the document, give intelligent search results, and allow the judge to improve these results with their annotations.

1.2 Objectives

As said before, it is relevant to explore the "desktop applications" domain from a "web technology" perspective. The project's main objective is to propose a generalised reference architecture with mature technology, validating the proposed solution using the case studies mentioned above. For each case study, there is the secondary objective of improving the workflow of the end-user.

The solution ought to allow the development of new technologies to be more simplistic, for instance, without worrying about the user interface. Both case studies mentioned have in common: workflows inside an administrative environment; and the need to make OCR on files.

Although the primary objective is to know the value of the technology proposed, it should also implement the case studies applications with the solution to be the most valuable to the end-user.

1.3 Methodology

This project started with defining a mental image for the case studies, researching and hypothesising the best solutions. Afterwards, the project starts implementing a solution. There are three central moments during the implementation process:

1. The first moment is to implement the solution's core concepts. It relied on an understanding of the technologies used "under the hood".
2. The second moment is to apply the solution to small use cases and the case studies, yielding insights on improvements that could be made for the solution.
3. The last moment is to implement the final applications for the end-users.

Regarding the releases to the end-user, the first proof of concept for ARMS was very insightful to understand the end user's necessities. It was followed by a second application that had smaller improvements according to the end user's feedback.

1.4 Contributions

The main products of this project are maintained on different projects of Github.

The principal solution, which is called Webfocus, is available under the Github Organisation `webfocus-js`[1]. The source code of its main structures, `WebfocusApp`, `WebfocusComponent` and specific component implementations are within it.

[1] See www.github.com/webfocus-js/

The ARMS releases are maintained into two different Github Repositories. The first proof of concept is available at `diogoalmiro/arms-node-webapp`[2] The second implementation with some application releases is available at `diogoalmiro/arms-manager`[3].

A “nice to have” feature that controls the notification area/system tray outside the solution’s scope is available at `diogoalmiro/ctray`[4].

The previous project solution, `ARMS-Workflow` on the repository `diogoalmiro/arms_docker`[5], was forked and modified.

Finally, the project encountered an issue when using an external tool to generate an executable. [6]

1.5 Document Outline

The document starts with section ”2. Core Concepts” introducing themes related to the project, such as, but not only, types of applications and paradigms used.

Section ”3. Requirements” follows with a vision of the final product that the solution must create. It also describes the problem using user scenarios and finishes with the functional requirements of the final product.

Then, section ”4. Analysis of the State of the Art Technology” studies available tools. This study starts by defining criteria to evaluate the tools before evaluating them.

Section ”5. Solution Proposed”, using the study of the previous section, choose the tools to use. It also describes the final solution architecture, comparing it to the first planned architecture.

With the architecture in mind, section ”6. Implementation” describes the implementation of the solution, giving insights on what was done and why and describing issues found.

Section ”7. Demonstration” is similar to the previous section. However, it describes the application of the solution to the case studies.

Lastly, section ”8. Conclusions” summarises this document, describes the achieved results and future work related to this project.

[2] See github.com/diogoalmiro/arms-node-webapp/
[3] See github.com/diogoalmiro/arms-manager/
[4] See github.com/diogoalmiro/ctray/
[5] See github.com/diogoalmiro/arms_docker/ forked from github.com/joaovizoso/arms_docker/
[6] See github.com/vercel/pkg/issues/1075#issuecomment-812796695

Chapter 2

Core Concepts

This section introduces essential concepts during the project's implementation. It describes different types of applications and the security each type entails. It explores the client-server paradigm from a developing perspective. It introduces the concept of a framework and its utility. Finally, it inspects ways to ensure the quality of a codebase, namely test-driven development (TDD) and end-to-end (E2E) tests.

2.1 Application

A software application, or application for short, is a program created to perform specific tasks. Applications are predominant in modern times, and they show up in several forms and types.

2.1.1 Categories

Concerning its installation mode, there are three main categories to classify an application[1, 2]:

- **Native Applications**: Also known as desktop or mobile applications, users install them on their devices, allowing the application to have significant access to the system.
- **Web Applications**: Users do not install the application on the computer or mobile phone but rather access them through a web browser. Nowadays, the web browser allows limited access to the system and creates a sandbox environment for the application.
- **Hybrid Applications**: Much like a native application, they require the user to install them. However, their interface functions as a web application and also require a browser. This way, the applications have access to the user's system but do not need to "reinvent the wheel" when creating an interface for multiple platforms.

Each category described has its positive aspects and drawbacks. A solution should choose the most suitable category according to its problem.

2.1.2 Permissions

Alongside the installation method, an application has different permissions on the user's device. When a user installs an application on their system, the operating system (OS) grants most permissions to the application. In mobile applications, the OS explicitly warns the user of what permissions he is granting. The user can also accept other optional permissions. In desktop applications, the OS only asks the user if they trust the application before installing it[3]. Either way, after the installation, the application receives, generally speaking, read, write and execute permissions inside the device.

However, when a user accesses a web application, the system does not warn about permissions because it uses a web browser installed on the computer beforehand. The web browser creates a sandbox environment for each web application. In order to read and write files, the web application requires an explicit request from the user, either an upload or a download, and it cannot execute files on the user's system. The browsers can later grant extra permissions to the web application environment after a clear intention from the user. For instance, allow the application to send notifications or request the camera stream.

The browser never allows code to execute outside the web application's sandbox. Hybrid applications can circumvent this restriction. When the application needs something outside the browser's sandbox, it makes a standard request to the local server, as a standard web application would do to an external server.

2.2 Client-Server Paradigm

Web applications can have a 2-layer or n-layer architecture. A 2-layer architecture is also called a client/server architecture. The generalised n-layer architecture extends the 2-layer architecture creating more layers on the server-side for a more modular approach to a problem. For instance, the server can access sub-services like a database or a legacy feature, see figure 2.1.

Regardless of the architecture, there is an evident separation of the client device and the server device. In this architecture, the client accesses the server, usually via a browser. Although the browser can perform tasks, it is essentially just the interface for the client.

Traditionally webpages would reload for every request to the server. However, nowadays, web applications request information and update their visual interface without refreshing the page, commonly known as AJAX, Asynchronous JavaScript and XML [4].

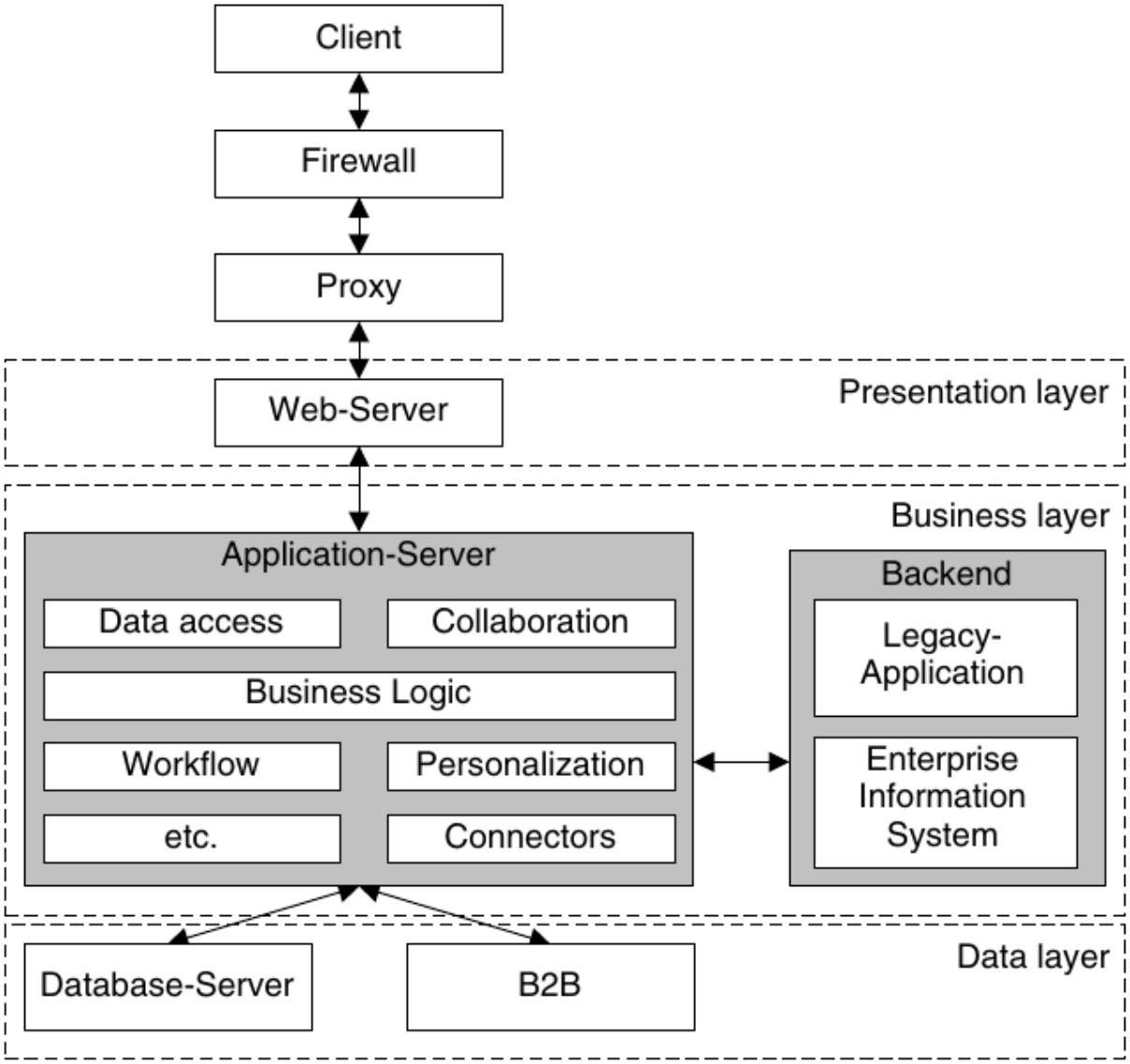

Figure 2.1: N-layer architecture for Web applications [4].

2.2.1 Interface Development

Using a browser as an interface allows a clear division of the project on a presentation and logic layers ("Business Layer" in figure 2.1). The browser interprets a HyperText Markup Language (HTML) file to display a basic web page. HTML defines the content and structure of the page. Several user-interface elements are defined, such as anchor links, buttons or text inputs. The browser automatically handles events (like mouse-clicking, selecting characters, closing the window, etc.). HTML allows the use of external technologies, such as Cascading Style Sheets (CSS) file to change the browser's presentation style, and JavaScript to define client-side tasks or create custom events handlers[5].

The trio, HTML, CSS and JavaScript, consist of technologies standardised by the World Wide Web

Consortium (W3C)[1]. They are the base of any web application[2].

On top of the project's division on developing the presentation and logic layers separately, the users also benefit from using the browser as an interface. First of all, any computer comes with a browser installed by default. Users are familiar with the browser and how to navigate between different web pages. This fact leads to a more comfortable learning process when working with new technologies that use browsers.

2.2.2 Back-end, Front-end and Full-stack Development

Concerning the division of presentation and logic layers, each programmer might only work on a layer of the n-layer architecture when developing a web application. In this case, if the layer is on the client-side, it is called front-end development. Similarly, if the layer is on the server-side, it is called back-end development[6].

When the programmer develops all of the layers, it is called full-stack development.

2.2.3 Database

A database is a collection of data structured according to a schema. There are several data storage paradigms[3]. Regardless of its structure, the database usually exists in a separate server accessed by the webserver application. Notice the "Data-Layer" in figure 2.1. The choice and management of a database also fall within the back-end development.

2.3 Framework

A framework is an artefact that helps solve a group of problems. It improves development efficiency and reduces cost by implementing several functions or classes to provide the typical behaviour for similar problems. A framework can exist for any development layer and is usually specific to a programming language. Additionally, it is usually possible to combine multiple frameworks to solve a problem[7].

To understand the potential of a framework bellow is the "Hello World"[4] code that uses the Express framework, see 2.1. This example is described by Express as follows:

> "This app starts a server and listens on port 3000 for connections. The app responds with"Hello World!" for requests to the root URL (/) or route. For every other path, it will respond with a 404 Not Found."

The Express framework is language specific to NodeJS[5], a runtime for JavaScript in the server-side. Albeit the example does not transmit the full potential of this particular framework, it shows that, without it, the code would need a little more logic, see 2.2.

[1] See www.w3.org/
[2] See archive.webstandards.org/mission.html
[3] See fireship.io/lessons/top-seven-database-paradigms/
[4] Adaptation from expressjs.com/en/starter/hello-world.html
[5] See nodejs.org/en/

Listing 2.1: ”Hello world” example.

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.json('Hello World!')
})

app.listen(port, () => \{
  console.log(`Example app listening at http://localhost:${port}`)
\})
```

Listing 2.2: ”Hello world” example without a framework

```
const http = require('http')
const app = http.createServer(server);
const port = 3000

function server(req, res){
    if( req.url == "/" && req.method == "GET" ){
        res.end(JSON.stringify('Hello World!'))
    }
    else {
        res.writeHead(404)
        res.end(`Cannot ${req.method} ${req.url}`)
    }
}

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
})
```

Without the framework, the developer needs to write and test more code to implement the same features. In the simple example, notice a need to define the logic that verifies the path explicitly. There is also the need to explicitly respond with a 404 Not Found in the else condition. Otherwise, the request would hang indefinitely until the browser gives a timeout error. The framework also changes the `req` and `res` objects to have more functionality. In this case, the framework defines a function `.json` that

simplifies the code to respond with a JavaScript Object Notation (JSON).

2.4 Test-Driven Development and End To End Tests

A test-driven development approach can be used to validate the development process, particularly unit testing. With unit tests, solution's tasks are divided into small and independent test units. With a strong unit cases base, the developer can, during the development, rely on it to ensure the quality of the solution. Each test unit has three main stages[8, 9]: First, it starts with clearly defined requirements of the task. Second, it executes the implemented code. Finally, it compares the simulated result with the expected result.

Unit tests are a local approach to ensure the quality code individually. There is also a more broad approach for testing a codebase. End-to-End (E2E) tests inspect the whole application from start to end, ensuring that everything is working as expected[10]. On a client-server paradigm, E2E focus on the user's perspective and tests how the whole application behaves[11].

2.5 Conclusions

It is possible to classify an application regarding its installation mode. This classification defines the permissions the application has on the user device. From the objectives, it is understandable that this project is related to creating hybrid applications and joining "web technology" with "desktop applications.

Another objective is to "propose a generalised reference architecture with mature technology", relating the project's solution to a framework. Furthermore, the framework must handle the separation of responsibilities between interface development, back-end development, and data management. A TDD approach can be used to ensure the quality of the framework.

Chapter 3

Requirements

This section specifies the requirements of the solution, a framework, and the final product, hybrid applications, according to each case study problem. It starts with defining the application category. Afterwards, it describes some user scenarios. Finally, it introduces the functional requirements of the application.

3.1 Vision

This project aims to create a framework that helps generate a hybrid application, specifically a local application that uses a client-server paradigm, in other words, a local-only web application[12]. The user interface is an already installed browser.

As a hybrid application, it requires the installation of a server in the user's device. This server will allow handling the data locally without requiring more user interactions and without transferring sensitive information over the internet. It also allows for the server to install any technology needed for a use case.

An additional plus for a hybrid application is that it can be configurable to allow other users inside the network to access the application, only requiring one installation.

3.2 User Scenarios

To better understand the problem, follows some descriptions of the main user scenarios of the solution. A user scenario is a design process to show at a high level how a person might act to reach a goal using a specific system[1]. A user scenario is a hypothetical situation describing the user and environment, what they want, and what they might do to achieve it. There are three similar user scenarios for each case study: the developer, the administrator, and the end-user.

[1] See www.interaction-design.org/literature/topics/user-scenarios

3.2.1 ARMS End User Scenario

John Doe is an archivist at the ARMS. He uses a personal computer to perform his job, which involves process and publishing historical documents. The originals of those documents were digitised by a specialist for the purpose of digital access. Each is digitised page by page, at 600 dpi or more, and all the images of the same document are stored in a single TIFF file, which means these files can be quite large (sometimes close to two gigabytes). They can be in any language, and it is even possible to find multiple languages on one single document. Furthermore, its content may vary between digital characters, handwritten characters, images, signatures, etc. The TIFF files can be available from a server in the local network or from an external driver that John can plug into his computer. To do his job, John starts a local application, opens the desired location on that application with the TIFF files and specifies the definitions for the OCR. Then the application performs for each document the OCR for each page generating a final PDF file as intended. Because the documents can have large numbers of pages, John can ask the application to execute those tasks without his permanent attention. Therefore, when the application ends its workflow, it sends an email to John. John then checks the results, having a particular interest in the quality of the OCR results. If the quality is too bad, he deletes all the results and reruns the process with other parameters (for example, one of the parameters he can consider is to explicitly inform the OCR engine of the languages of the text). If the results are acceptable, John can correct some OCR errors himself, editing the results. If John changes the OCR, he needs to create a new PDF, deleting the previous one. In the end, John saves all the files on the hard drive or the internal server and reports his job as done. ARMS then provides access to those PDF files to its patrons in the local network or its website.

3.2.2 IRIS End User Scenario

Jane Doe is a judge at the STJ. She uses a personal computer to perform her job, which involves analysing and making decisions over attributed legal-related documents. The documents are provided to John as a singular PDF file, which Jane must handle as a private document, not sharing it with any third party. These documents result from the sequential assembling of many original documents, most of them scanned documents. Therefore, the PDF files can range from a few hundred pages to more than a thousand, and the most usual content is images with text. Graphics, tables, or photos are typical in those documents, but most of the contexts are legal discursive text in relative terms. Jane imports the PDF file to the local application. Internally the application splits the file page by page and saves it in a local database. The application also applies OCR to each page, saves it in the database and indexes it for Jane to search. To form a decision, Jane carefully reads and annotates the document via the application. She can search the document and her annotations and browse, side by side, the original pages from the PDF file, the text of those pages that had resulted from the OCR, and her annotations. During this task of analysing a document, Jane also can correct errors from the OCR task and save it. Once she reaches a decision: she writes her decision on a regular text editor synced with the application, allowing the application to save this text on the database; she exports the decision and all the pages

of the original PDF together to deliver it. Afterwards, Jane asks the application to save the database's contents to a structure of files that can work as an archive, deleting the database so that she can use the application on a new PDF file. If necessary, the application also can import the archive and make it possible to work on it again.

3.2.3 Administrator Scenario

Sandy Doe is an administrator for the job location of the end-user. He manages the computers used in his location. His task is to install and assist any problem on the end user's computer. He downloads and installs the application that the end-user needs to perform his job.

3.2.4 Developer Scenario

Richard Roe is a developer of the application an end-user requires. He understands the end-user requirements and creates the application using the framework. He also maintains the application and receives feedback from the administrator and the end-user.

While developing the framework solution, we put on the application developer's shoes to understand the framework's requirements.

3.3 Functional Requirements

The user scenarios help with understanding what characteristics the framework should contain. For instance, it should be configurable for each scenario. In other words, the application should be easily adaptable for the workflow of the end-user.

Next, a list of characteristics the application has for each user scenario. This list is not fixed and can change as both projects are still in development phases, IRIS in a very early stage and ARMS in an almost finished stage.

- For ARMS:
    1. Select a TIFF files' location.
    2. Define OCR settings (e.g. specify dictionaries).
    3. Execute a bulk OCR on a location.
    4. Notify the user.
    5. See TIFF files.
    6. See OCR metadata.
    7. Edit OCR results.
    8. Regenerate a PDF from manually edited OCR.
- For IRIS:
    1. Select a PDF document to use.

2. See the document and annotations.
3. Exectue OCR on demand.
4. Annotate the document.
5. Correct the OCR.
6. Synchronise an external editor to the application.
7. Export an archived document.
8. Import an archived document.

3.4 Conclusions

The idealised solution for the problem is a framework that should be versatile and allow the creation of local-server applications. The choice of a hybrid application comes from the necessity of using web technology and ensuring that sensitive information does not travel outside the private network.

There are a lot of different use cases for each case study. However, both relate as they mainly focus on managing documents and executing OCR. Besides meeting use cases' requirements, the application should be "end"-user friendly.

Chapter 4

Analysis of the State of the Art Technology

This section studies the use of frameworks for web applications, more precisely, back-end solutions. It starts by defining some reasoned criteria and then evaluates several frameworks based on those criteria. It also addresses a specific framework and why it is not suitable for the solution. Finally, it refers to a specific type of database, embedded databases. This study fundament the choices made in the next section.

4.1 Criteria to Evaluate Frameworks

There are several programming languages and frameworks available that to use. However, to select the most fitting framework for the solution, some criteria need to be identified first. Follows a list of references used to define some criteria:

S1: An answer to the question "How can I choose a web framework?" on the sharing knowledge platform Quora[1].

S2: A post on choosing frameworks on the platform Hacker Noon, built for technologists to read, write, and publish[2].

S3: A description of considerations to have when choosing a framework on the learning platform Envato Tuts+[3].

S4: A publication on choosing a web framework by a company focused on web-technologies, CYCLE[4].

S5: Ten criteria defined by a PHP framework, symfony[5].

S6: A paper on choosing a web framework based on free, open-source software[7].

[1] See www.quora.com/How-can-I-choose-a-web-framework
[2] See hackernoon.com/how-to-choose-a-framework-ea8b5b1e1f44
[3] See https://code.tutsplus.com/tutorials/15-important-considerations-for-choosing-a-web-dev-framework–net-8035
[4] See cycle-interactive.com/s/choosing-web-framework
[5] See symfony.com/ten-criteria

S7: A whitepaper on choosing JavaScript frameworks for web applications[6].

S8: A comparison table to choose a JavaScript framework aimed at the Drupal, a PHP framework, community[7].

The following table, 4.1, can be created after identifying several criteria from the previous sources. Each criterion is sorted by its score, which is how many sources refer to it.

Table 4.1: Criteria to choose a framework used by the sources, sorted by the score of how many sources refer to the criterion.

Criteria	S1	S2	S3	S4	S5	S6	S7	S8	Score
Popularity & Community	x	x	x	x	x	x	x	x	8
Maintainability & Support	x	x	x		x	x	x	x	7
Documentation & Examples	x		x		x	x	x	x	6
Use Cases	x		x		x	x	x	x	6
License			x		x	x	x	x	5
Learning Curve			x			x	x	x	4
Language	x			x		x			3
Stack Paradigm						x	x	x	3
Performance		x					x		2
Software Pattern			x		x				2
Database & Object Relational Mapping			x			x			2
Installation			x				x		2
Testability			x					x	2
Convention over Configuration				x				x	2
Security					x		x		2
Schedule		x							1
Core Library			x						1
Scalability			x						1

The criterion with bold on table 4.1 are criteria that had at least half of the sources refer to it. The study was reduced to those six criteria as they appear to be enough to fundament the choices and to reduce the time spent on the study.

This report uses the most used criteria, with the higher scores, to evaluate frameworks. A description of each criterion selected follows:

- **C1, popularity and community:** The number of people interested in the framework. Contributing

[6]See softarchitect.files.wordpress.com/2018/03/choose-the-right-javascript-framework-for-your-next-web-application_whitepaper1.pdf

[7]See dri.es/files/javascript-framework-comparison-january-2016.pdf

to it either in its development or answering questions of it.

- **C2, Maintainability and support:** Age of the framework and interest to keep it evolving.
- **C3, Documentation and examples:** Simplicity of the documentation and quantity of examples, snippets, articles and tutorials on the framework.
- **C4, Use cases:** Where is the framework used in the real world.
- **C5, License:** What type of license does the framework have. If it can use it for the project.
- **C6, Learning curve:** Line shape of the difficulty over time for any developer to learn the framework.

The following section evaluates some frameworks using each criterion.

4.2 Evaluation of the Frameworks

Given the first criterion, C1, and to reduce the search, this study analyses only the most popular back-end web frameworks of 2020, based on StackOverflow's survey [8]. This filter focuses the evaluation on the following frameworks: Express, ASP.NET Core, Spring, Flask, Django, Laravel, and Ruby on Rails. In the following paragraphs, there is a short description of each framework.

Express: The most popular back-end framework. It allows for quick API creation in NodeJS and is small in size. However, it has limited functionality because it is "unopinionated", and the developer can extend it[13].

ASP.NET Core: The second most popular web framework. It is an open-source, multi-platform framework that helps to create web applications on the .NET platform [9].

Spring: The most popular Java web framework. Spring focuses on speed and simplicity[10].

Flask: It is a lightweight Python web framework[11].

Django: It is also a Python web framework. It aims for rapid development and a pragmatic design[12].

Laravel: It is a PHP web framework. It provides a structure and a starting point for the application[13]-

Ruby on Rails: It is a Ruby web framework. It includes everything needed to create an application[14]. Its early popularity influenced several other frameworks[7].

From the fact that the previous frameworks are StackOverflow's most popular frameworks, they achieve criterion C1. To better understand their popularity, this study also looks at the number of Github stars. Table 4.2 contains the percentage of StackOverflow Users and the number of Github stars.

[8] See insights.stackoverflow.com/survey/2020
[9] See dotnet.microsoft.com/learn/aspnet/what-is-aspnet-core
[10] See spring.io/why-spring
[11] See palletsprojects.com/p/flask/
[12] See www.djangoproject.com/
[13] See laravel.com/
[14] See rubyonrails.org/

Table 4.2: Popularity of Frameworks.

C1	Express	ASP.NET Core	Spring	Flask	Django	Laravel	Ruby on Rails
SO Users (%)	**21**	20	17	14	13	11	7
Stars (thousands)	51	21	41	53	55	**63**	47

As depicted in figure 4.1, that the number of Github stars does not depend on the number of StackOverflow users. The critical difference is for Laravel and ASP.NET Core, respectively, the most and the least stars. However, in terms of community size, this difference can be disregarded.

Concerning C2, this study analyses each framework initial release date and last release date in table 4.3.

Table 4.3: Frameworks release date and last update.

C2	Express	ASP.NET Core	Spring	Flask	Django	Laravel	Ruby on Rails
Initial Date	2010	2016	2002	2010	2005	2011	2004
Update date	2019	2020	2020	2020	2019	2020	2020

Most frameworks are actively updated. The exceptions are Django and Express. However, looking closer, Django continues to have recent Github commits on their repository. Although Express repository does not have many commits, Express's organisation has a series of related projects used by Express that are still active [15]. Therefore, all frameworks are considered to be continuously active and maintained.

Regarding C3, the most popular frameworks have good documentation and multiple online articles about it. Even when the official documentation is not excellent, many community examples or articles can help developers with any framework.

> "There is rarely a reason to choose a framework without a great community, great documentation, and a mature set of companies using it to run successful production applications" [7]

Therefore, any of the frameworks have good documentation. There are also many examples, tutorials, and "Getting Started" pages for every framework with a quick search on the internet.

For C4, there are a lot of recognisable companies using each framework on a day to day basis. Here is a non-exhaustive list of examples:

- Express by IBM, Fox Sports, Accenture[16]
- ASP.NET by GoDaddy, StackOverflow[17]

[15] See techsparx.com/nodejs/news/2019/express-not-dying.html
[16] See expressjs.com/en/resources/companies-using-express
[17] See dotnet.microsoft.com/platform/customers

- Spring by Netflix, Mercedes, Target.
- Flask by Reddit, Uber, Airbnb[18]
- Django by Pinterest, Instagram, Mozilla[7].
- Laravel by BBC, Pfizer[19]
- Ruby on Rails by Github, Shopify[7].

Table 4.4: License for each framework.

C5	Express	ASP.NET Core	Spring	Flask	Django	Laravel	Ruby on Rails
License	MIT	Apache 2.0	Apache 2.0	BSD	BSD	MIT	MIT

For C5, the table of licenses for each framework, 4.4, shows that every framework is open source, and, between the licenses, there are no significant differences that affect the project. All grant modification, distribution, commercial and private use, no warranty and no liability [20].

C6 is where the frameworks differentiate the most. The learning curve of a framework connects directly to the programming language used. Another factor that affects it negatively is if the framework uses convention over configuration, see table 4.5. For instance, by forcing a project structure, the framework can be harder to learn [21]. By being "unopinionated", Express does not force any project structure. Flask is also more relaxed in this feature. The remaining frameworks focus on convention over configuration and have a prominent project structure, usually with tools helping generate the system.

Table 4.5: Use of "Convention Over Configuration" in the frameworks.

C6	Express	ASP.NET Core	Spring	Flask	Django	Laravel	Ruby on Rails
License	no	yes	yes	no	yes	yes	yes

[18] See github.com/rochacbruno/flask-powered
[19] See donatix.net/top-companies-use-laravel/
[20] See choosealicense.com/licenses/bsd-2-clause/
[21] See facilethings.com/blog/en/convention-over-configuration and devopedia.org/convention-over-configuration

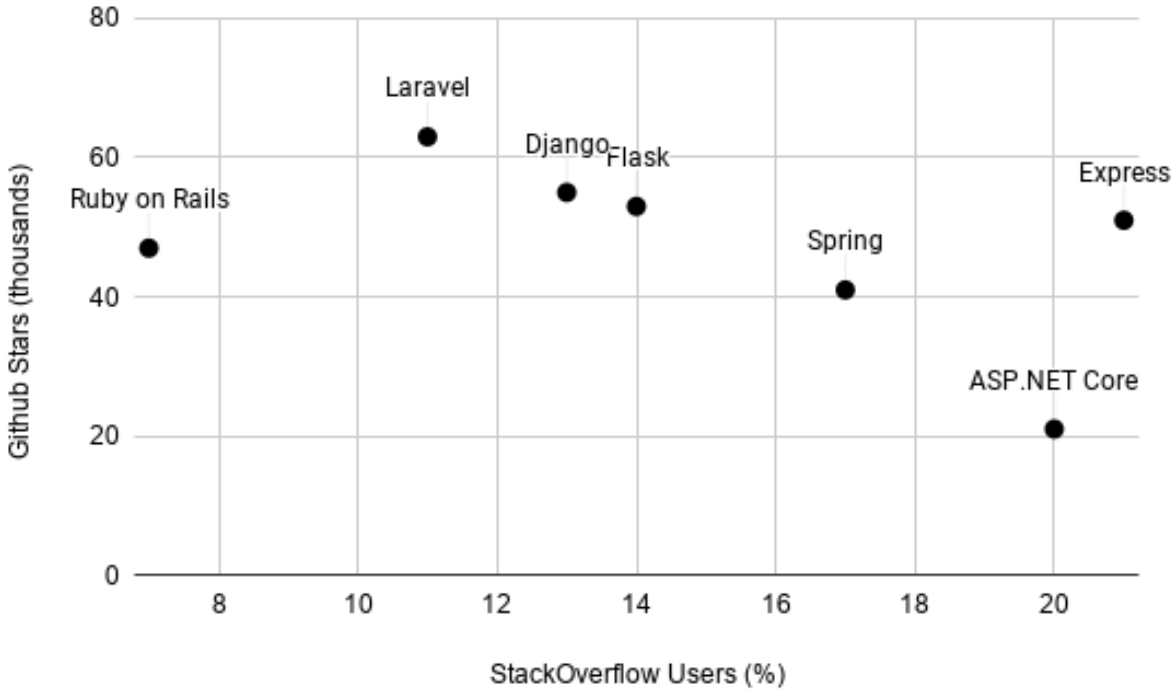

Figure 4.1: Number of StackOverflow users and Github stars for each framework

4.3 The Special Case of the Electron Framework

Electron[22] is a framework that allows developers to build native applications using web technologies. When a user installs an Electron application, it also installs a built-in version of the Chromium browser, making the application require more storage to install. Using Electron would also prevent having a distributed application within a network as it does not automatically run a server.

There is also a user experience problem with this tool. The user is used to a series of features already implemented by the browser that would need to be rewritten or disregarded, such as the browser's typical navigation elements, like the back button or the home button, or the possibility to navigate between webpages [23].

For the reasons mentioned above, Electron is not suitable for the framework. This project could create a better solution using a more straightforward web framework to support a browser interface.

4.4 Embedded Database

As previously mentioned in section 2.2.3, usually databases are external programs that an application accesses. Using an external database would mean the need of having an internet connection to the

[22] See www.electronjs.org/
[23] See clojureverse.org/t/making-a-desktop-app-vs-local-webserver-vs-electron/2261/6

database server or having a different server for the database consistently running in the background of the user's device, spending more resources than needed.

Using a particular type of database, an embedded one can exclude the previous problem. With them, there is no external server. Instead, the database coexists inside the webserver application. It can also remove the extra processing time of continuously running a database and have a data connection between servers using an embedded database. [14, 15] An example of an embedded database is SQLite, written in C has several bindings for different programming languages.

4.5 Conclusions

There exist several frameworks that the solution could use. This study explores the most popular back-end ones, soo that one can be chosen with a reasoned basis. It also specifies that the data management tool to use should be an embedded one to minimise resources used and avoid data transfers. Regarding the requirements, although they enforce web technology on the front-end, there are no particular server-side languages to use.

Chapter 5

Solution Proposed

This section presents the idea for the solution as a framework and the several elements around it. It starts by using the previous study to choose an existing framework to extend. Then, it describes the solution's architecture, comparing it to the planned architecture. Finally, it addresses the selection of a database.

5.1 Framework Decision

Given the comparisons in the previous section, Express is the most suitable framework to develop the application's back-end. Previously, it was already shown a few capabilities of this framework with the example, in section ”2.3. Framework”.

All frameworks are relatively alike, with similar popularity and community. However, Express's “un-opinionated” characteristic allows it to have a more flexible project structure. Since there is a need to configure the application for each use case, it sounds more reasonable to be able to configure the project as well. The project does not require any specific language for the server-side. Albeit, Flask would also give the configuration freedom. The fact that Express extends the functionality of NodeJS allows having a consistent language (JavaScript) in both the front-end and back-end of the project and a consequent JSON communication between both layers.

Another factor to use the Express framework is the community around the NodeJS runtime and JavaScript language. On the one hand, looking again at StackOverflow's top web frameworks, from the top five most used frameworks, notice that only ASP.NET is not JavaScript based.

On the other hand, JavaScript developers also have an essential tool for community sharing code, the Node Package Manager (NPM). NPM is a handy tool and the centre of sharing JavaScript code. It improves the community around JavaScript, enabling a centralised repository for JavaScript technologies. Each technology is open-source and consists of a package. This package defines several attributes, like the name, author, and, more importantly, other NPM packages' dependencies. It allows any developer to share their JavaScript technology and to use any technology.

An open-source project has a positive aspect that the community reports issues and suggests and

contributes a lot, allowing it to evolve rapidly.

NPM allows a developer to know of any vulnerability and fix them, if possible, of packages he is dependent on by requesting it to NPM[1].

It is also worth noticing other positive aspects of the NodeJS runtime. As already said before, it uses JavaScript, which is an event-driven and non-blocking I/O language. This fact makes it very lightweight and efficient. There are also several big companies contributing to its development. Concerning the NodeJS runtime, it is also worth noticing that it uses an event-driven non-blocking I/O model. This fact makes it very lightweight and efficient. There are also large companies in the web industry that contribute to its development. [16]

5.2 Solution Architecture

There is a need to run a server on the user's device to create a local-only web application. Then, to access the application, the user opens the browser using an URL similar to `http://localhost/`. The initially planned architecture consisted of the server reacting to a configuration file in order to define the URL routes of the Express framework, the actions he needs to perform, the external technologies to use, or the information to store on an embedded database. The remaining operations inside the server would return general static pages to the browser, as depicted in figure 5.1.

After reviewing this architecture, the approach on the server-side was changed to a more modular view of the problem, see figure 5.2, segregating the application's components and having an unopinionated view over the database usage, enabling the reuse of components for other applications.

5.2.1 Database Decision

Previously, the study also did a short study to find a suitable embedded database, which runs on NodeJS alongside the server. To choose it, it considered databases available inside NPM and considered its popularity, size and data paradigm to select one.

Specifically, it was interested in either a document-based database or a relational database.

Document-based databases are more general and easier to use [2]. They would also allow the use JSON in all layers of the application[17].

Relational databases are the most traditionally used databases where the data is organised in tables and use keys to make relationships between tables[17].

Since there were no specific requirements for databases, it compared two embedded solutions inside the npm community, SQLite3 [3] and LokiJS [4]. The first one is a relational database, while the second is a document-based one. Table, 5.1 compares SQLite3 and LokiJS in terms of age (initial release's year), maintainability (last update's year), size and popularity (Github stars)[5].

[1] See www.npmjs.com/
[2] See www.youtube.com/watch?v=W2Z7fbCLSTw
[3] See www.npmjs.com/package/sqlite3
[4] See www.npmjs.com/package/lokijs
[5] See www.npmtrends.com/sqlite3-vs-lokijs

Table 5.1: Comparison of SQLite3 and LokiJS.

Database	Paradigm	Initial Date	Last Update	Size (KB)	Stars
SQLite3	Relational	2011	2020	240.9	4617
LokiJS	Document-based	2013	2020	19.3	5595

The two databases are similar in most aspects. LokiJS is, albeit relatively smaller in size, also gives the possibility of using JSON throughout all application layers.

The study concluded that a database reasonable to use was LokiJS. However, during the implementation, it was rapidly concluded that it did not require databases for the case studies during the implementation. Therefore, each component should be responsible for choosing its database. Albeit, It is still recommended the use of an embedded database.

5.2.2 Reusable Components

Each component associated with the application is responsible for bridging the browser interface and the external technology. It is also responsible for choosing the best database, if any, for the problem it solves. They should solve just one problem or a small group of related problems. With a well-defined interface between the framework and the component, a component can also be reused on different applications.

5.2.3 Front-end

Although there were no studies about front-end frameworks, some were chosen on a similar popularity basis. Regarding CSS and Javascript, Bootstrap[6] and PopperJS[7], which Bootstrap might use, should be available, allowing the reusable components to have a set of well-defined CSS classes to use.

Regarding server-side rendering, Pug[8] helps to have a similar structure throughout the application using the extend and include capacities.

5.3 Conclusions

The study found Express as the most suitable framework to use as the base of the solution. Express uses NodeJS, which is an open-source project that continues to improve rapidly. The planned architecture for the solution is very modular, allowing the developer to create reusable components. Furthermore, although each component can use what is most suitable, a recommended database to use is SQLite3. Regarding the front-end Bootstrap, PopperJS and Pug were chosen. The modularity of this framework ensures the possibility to use it for several use cases.

[6] See getbootstrap.com/
[7] See popper.js.org/
[8] See pugjs.org/api/getting-started.html

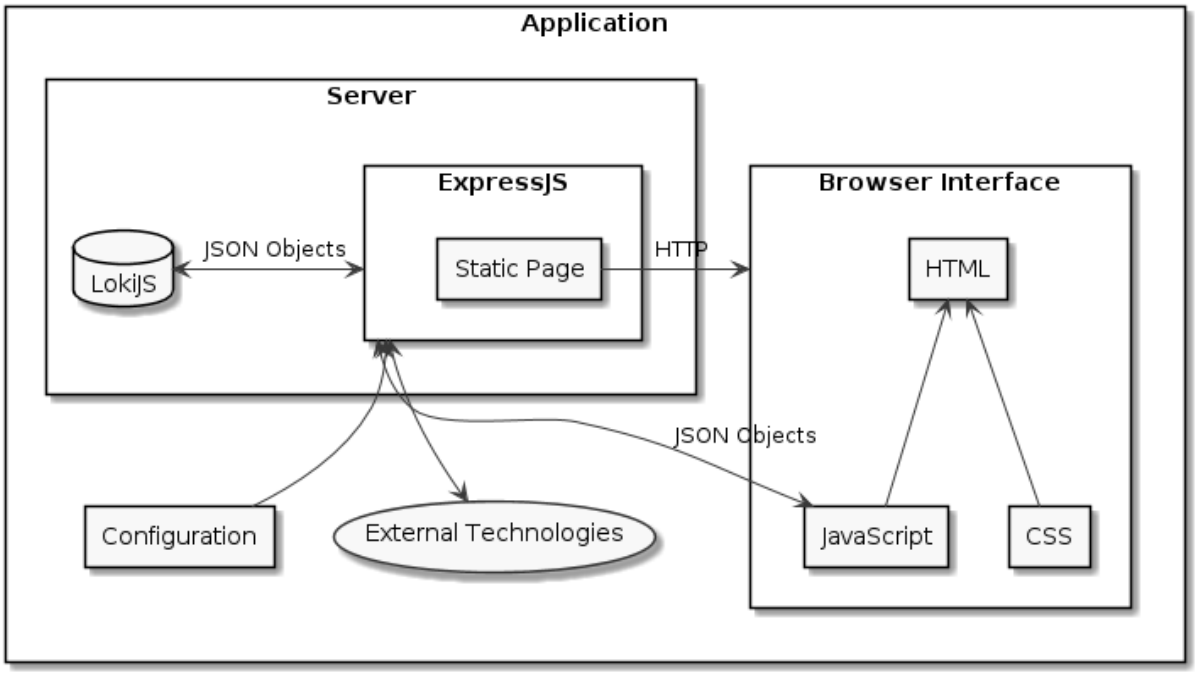

Figure 5.1: Previous framework architecture overview.

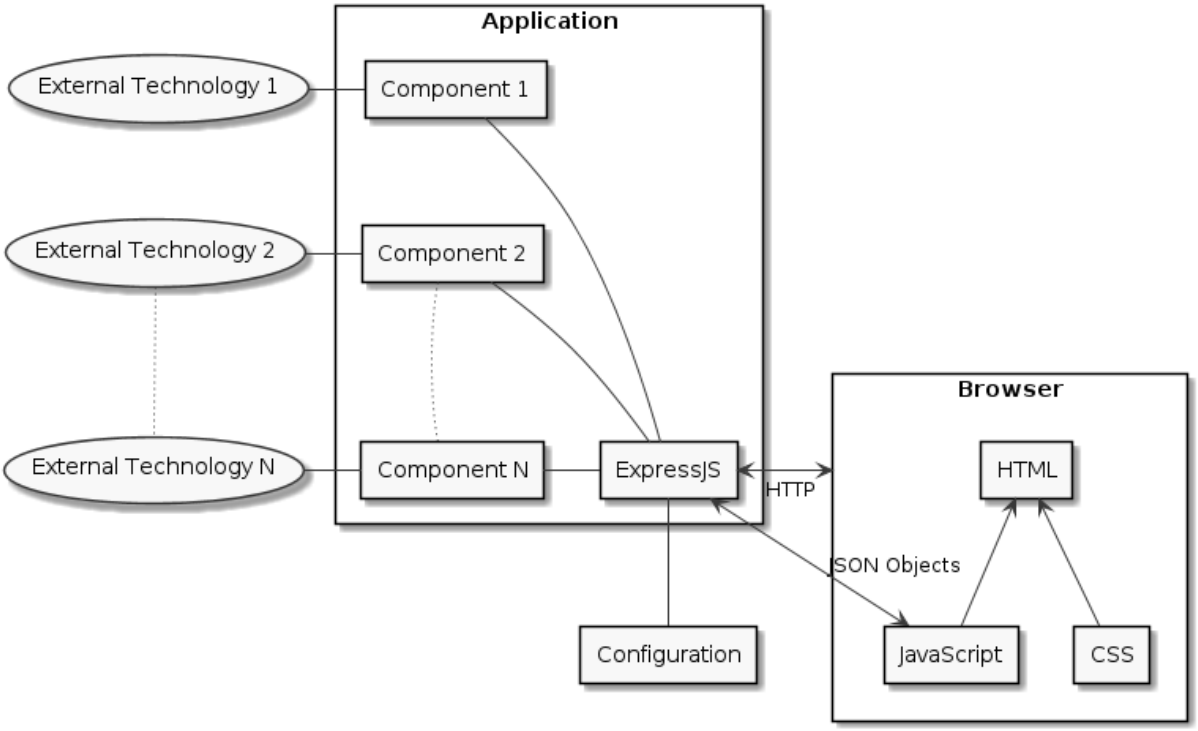

Figure 5.2: Reviewed framework architecture.

Chapter 6

Implementation

This section describes the implementation of the solution. It starts with a short description of the NodeJS and NPM environments and how to use them. Afterwards, it discusses the core of the framework and its structure, referring to the main class, the component class, reusable components and tests used. It also talks about a nice-to-have feature built natively.

6.1 Environment

The solution was mainly developed on a Linux OS machine. However, it was also tested on a Windows machine. A relatively recent version of NodeJS with a "long-term support" until April 2024 was used[1]. The Node Package Manager, NPM, is used to manage the solution's dependencies and comes bundled with the NodeJS installation, see 6.1.

Listing 6.1: Environment versions.

```
1 $ node -v          $ npm -v
2 v16.10.0           7.24.1
```

To create a new NPM package, run: `$ npm init` on a new empty folder. Followed with `$ npm install <dependency-name>` within the folder to install a dependency on the new package. The new directory after the previous commands contains three entries:

- `node-modules/`: A folder containing all installed dependencies.
- `package.json`: A file containing metadata of the project.
- `package-lock.json`: A file similar to `package.json` but with more rigid control on the dependencies to ensure different machines use the same version.

[1] See nodejs.org/en/about/releases/#releases

6.2 Webfocus Framework

The framework is structured into an “app” and a “component” package. The first package defines the `WebfocusApp` class, while the second defines the class `WebfocusComponent`. There are also three packages with reusable components: “send-mail”, “util”, and “tray”. Use `$ npm install @webfocus /<package-name>` to install and use them. See figure 6.1 for an overview of the framework packages and classes. With this structure, it is possible to implement components without requiring the whole application package.

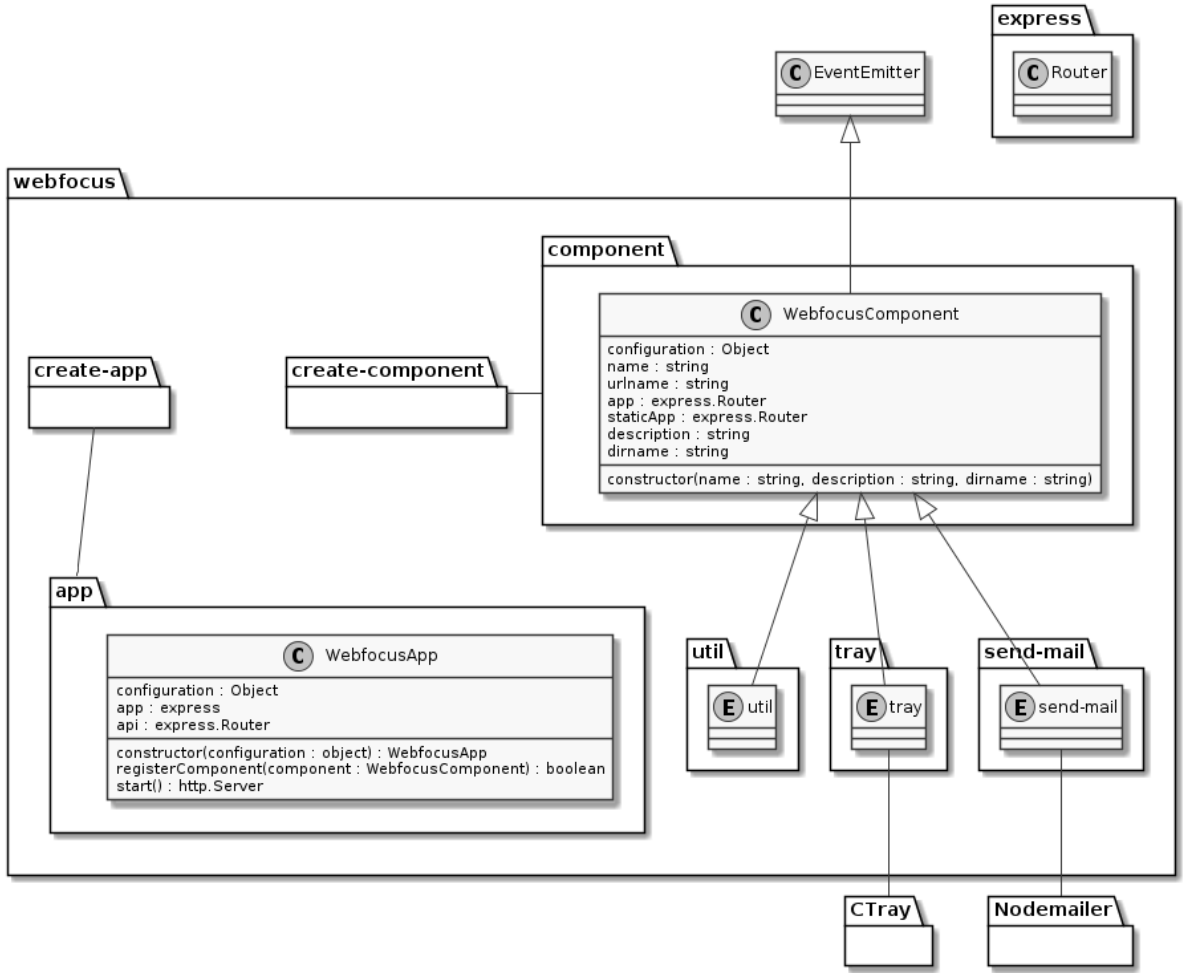

Figure 6.1: Webfocus Framework overview.

6.2.1 App

To start implementing this package, install the following dependencies explicitly:

- “bootstrap”: containing a front-end framework[2].
- “debug”: containing a debugging utility[3], that express also uses.

[2]See https://getbootstrap.com/docs/5.1/getting-started/introduction/
[3]See www.npmjs.com/package/debug

- “express”: containing the web framework[4].
- “pug”: containing a template engine for server-side rendering (SSR) [5].

It also requires to install some development only dependencies, described later on the Test-Driven Development section:

- “mocha”: containing a test framework[6].
- “supertest”: containing another test framwork for HTTP requests[7].

This package only exports the `WebfocusApp` class. When creating an instance of this class, the constructor receives a configuration object. This object is accessible from the configuration property and has the following overwritable default values:

- `port: 0`, the port number where the application server will be listening.
- `name: "Default Application Name"`, the name the end-user sees.
- `dirname: path.join(__dirname, "views")`, a path to the `views` directory of the package.
- `static: path.join(__dirname, "static")`, a path to the `static` directory of the package.
- `components: []`, the array of components[8].

Any extra values are saved to the configuration. The line of code that creates the configuration uses `Object.assign` as follows `this.configuration = Object.assign({}, DEFAULT_VALUES, configuration)`. As “The properties are overwritten by other objects that have the same properties later in the parameters order.”[9].

The constructor also initialises and setups the `express` object - `app` and an `express.Router` - `api` (lines 1 and 2 of 6.2). Enabling the following behaviours:

- The SSR engine is set to pug, and its default directory is set to the value of `configuration.dirname` (lines 5 and 6).
- The requests are routed[10] to either the `app` or the `api`, if the request path starts with `/api`, the body request is parsed before sending them to the `api` router (line 9), see figure 6.2 for the initial routing diagram.
- If the request exactly matches with `GET /api/`, it responds with the names for each registered component in a JSON format(line 12 to 15).
- If the request exactly matches with `GET /`, it responds with the SSR of `layouts/index.pug`, which is inside the default directory of SSR (lines 17 to 20).

[4] See www.npmjs.com/package/express
[5] See www.npmjs.com/package/pug
[6] See www.npmjs.com/package/mocha
[7] See www.npmjs.com/package/supertest
[8] The “components” array is always transformed into an empty array empty even if it had a value.
[9] See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign
[10] “Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).” from expressjs.com/en/starter/basic-routing.html

Listing 6.2: Initialise express routers.

```
1  this.app = express();
2  this.api = express.Router();
3
4  // Enable Server-Side Rendering for files within the configuration folder
5  this.app.set('view engine', 'pug');
6  this.app.set('views', this.configuration.dirname);
7
8  // Enable JSON and HTTP-form-submit communication
9  this.app.use("/api", [express.json(...), express.urlencoded(...), this.api
       ])
10
11 // Express initial handlers
12 this.api.get("^/$", (req, res, next) => {
13     debug("Route Api Handler")
14     res.json(this.getAllComponentNames())
15 })
16
17 this.app.get('^/$', (req, res, next) => {
18     debug("Route App Handler")
19     res.render('layouts/index', ...)
20 })
```

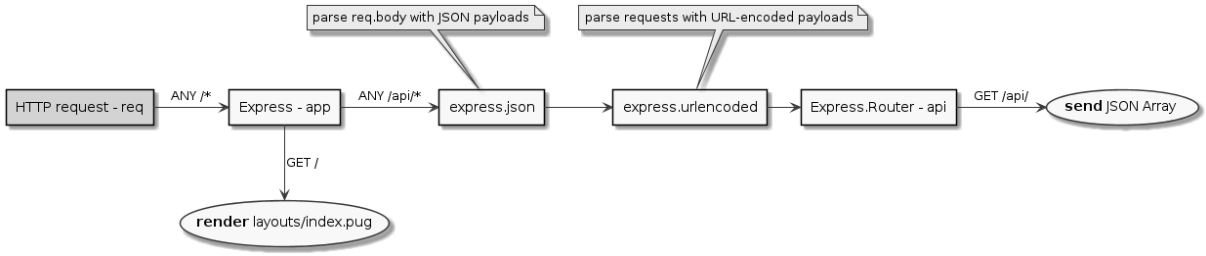

Figure 6.2: Initial routing diagram.

When a component is registered to the `WebfocusApp` instance, the following happens: (See 6.3)

- The instance is emmited to the component (line 1).
- A handler for `/api/<component-urlname>` is defined within the `api` router to use the `component.app` router (line 3).
- A handler for `/<component-urlname>` get HTTP requests is defined within the `app` router to respond with the first success call (line 5):

1. Using `component.staticApp` (line 5).
2. Rendering the `requested/path.pug` file within the `component.dirname` (line 11).
3. Render `index.pug` within the `component.dirname` (line 11 and 17).
4. Call the next handler with an optional error object (line 8, 14 and 20).

Listing 6.3: Registering a component.

```
1  component.emit("webfocusApp", this);
2
3  this.api.use(`/${component.urlname}`, component.app);
4
5  this.app.use(`/${component.urlname}`, component.staticApp, (req, res, next)
       => {
6      // Ignore POST, PUT requests
7      if( req.method !== 'GET' && req.method !== 'HEAD' ){
8        return next();
9      }
10     ...
11     res.render(path.join(component.dirname, subpath), ..., (err, html) => {
12         if( err ){
13             if( subpath == "/index" ){
14                 next(err);
15             }
16             else if( err.message.indexOf("Failed to lookup") >= 0 ){
17                 res.render(path.join(component.dirname, 'index'), ...);
18             }
19             else{
20                 next(err);
21             }
22         }
23         else{
24             res.send(html);
25         }
26     })
27 })
```

After registering all required components, the server can start to listen. Before start listening for HTTP requests, the instance emits the configuration object to the components, see line 1 of 6.4 and define a few more handlers:

- `api` not found handler (line 4-6)
- `api` error handler (line 7-9)
- `app` reserved handlers:

1. `/webfocus-static/` for files on the `path.join(__dirname,"static")` directory (line 12).
2. `/bootstrap/` for the bootstrap framework static files (line 14).
3. `/popperjs/` for the popperjs framework static files (line 15).
4. For serving the `this.configuration.static` directory (line 18).
5. For render requests the not found, not allowed or with internal errors using `layouts/error.pug` (lines 20 to 30).

The `/webfocus-static/` handler ensures that the files are from within the application folder and not from a different location that might have been redefined while instantiating. This handler ensures any component access to, for instance, `static/js/fetch.js`, which contains some client-side utilities to make AJAX calls with JSON with ease.

See figure 6.3 to check the final routing diagram.

Listing 6.4: Starting server.

```
1  this.emit("configuration", this.configuration);
2
3  // Last express handlers
4  this.api.use((req, res, next) => {
5      res.status(404).json({error: `API Endpoint ${req.path} not found.`})
6  });
7  this.api.use((err, req, res, next) => {
8      res.status(500).json({error: err.message, stack: err.stack})
9  })
10
11 // Ensure webfocus-static, bootstrap and popper files are always available
       (such as fetch)
12 this.app.use('/webfocus-static/',express.static(path.join(__dirname, '
       static')));
13
14 this.app.use('/bootstrap/', express.static(...))
15 this.app.use('/popperjs/', express.static(...))
16
17 // Serve static files under the static folder
18 this.app.use(express.static(this.configuration.static));
19
20 this.app.get("*", (req, res, next) => { // Not found handling
```

```
21      res.status(404).render('layouts/error', this.pugObj({req, error: `Not
        found ${req.path}`}));
22  })
23
24  this.app.all("*", (req, res, next) => { // Method Not Allowed handling
25      res.status(400).render('layouts/error', this.pugObj({req, error: `
        Method not allowed (${req.method})`}));
26  })
27
28  this.app.use((err, req, res, next) => { // Internal error handling
29      res.status(500).render('layouts/error', this.pugObj({req, error:err.
        message, stack: err.stack}));
30  })
```

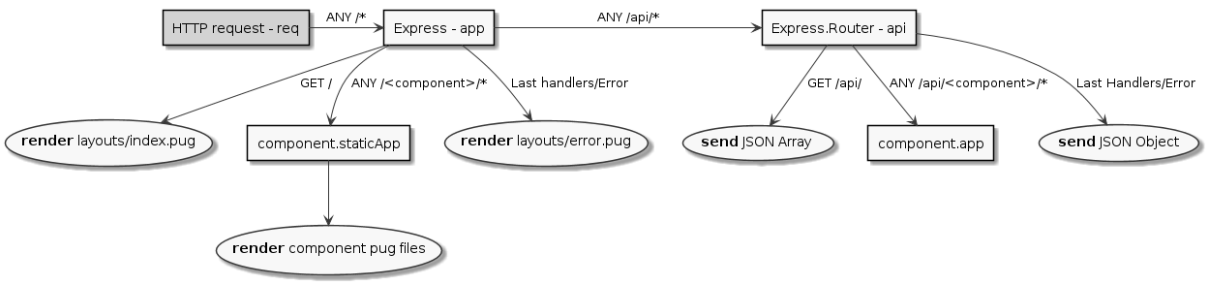

Figure 6.3: Final routing diagram.

The `configuration.static` directory specifies the location for statically sending files to the browser. The `configuration.dirname` specifies a location that defines the HTML when using SSR. This property allows the developer to create different user interfaces for different applications. The folder it specifies must have the following structure:

- `layouts/` a folder.
  - `error.pug` the page to render on any error response.
  - `index.pug` the page to render for a get `/` response.
  - `main.pug` an extendable file for WebfocusComponents to use, must define the blocks `head` and a `main`, see 6.5 lines 5 and 10.

Listing 6.5: Default main.pug file.

```
1  doctype html
2  html
```

```
3   head
4     //...
5     block head
6   body
7     header
8       //...
9     main
10      block main
11    footer
12      //...
```

6.2.2 Component

To start implementing this package, install the following dependencies explicitly:

- "app-data-folder": containing a utility to get the APPDATA path in any OS[11].
- "debug".
- "express".

It also needs the installation of the same development only dependencies as the "app" package.

This package defines the `WebfocusComponent` class and a utility function `createComponent` to call the constructor with fewer arguments.

The class's constructor requires a name, description and the directory of the component. The function `createComponent` only requires the name and description, and it gets the folder automatically, avoiding the need to write `__dirname` and **`new`** explicitly, see 6.6.

Listing 6.6: Create component function versus constructor.

```
1 fuction createComponent(name, description){
2     // https://github.com/detrohutt/caller-dirname/blob/master/src/index.ts
3     const _ = Error.prepareStackTrace;
4     Error.prepareStackTrace = (_, stack) => stack;
5     const dirname = path.dirname(new Error().stack.find(s => s.getFileName
      () != __filename).getFileName());
6     Error.prepareStackTrace = _;
7     return new WebfocusComponent(name, description, dirname);
8 }
9
```

[11] A local directory that might contain data associated with this component, it needs to be created if needed. See www.npmjs.com/package/app-data-folder

```
10 let component = new WebfocusComponent("Component 1", "Description",
       __dirname);
11 // vs
12 let component = createComponent("Component 1", "Description);
```

An instance of this class will have several properties. The most interesting properties cases are:

- `staticApp` - the `express.Router` that handles `/<component.urlname>` requests before any SSR.
- `app` - the `express.Router` that handles `/api/<component.urlname>` requests.
- `urlname` - the transformation of the component's name to lowercase with dashes replacing the spaces, also known as kebab-case.
- `configuration` - the read-only property of the app's configuration (undefined before the emission of the `"configuration"` event).

If `staticApp` does not handle the request, the component needs at least an `index.pug` file on its directory to have SSR applied. However, the component can create a different PUG file to handle other requests:

For example, assume the following files a component with name `c` that does not redefine `staticApp`:

- index.js
- index.pug
- foo.pug
- bar.html
- sub/
  - bar.pug

Table 6.1 shows the actions that would happen for a specific request.

Table 6.1: Examples of actions for some requests.

Method	URL	Action
GET	`/c/`	**render** index.pug (Note that it would **send** index.html if available)
GET	`/c/foo`	**render** foo.pug
GET	`/c/bar`	**render** index.pug (There is no bar.pug)
GET	`/c/bar.html`	**send** bar.html
GET	`/c/sub/bar.html`	**render** index.pug
GET	`/c/sub/bar`	**render** sub/bar.pug
ANY		**render** /layouts/error.pug from app with a 400 not allowed.

6.2.3 Creating a Component Instance

Creating a new component is as simple as creating a new package and the file `index.js` with the contents in 6.7. Afterwards, define the handlers for the requests `/api/first-component/` via `component.app`, see 6.8.

By default, the static handler sends all files from the component directory. This behaviour can be redefined by overwriting the `staticApp` property, line 1 of 6.9.

Listing 6.7: Create component, index.js file.

```
1 module.exports = require('@webfocus/component')("First Component", "My
    First Component");
2
3 // OR Keeping a component reference for easier use:
4 let component = module.exports = require('@webfocus/component')("First
    Component", "My First Component");
```

Listing 6.8: Create component, index.js file.

```
1 component.app.get("/", (req, res) => res.json("Hello From First Component")
    ) // GET /api/fist-component/
2 component.app.post("/", (req, res) => res.json("Hello From First Component"
    )) // POST /api/first-component/
3 component.app.get("/sub-path", (req, res) => res.json("Hello From First
    Component")) // GET /api/first-component/sub-path/
```

Listing 6.9: Redifining staticApp property, index.js file.

```
1 component.staticApp = express.Router();
2
3 component.staticApp.get("/", (req, res) => res.end("Hello From First
    Component Static Router")) // GET /first-component/
```

To define the component front-end either:

1. Create a `index.pug` file that might `extends /layouts/main`.
2. Create a `index.html` file.
3. Redefine the staticApp router to handle `/`.

Using `index.pug` extending `/layouts/main` can guarantee that every component looks similar, having the same headers and footers through the application.

The initialiser `$ npm init @webfocus/component` can be used to generate a primary template component. It creates the primary files automatically and installs the `@webfocus/component` dependency (see figure 6.4).

Listing 6.10: Simple index.pug example

```
1 extends /layouts/main
2
3 block head
4   script.
5     windows.addEventListener("load", alert("Page Loaded!"))
6
7 block main
8   p Hello From First Component.
```

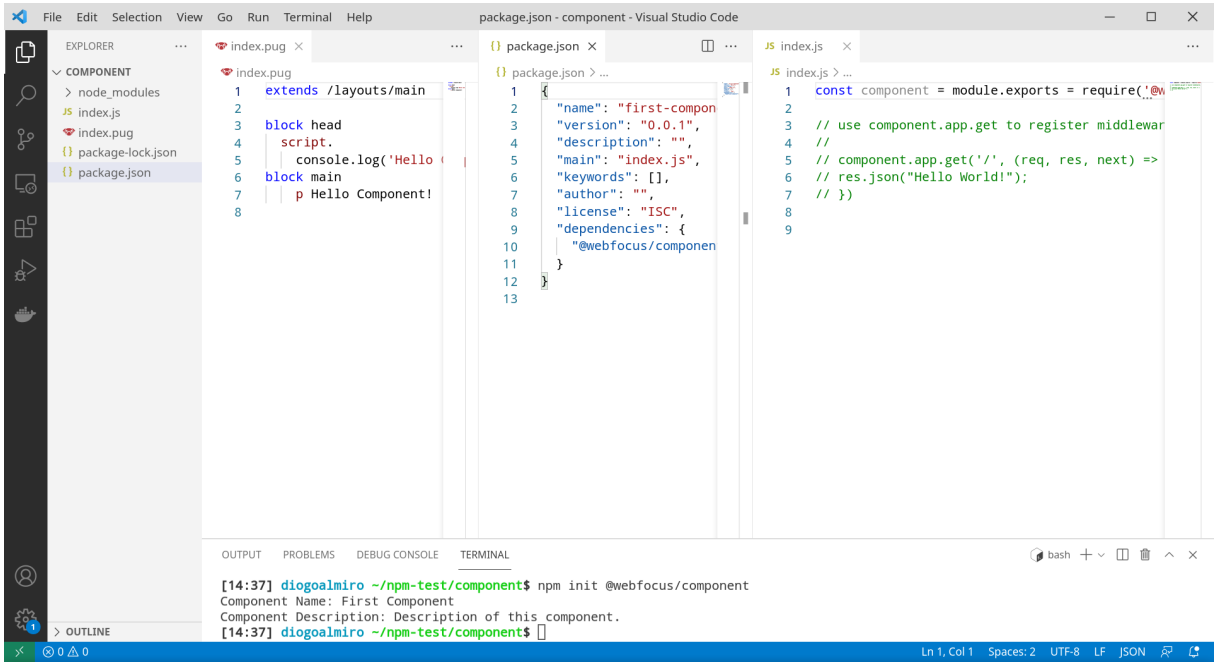

Figure 6.4: Component initialiser example.

6.2.4 Creating an App Instance

Creating a new application is as simple as creating a new package and the file `index.js` with the contents in 6.11. It can also be used with the initialiser `$ npm init @webfocus/app`.

A component can also be created locally within the application package on a new folder. In that case, when registering the component, it should use a relative path to the component, see 6.5.

Listing 6.11: Create component, index.js file.

```
let WebfocusApp = require('@webfocus/app');

let configuration = {
    port : 0, // Specify your port here
    name : "My First Application",
    // Add more configurations here
}

let webfocusApp = new WebfocusApp( configuration );

// Register webfocus/app comonents here
// e.g. webfocusApp.registerComponent(require('../component-examples'));
webfocusApp.registerComponent(require('@webfocus/util/component')); // installed by the initialiser

webfocusApp.start();
```

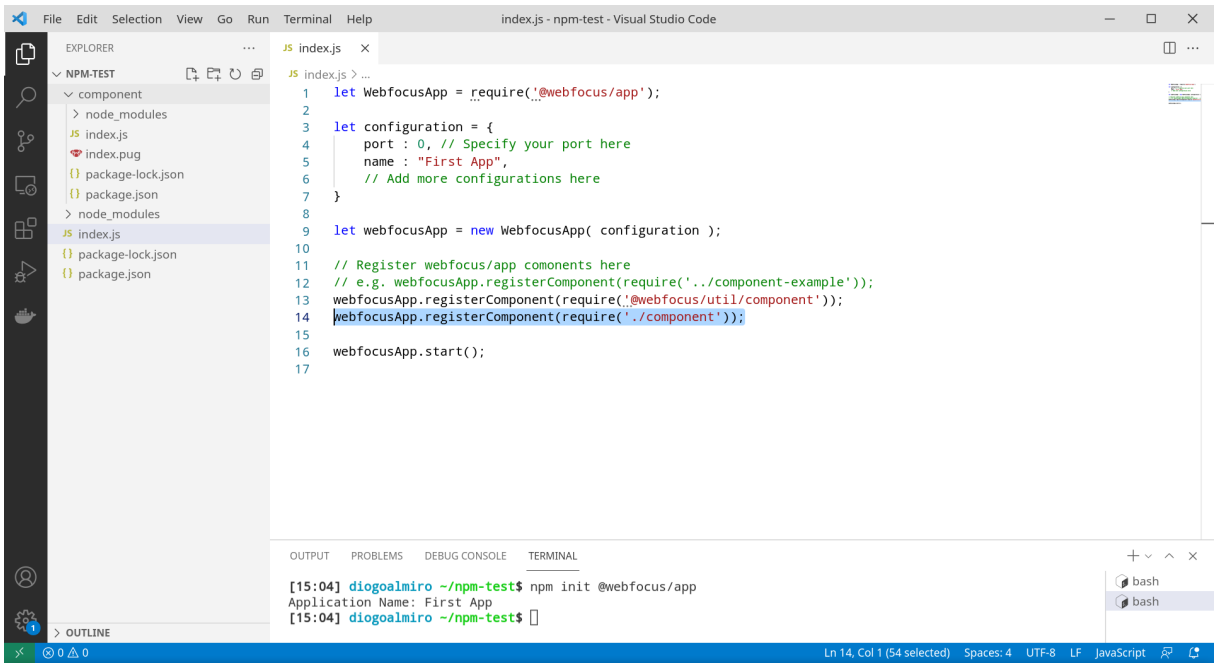

Figure 6.5: Local component example.

6.2.5 Framework Defined Components

Next follows a short description of the three main reusable components implemented.

Util: Some client-side and server-side generic utilities. It sets a property `hidden` to true soo that the end-user does not see it. It provides the static files `submit-json.js`, `pagination.js` and `inline-fecth.js` to the browser, all of them depending on the previously mentioned `/wefocus-static/js/fetch.js` file. It also provides on the server-side the functions `serversideevents` and `pagination`.

Send Mail: It allows configuring an email server in order to be able to send emails to the end-user. It uses the `nodemailer` package and defines the function on the app instance `sendMail` so that other components can also send emails if defined. Its `index.pug` is a submittable form to update the server configuration locally.

Tray: It allows the application to have an icon and menu on the system tray of the end-user. It uses the `ctray` package that this project implemented outside the framework's scope as a nice-to-have feature. Its `index.pug` is a repetition of the actions on the system tray in case it is not available.

6.2.6 Test-Driven Development

For the two main packages, some behaviours are ensured with a TDD approach. It was created the `test` directory on each package and described the behaviour expected with `mocha`, see 6.12. For simulating the HTTP requests, it uses `supertest` (lines 8 to 16).

A stub function was created to test the `registerComponent` of the `WebfocusApp` class. This function generates the `WebfocusComponent` without depending on it. The `WebfocusComponent` does not depend on `WebfocusApp` as well. Afterwards, a context is described with what should happen within it, and it is simulated. In figure 6.6 see the simulation test running for `WebfocusApp`.

After the initial tests, the main objective of mocha is to guarantee that updates to the class do not have breaking changes that are not noticed.

Listing 6.12: Example of a mocha test.

```
1  describe("WebfocusApp#start", function(){
2    it("should send html on the root", function(done){
3      let configuration = { port: 0, name: "Test app" };
4      let webfocusApp = new WebfocusApp(configuration);
5
6      let server = webfocusApp.start();
7
8      request(`http://localhost:${server.address().port}`)
9        .get("/")
10       .expect('Content-Type', /html/)
```

```
11          .expect(200)
12          .then(function(res){
13            assert(res.text.match(configuration.name));
14            server.close();
15            done();
16          })
17      })
18  })
```

```
[12:18] diogoalmiro /run/media/diogoalmiro/LENOVO/diogoalmiro/tese-framework/webfocus/app (main *)$ npm test

> @webfocus/app@0.2.16 test
> mocha


  WebfocusApp
    #constructor
      ✓ should return an webfocusApp instance on port 0
      ✓ should change the port property to 0
  webfocus:app:warning Invalid port in configuration, a random port will be provided on start +0ms
      ✓ should change the port property to 0
  webfocus:app:warning Invalid port in configuration, a random port will be provided on start +5ms
      ✓ should expect the port property to be integer
      ✓ should keep the port property
      ✓ should keep the name property
    #configuration
      ✓ should contain an empty array of components
      ✓ should be a superset of the recieved configuration
      ✓ should save the component name
    #... (other properties)
      ✓ should save a components dictionary
      ✓ should define an express application
      ✓ should define an express middleware application
      started status
        ✓ should not start
        ✓ should start
    #pubObj
      ✓ should contain the configuration
      ✓ should contain extra properties
    #start
      ✓ should send html on the root (493ms)
      ✓ should send json on the api
      ✓ should send fetch.js file
    #... (methods)
      registerComponent
        without component
          ✓ should throw an error
          ✓ should throw an error
          ✓ should throw an error
        with component
          ✓ should register component
  webfocus:app:warning Ignoring component with the same urlname as a previous component. (component: test-component) +600ms
          ✓ ignore repeated components
        after application started
  webfocus:app:warning Ignoring component after start application started. +3ms
          ✓ should ignore component
      getComponent
  webfocus:app:warning Component "undefined" not fount. +1ms
        ✓ should be null
  webfocus:app:warning Component "" not fount. +1ms
        ✓ should be null
        ✓ should return a component
      getAllComponentNames
        ✓ should return an empty array
        ✓ should contain the registered components' urlname


  30 passing (639ms)
```

Figure 6.6: Mocha simulation run for `WebfocusApp`

6.2.7 End-to-end Tests

To test the integration between WebfocusApp instances and `WebfocusComponent` E2E tests can be run on a sample application that uses some components. To run the tests, use the cypress tool[12]. Similar to `mocha` tests, it describes a context and what should happen within it. However, the simulation is through HTTP requests to an application running locally, see line 4 of 6.13. Figure 6.7 shows all the tests done on the sample application.

Listing 6.13: Cypress test description.

```
1  // baseUrl: "http://localhost:9999/"
2  context('Test WebfocusApp Server',() => {
3      describe('Root', () => {
4          beforeEach(()=>{
5              cy.visit('/') // Visits baseUrl + "/" == http://localhost:9999/
6          })
7          it('Displays application name', () => {
8              cy.contains('Cypress Test Application')
9          })
10         it('Lists Components', () => {
11             cy.contains('Cypress Component Test')
12             cy.contains('Static Tests')
13         })
14     })
15 })
```

These tests caught that the implementation was not working as expected. During the creation of the E2E tests, the previous table, 6.1, was used as a reference to describe what was the expected behaviour. When a failure was found, the tests were rechecked. However, there was missing some conditions on the implementation side[13]. When updating, the E2E tests also ensure that old behaviours are not broken unnoticeably.

[12] See www.cypress.io/
[13] See github.com/webfocus-js/app/commit/e79a762884079dfb601913c9d99503ea2df5a817

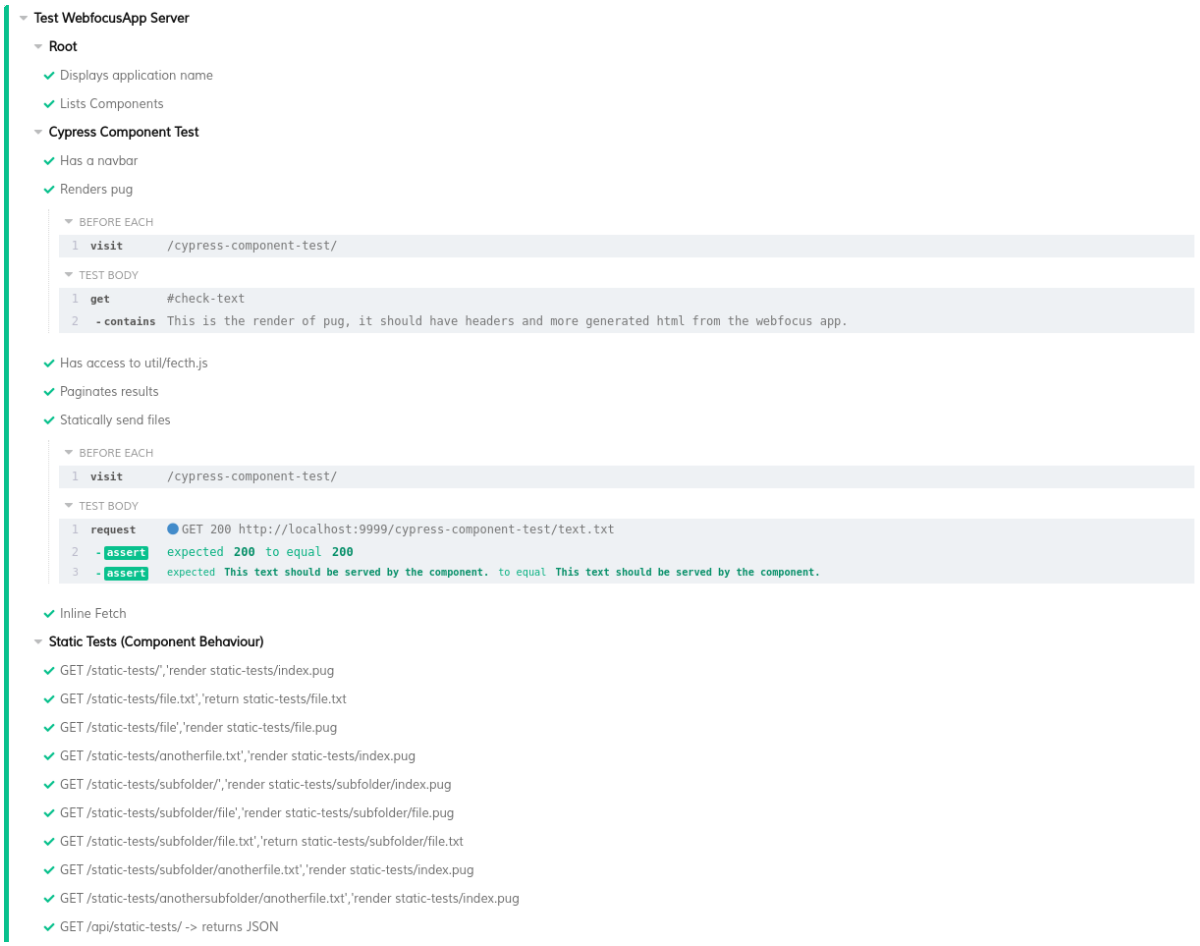

Figure 6.7: Cypress E2E tests for a sample application.

6.3 CTray

This package creates a `CTray` class to control the system tray. It is implemented using Node-API[14] to implement native addons, which means it is written in C++ and needs to be compiled before using it. For Linux, it depends on `appindicator` and `gtk` to work. On Windows, it does not have any extra dependencies. To use it, require the `CTray` class and specify the icon path and the menu.

The `@wefocus/tray` wraps the `CTray` class. Internally it creates an instance of `CTray` exposing functions to control it. The function `addAction` adds an item to the tray and registers the action for access on the web version, see figure 6.8.

Although the essential features to handle the system tray in Linux and Windows were achieved, there are still improvements for this package, such as handling the click callbacks for the tray menu or closing the tray gracefully to be made.

[14] See nodejs.org/dist/latest/docs/api/n-api.html

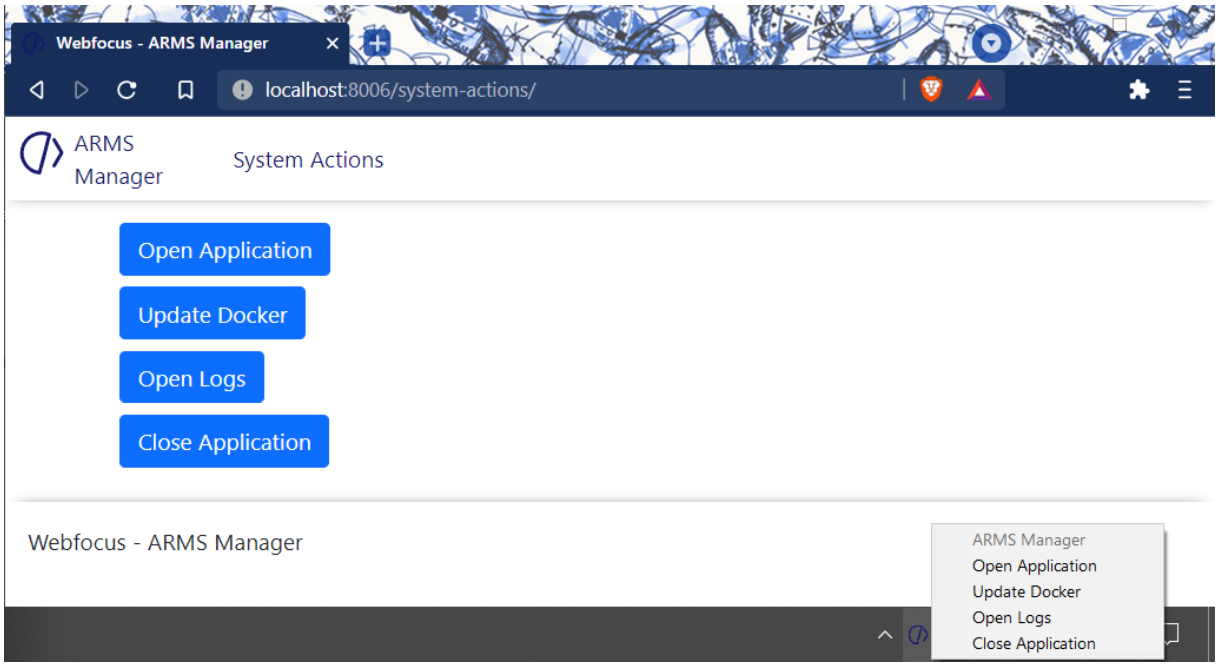

Figure 6.8: Webview and System Tray of the package.

6.4 Conclusions

A core solution was implemented: a framework to help develop hybrid applications. Furthermore, it validates the main components of the framework with automatic test scenarios.

Chapter 7

Demonstration

This section describes several real-life implementations of the framework. It demonstrates an implementation from the early stages of the framework, featuring some of its utilities and an embedded database. Afterwards, it shows implementations related to the two case studies. Finally, it compares the implementations with the expected use cases from section ”3. Requirements”.

7.1 Early Stage Showcase Application

This application showcases some utilities from the `util` component as well as a database connection. It contains one component with an HTML form asking for a Portuguese cellphone number. It then checks if that phone number was leaked by Facebook[1].

From `util`, the component uses the `sumbit-json.js`. This file defines an event that catches form submits with a specific data attribute[2]. It transforms the regular HTML form submission, which reloads the page, into an AJAX request. Once the request is fulfilled, the function associated with the attribute is called.

In this case, the function pops up an alert box with the result from the search, see 7.1.

The code for the component has 13 lines and only defines one handler for a POST request. See 7.1 line 11. The constructor is called with the location to open (line 5) the embedded database, using the package `better-sqlite3`, and the SQL statements are prepared (line 7).

Listing 7.1: All backend component's code of the showcase application.

```
1 let component = module.exports= require("@webfocus/component")("Check", "
    Check a cellphone number.")
2
3 let Database = require("better-sqlite3");
4
```

[1] See www.theguardian.com/technology/2021/apr/06/facebook-breach-data-leak
[2] See developer.mozilla.org/en-US/docs/Learn/HTML/Howto/Use_data_attributes

```
5  let db = new Database("phone-hashes-small.db");
6
7  let get = db.prepare("SELECT phonehash FROM phonehashes WHERE phonehash = ?
       ");
8
9  let sha256 = phone => require('crypto').createHash("sha256").update(phone).
       digest('base64');
10
11 component.app.post("/", (req, res, next) => {
12     let number = req.body.number;
13     if( !number ) return next(new Error("number not provided"));
14     if( number.indexOf("351") != 0 && number.length == 9){
15         number = `351${number}`;
16     }
17     res.json(get.all(sha256(number)))
18 })
```

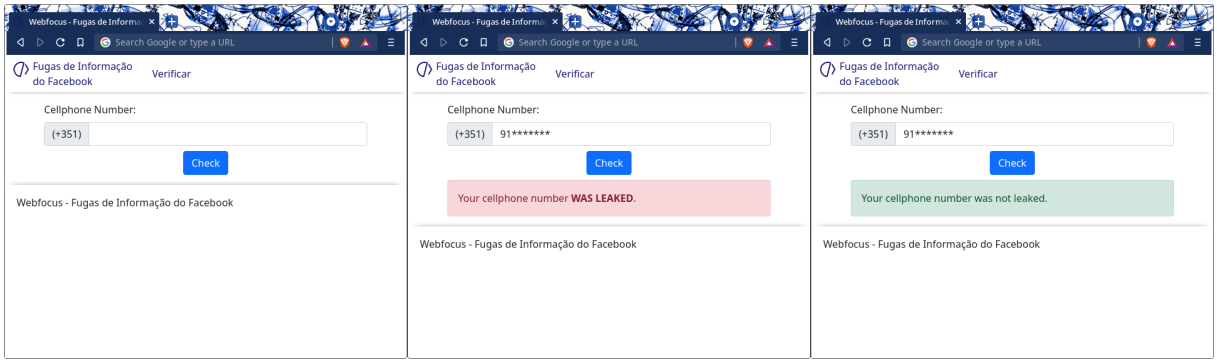

Figure 7.1: Front-end Showcase Application

7.2 IRIS Demonstration

For the case study IRIS, tools to help navigate several documents and see the metadata associated were created and a proof of concept that integrates Microsoft Word and automatic reloads when it changes.

7.2.1 Data Visualiser Application

This application showcases three relatively similar components to view files related to IRIS. The first component is the DGSI Explorer[3]. This component displays a table showing if "Summary" and a "Full Text Decision" fields are available for each DGSI process. It uses the `pagination` plugin from `util` to request only a subset of the whole database. It can also open the process and see in more detail the metadata of the process and compare the fields mentioned above, see figure 7.2.

Figure 7.2: DGSI Explorer component pages.

The second component was intended to help manually label the processes related to IRIS that the project received. It also uses `pagination` to show a table with the PDF and the number of characters. The process' labelling never got implemented, and it only demonstrates how to show a PDF file and metadata side by side, see figure 7.3.

[3]Beforehand, the whole DGSI public database was collected, `http://www.dgsi.pt/jstj.nsf/` to local files.

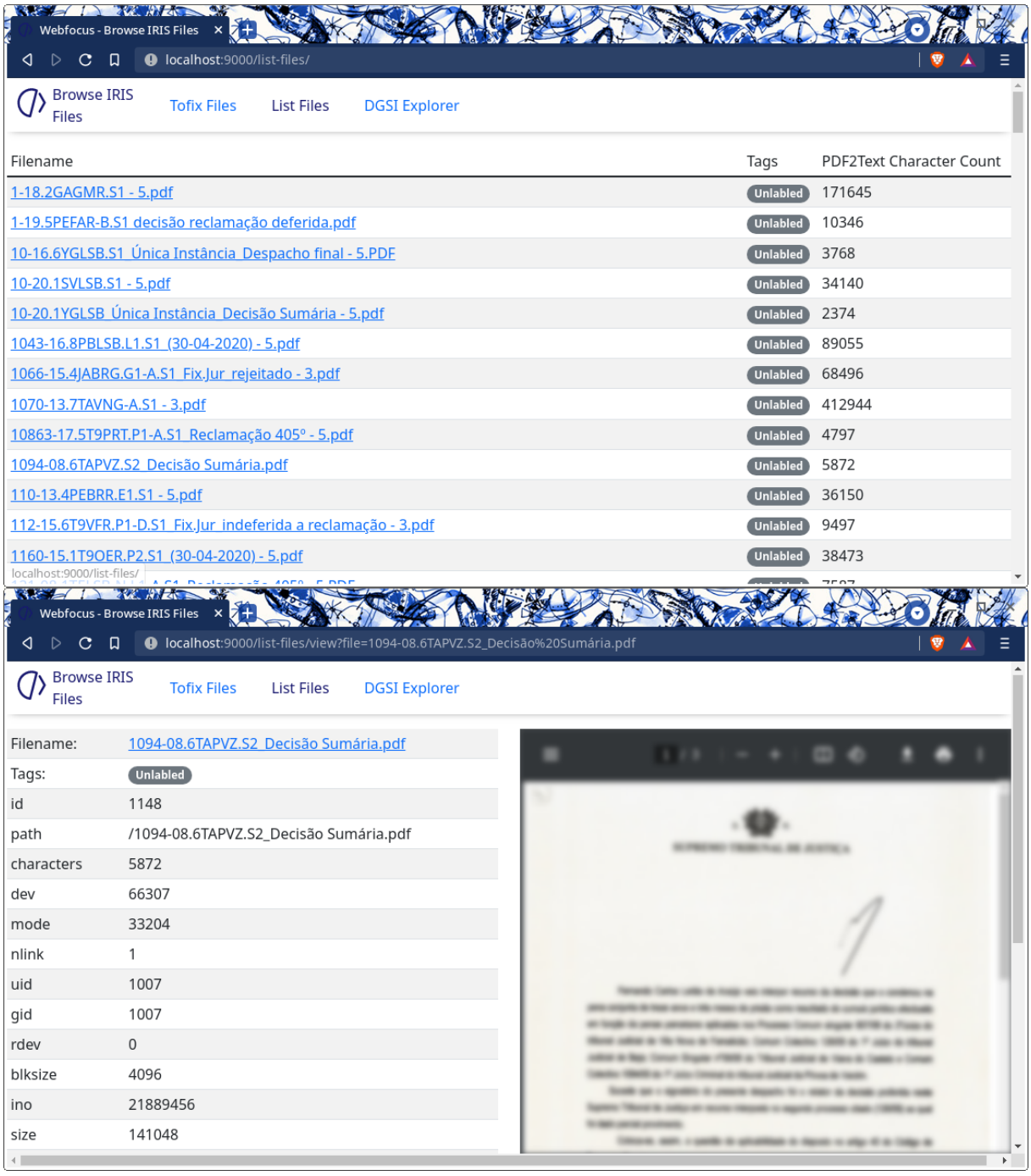

Figure 7.3: Labeling component pages.

The third component shows the conversion from PDF to test the `pdftotext` command-line tool and the extra characters `<line-break>` representing `\n` and `<line-feed>` representing `\f`, see figure 7.4. It also uses `pagination` to show a list of textfiles that were processed using the tool.

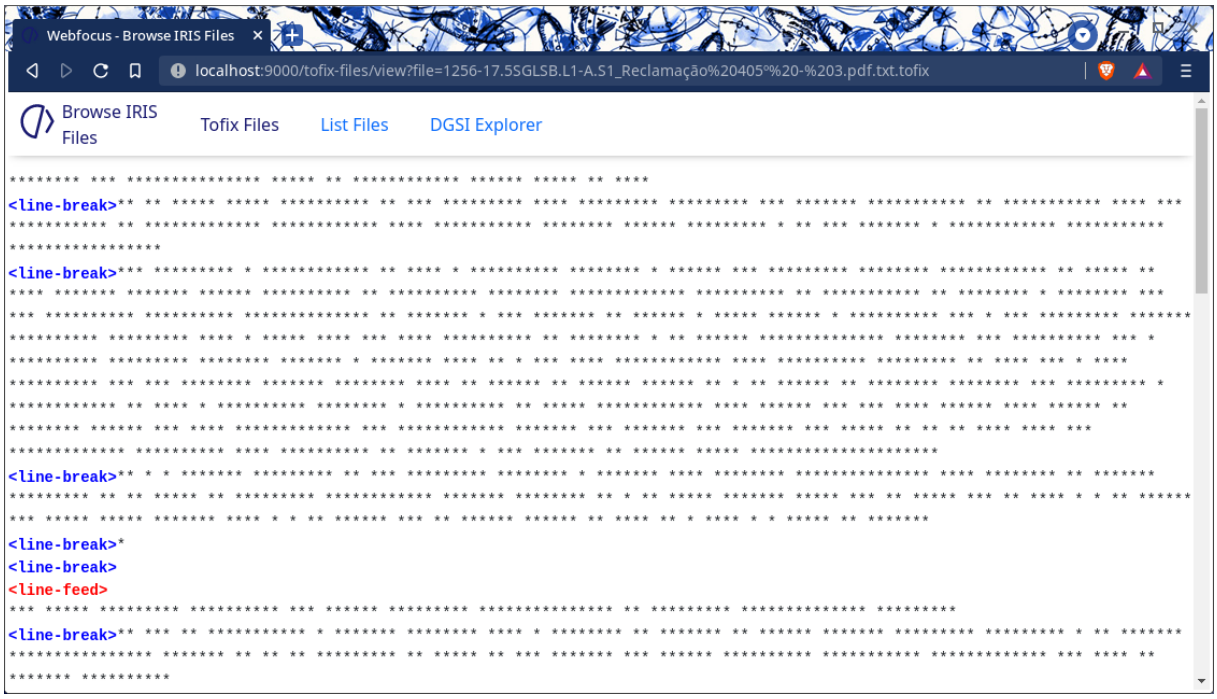

Figure 7.4: `pdf2text` tool on a process. Note that the text was censured to hide sensitive information.

This implementation inspired a more profound report over the DGSI database, as abnormal fields were found. From the report, 24 out of 36319 processes were filtered, and then each was checked manually using the implementation. With the manual check, it was possible to specify that the filtered processes only had signatures, dates, or a small explanation, on the summary field.

7.2.2 Microsoft Word Integration

The second application related to IRIS showcases one possible integration between a `docx` editor and the application. It has a button to open the file default in the editor, usually Microsoft Word. The application watches for changes in the file on the server-side and notifies the client-side when it is saved. It also demonstrates that the application can show the contents of the `docx` on the browser, see figure ref{file-watcher}.

This component dependes on packages `open` to open the editor (line 17 of 7.2), and `decompress` to get the `document.xml` from the `docx` archive (line 5). It also uses the `serversideevents` function from `util` to send events (line 12) and the `EventSource` from the browser to get those events and update the content on the front-end, see 7.3. It emits an event when the OS notifies that the interested file has changed (lines 7 to 10).

Issues

The application only reacts to updates when the user explicitly saves the file. There might be ways to check the autosave versions of the file and its content.

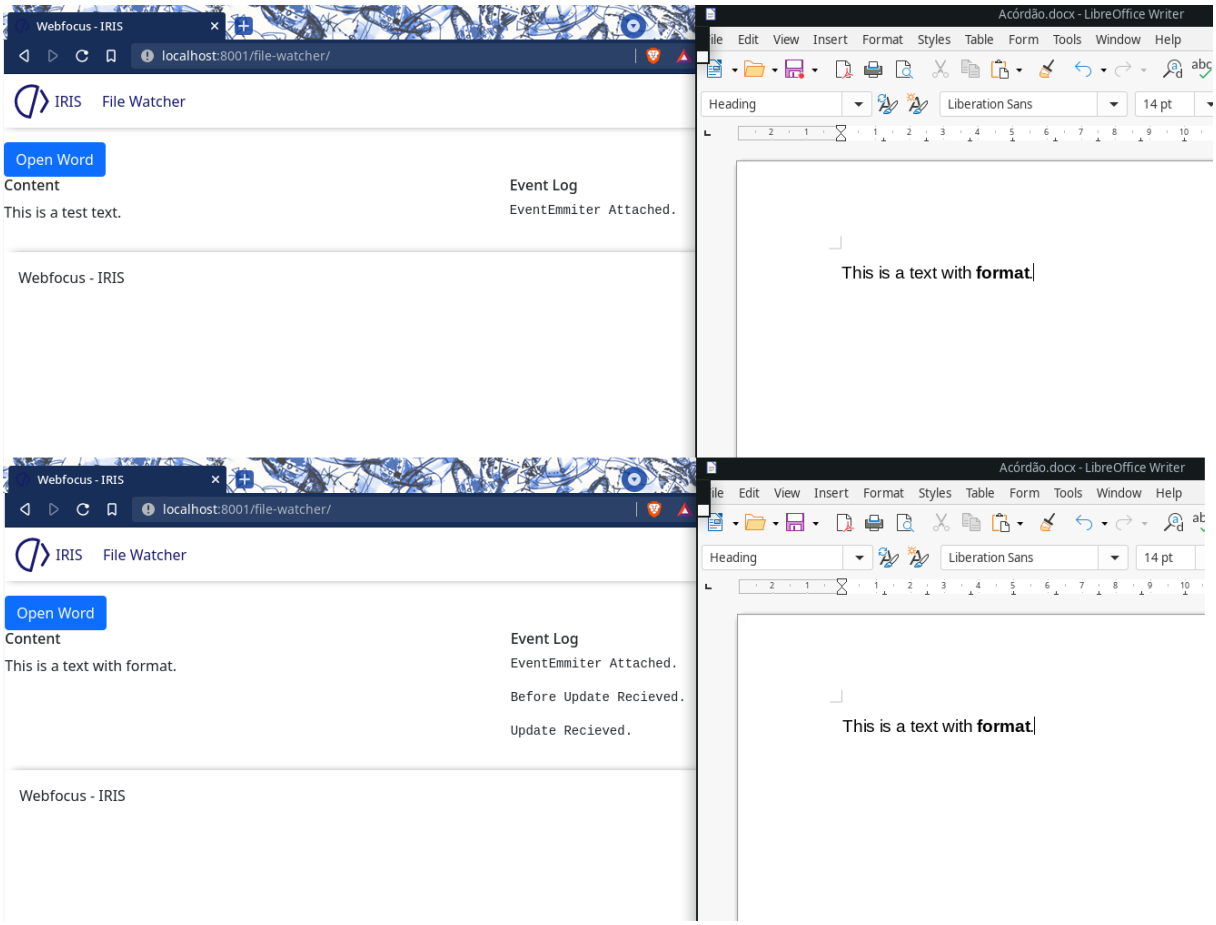

Figure 7.5: File watcher example.

Listing 7.2: File watcher component code.

```
1  const e = new events.EventEmitter()
2
3  const DOCX_FILE = // File to watch
4
5  const decompressFile = () => decompress(DOCX_FILE, ...).then( () => e.emit(
       "update") );
6
7  fs.watchFile(DOCX_FILE, (curr, prev) => {
8      e.emit("before-update");
9      decompressFile()
10 })
11
12 component.app.get("/file-events", serversideevents(e, ["update", "before-
       update", "attached"], () => {
```

```
13      e.emit("attached", "");
14  }));
15
16  component.app.get("/open-word", (req, res) => {
17      open(DOCX_FILE);
18      res.json(1);
19  })
```

Listing 7.3: EventSource usage on the browser.

```
1  const es = new EventSource("!{apibaseurl}file-events");
2  let getFile = () => fetch("!{componentbaseurl}filename/word/document.xml")
       // it also parses the xml to text and displays it
3
4  es.addEventListener("attached", (m) => {
5      getFile();
6      let log = document.getElementById("log");
7      let pre = document.createElement("pre");
8      pre.textContent = "EventEmmiter Attached."
9      log.appendChild(pre);
10 })
11 es.addEventListener("update", (m) => {
12     getFile();
13     // ...
14 })
```

7.3 ARMS Demonstration

For the case study ARMS, a proof of concept with several functionalities was created, and later a final solution after the feedback from the end-user. Firstly, this section describes the previous technology created, ARMS Workflow.

7.3.1 ARMS Workflow Technology

As previously mentioned, a previous project created a command-line based technology, `ARMS-Workflow`, to process ARMS documents. Although this technology is treated as a black box for the implementation, follows a short description.

`ARMS-Workflow` uses `python`, which depends on the program `tessereact-ocr`[4] to work. Its input is a file or folder path to process and some flags to specify the language and how it should process each file. Its output is an OCR'ed PDF file containing the original images and the text found by `tesseract-ocr`.

After installing `ARMS-Workflow` on the end user's computer, it could use the command-line to execute it with the syntax in listing 7.4.

Listing 7.4: ARMS-Workflow command line usage.

```
1  $ python3 workflow.py --help
2  Usage: workflow.py [OPTIONS] FILE
3
4  Options:
5    --comp                        Disable line segmentation comparison
6                                  [default: True]
7    --prep                        Additional prep-processing  [default: False
       ]
8    --lang [ara|eng|osd|por|rus]  Available languages.  [default: eng]
9    --force                       Start from the beggining  [default: False]
10   --tmp                         Keep tmp files. WARNING: It requires more
       free
11                                 disk space  [default: False]
12   --folder                      Workflow is performed in all images from
       this
13                                 folder  [default: False]
14   --help                        Show this message and exit.
```

However, this technology is on a stage that could not be executed on the end user's computer due to some problems when installing its dependencies on Windows, precisely installing a `python` library to use the `tesseract-ocr`. While the installation problem is not solved, a workaround is to use Docker [5] and virtualisation to containerise the Linux technology to run on Windows.

Docker

With Docker, any technology imaginable can be run in a controlled environment. It "creates a simple tooling and a universal packaging approach that bundles all application dependencies inside a container which is then run on Docker Engine."[6]

[4] See tesseract-ocr.github.io/
[5] See www.docker.com/
[6] See www.docker.com/products/container-runtime

Within Docker, there is a world of concepts to explore. Follows a description of Dockerfile, Docker Images, and Docker Containers, as the remaining environment is outside the framework's scope.

Dockerfile: Specifies a base environment for a Docker Image. It is a text document with instructions to define or extend a Docker Image. It can be imagined as being the description of a programming class.

Docker Image: It is a package with, for instance, resources, environment and dependencies for running a Docker Container. It would be the class itself.

Docker Container: It is a process running within the Docker Engine given the environment of a Docker Image. It would be an instance of the class[18].

Listing 7.5: Dockerfile of ARMS-Workflow

```
1  # ARMS-Workflow extends from the ubuntu image.
2  FROM ubuntu:18.04
3
4  # ARMS-Workflow requires python3, pip3 and tesseract, among other
       dependencies.
5  RUN apt-get update && apt-get install -y \
6      tesseract-ocr \
7      python3 \
8      python3-pip \
9      libtesseract-dev \
10     libleptonica-dev \
11     pkg-config \
12     ffmpeg \
13     libsm6 \
14     poppler-utils \
15     libxext6 \
16     && apt-get clean \
17     && apt-get autoremove
18
19 # ARMS-Workflow requires some local files.
20 ADD workflow /home/App
21 WORKDIR /home/App/exec
22
23 # ARMS-Workflow requires other dependencies from pip3.
24 RUN apt-get install libffi-dev -y # installing pip dependencies
25 RUN python3 -m pip install --upgrade pip
26 RUN pip3 install -r ../requirements.txt
```

```
27
28 # ARMS-Workflow environment variables.
29 ENV LC_ALL=C.UTF-8 # Setting environment variables
30 ENV LANG=C.UTF-8
31
32 ENV PYTHONUNBUFFERED=1
```

7.3.2 Proof of Concept

The initial proof of concept for the case study ARMS runs inside a docker container alongside the technology ARMS Workflow.

This application defines some custom properties on the configuration object to be used later on each component. Specifically, it defines the file locations for stages of the workflow. It also initialises a table on an embedded database.

Listing 7.6: ARMS Proof of concept application setup.

```
1  let configuration = {
2      port : parseInt(process.env.PORT), // get port from environment
       variable
3      name : "ARMS",
4      uploadFolder: path.join(__dirname, "persist", "tif-uploads"), //
       Specify file locations
5      resultFolder: path.join(__dirname, "persist", "ocr-results"),
6      hocrTmpFolder : path.join(__dirname, "persist", "ocr-tmp-files"),
7      statusDatabasePath : path.join(__dirname, "persist", "workflow-status.
       db")
8  }
9
10 fs.mkdirSync(configuration.uploadFolder, { recursive: true }) // Ensure
       locations exist
11 fs.mkdirSync(configuration.resultFolder, { recursive: true })
12 fs.mkdirSync(configuration.hocrTmpFolder, { recursive: true })
13
14 // Ensure table exists
15 new Database(configuration.statusDatabasePath).exec(`CREATE TABLE IF NOT
       EXISTS workflow (...)`)
16
17 let webfocusApp = new WebfocusApp( configuration );
```

Its first component lets the user submit a “tif” file to OCR by dragging and dropping it to the browser. After the submission is successful, it updates the database. It uses the pagination from `util` to show the already inserted files and allows the user to delete them.

The second component displays the current stage of each “tif” file for the OCR workflow. It also uses pagination while accessing the database to get the stage, automatically reloading if the stage changes.

The third component gives access to the result files generated. It also uses pagination, and it redefines the `staticApp` to send files from another folder.

Finally, a hidden component spawns a worker that checks the database from time to time for the next process to OCR. It emits server-side events while OCR’ing that the second component listens to and reloads the page if needed. The worker is a NodeJS script running a loop on a different thread. Once a new file appears, it executes the “ARMS Workflow” technology, generating a PDF file with the OCR result. It also generates a zip file containing the TIFF pages converted to PNG and the HOCR that can be opened on the browser. See figure for an overview of the components front-end 7.6 and figure 7.7 for an example of the archive file.

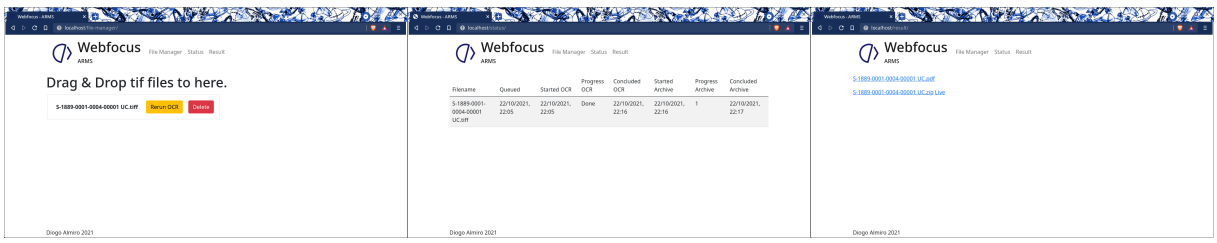

Figure 7.6: ARMS Proof of concept - main components.

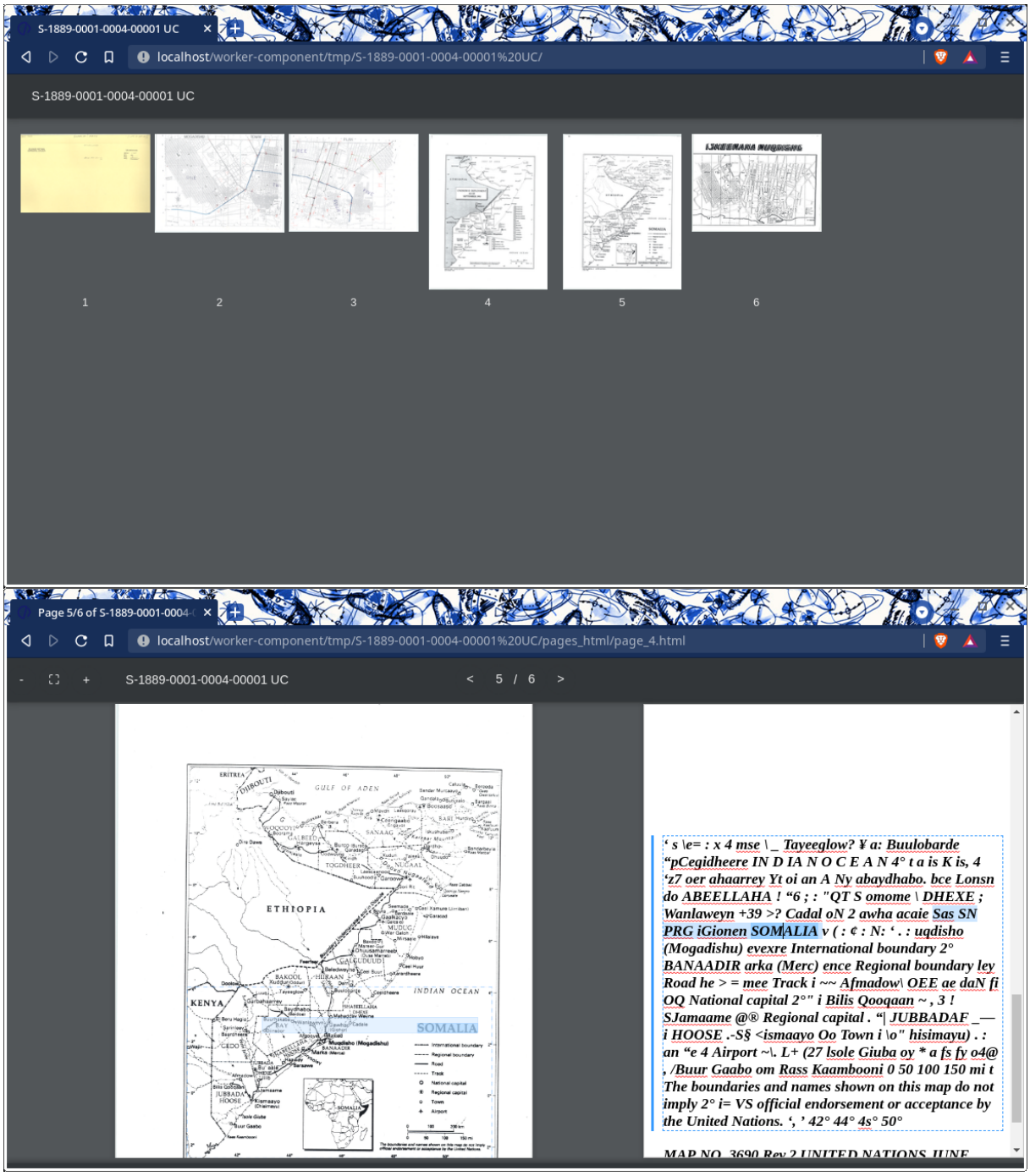

Figure 7.7: ARMS Archiving OCR result.

Issues

There is a significant problem with this proof of concept. It requires copying the files from the hard drive to the computer (by inserting them in the application) and downloading the results manually to the hard drive again. In order to solve this dealbreaker issue, the application cannot be run inside Docker. Because Docker does not give enough permissions to read/write from the hard drive dynamically. In other words, it is possible to continue running inside Docker, however, before each utilisation. The user

would need to create a new container and specify the location of the files manually, a non-solution for the end-user.

Some feedback was received from the end-user with some problems:

- The application only allows “drag and drop” and should also select files from the browser’s dialogue.
- Once in the system, the OCR would start without allowing to change definitions or the order for running the files.
- There was no “batch” concept to process multiple files at once with the same configurations.
- The status component could show more information.

7.3.3 Second Solution

Another application was implemented for ARMS to solve the issue mentioned above. This time, the application interacts directly with Docker Engine to spawn an OCR process on a folder in a new container. It only defines a component to interact Docker within this application and use the reusable component: `tray`, `util` and `send-mail`.

The application is packaged to an executable file using the `pkg` package to be more user friendly, see figure 7.8. A “packaged” application is an executable file `.exe` containing the NodeJS environment, the scripts and the assets required to run. To generate the executable, a script only for Windows was created, on the “package.json”: `npx pkg %npm_package_json% -t win -o %npm_package_name%-v%npm_package_version%.exe`. To run it use `npm run package` this command generates a “arms-manager-v_._._.exe” file. Now to start the server, the user needs to click this file.

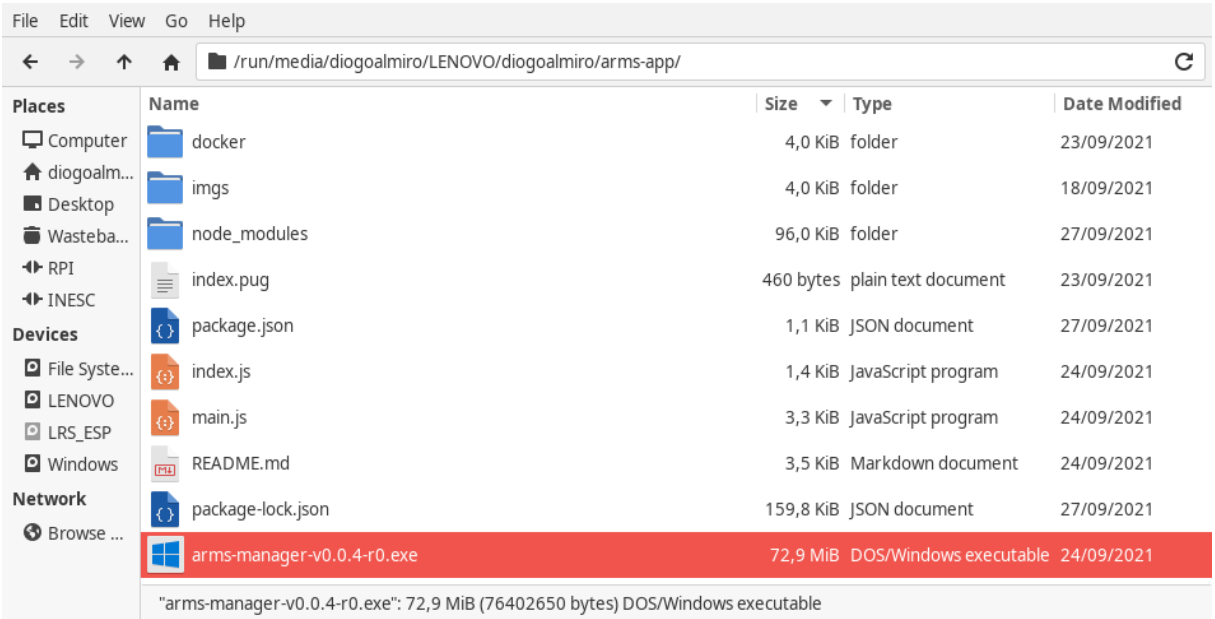

Figure 7.8: Execuable file generated by `pkg`.

When registering the `send-mail` component, it verifies if the user already has a configuration to use and if not, it defaults to a provided UN server. It also defines some actions to the `tray` component with system actions, including closing the application and opening the logs folder that the end-user can easily send in case of errors.

In the specific component of this application, the docker image is built, and the containers for it are created. It also watches docker events to reload the page automatically or send an email reporting the termination of an OCR process.

To create an OCR process, the user selects the path using the browser. Later he can define the configuration for the OCR process, including which dictionaries to use or the priority. See figure 7.9 for the pages the end-user sees. When the configuration is ready, the user requests the process to start. Internally the docker container is created and starts once other docker containers have terminated. This container is created with reading/writing access to the user-specified folder, allowing the container to output the pdf there. It uses a modified version of the original “ARMS Workflow”:

- That writes the output file on the same folder as the input file.
- That appends the suffix “-ocr” to the output name of the PDF. (Example, the final name of “input-name.pdf” or “inputname.tif” would be “inputname-ocr.pdf”, see figure 7.10)
- That, before starting execution for a file, checks if the output name exists only running if not.
- That downloads the dictionaries needed on each container to reduce the space used by the docker image.
- That does not depend on NumPy as it was taking longer to build the image as it was not used[7].

Once the container stops, it tries to send an email informing the end user about it, see figure 7.11.

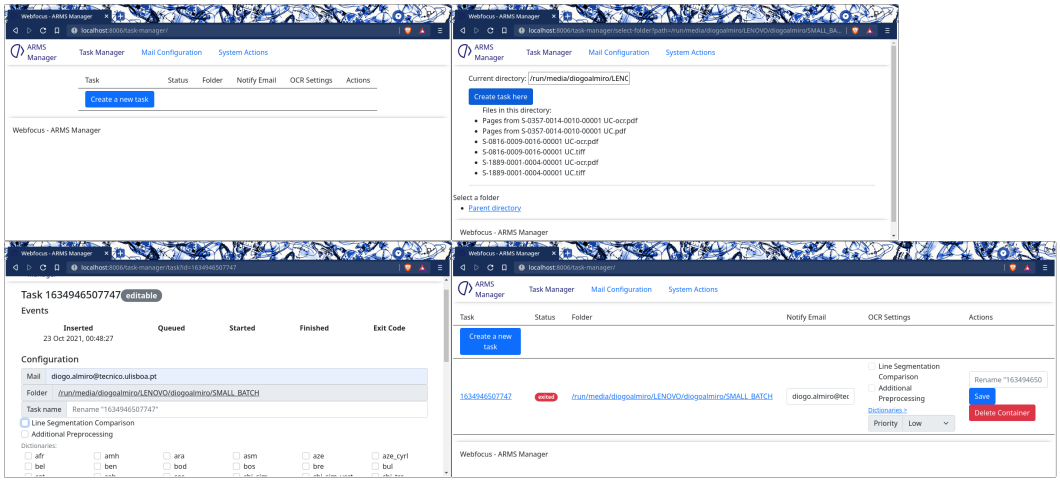

Figure 7.9: Front-end pages for the second ARMS solution.

[7]Another dependency had the same feature.

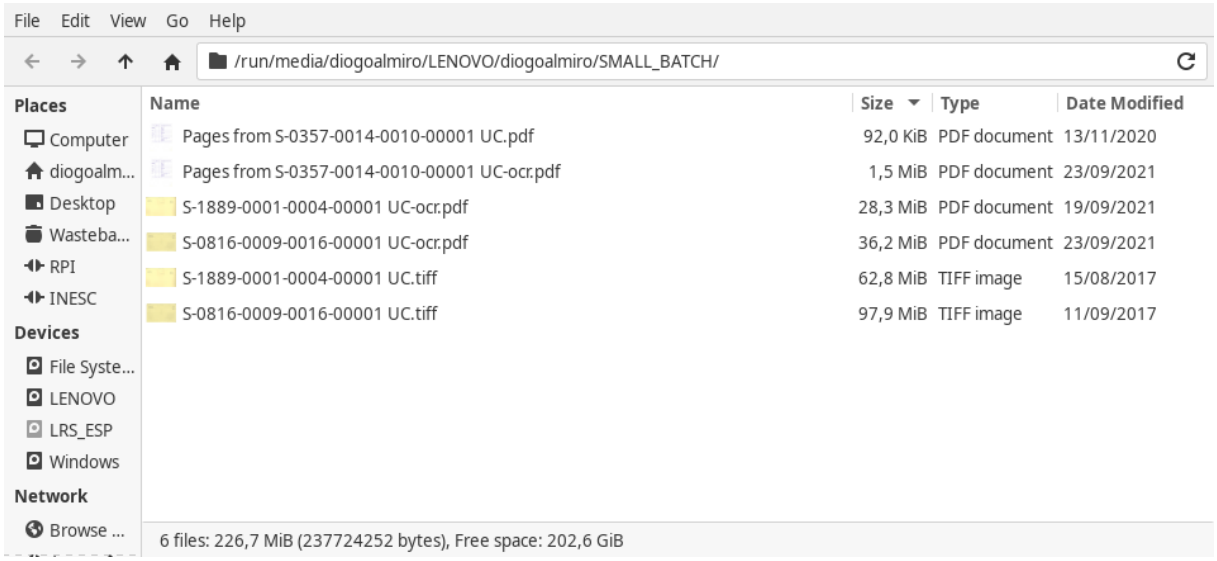

Figure 7.10: Folder input folder after the OCR process

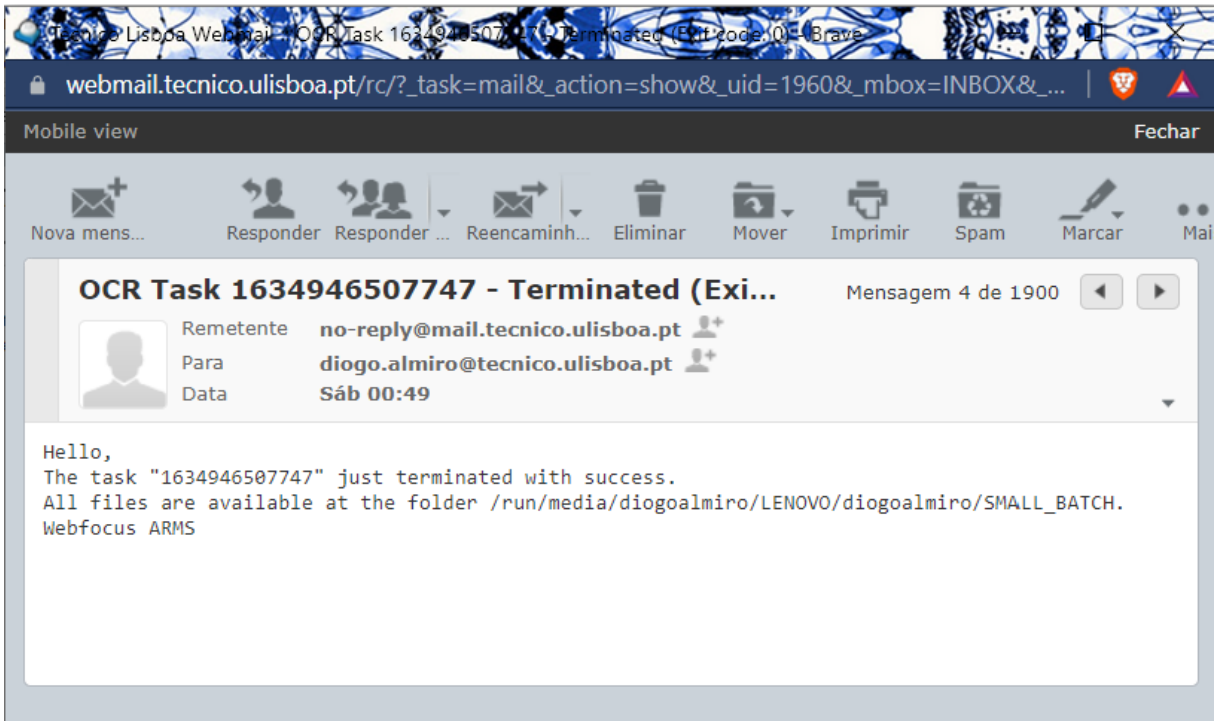

Figure 7.11: Notification email.

Issues

This solution solved the main issue from the first proof of concept. During the first releases, the ARMS end-user was able to find problems that were fixed on later results. For instance, it was not dealing correctly with names with spaces when renaming a container. Before changing the ARMS

workflow, it could not run it with the same folder for the output and input.

A significant error still showed up from version 1, where it could not create the container. At that time, it was assumed white spaces in the folder caused the error. However, in later versions, this error came back - “It seems like it is a roll of the dice at times and I have to click on”run” multiple times.”. This error is not constant, and it was not reproducible on the developer’s machine.

A feature on the framework is still missing to enable the creation of executable files with ease. The `pkg` can find most resources required, but it can not find native bindings, such as the `ctray` or `open` packages. It is also not able to know that most files under a component are assets that should be added to the executable. Therefore the application `package.json` must be more complicated to inform `pkg` about those files for every component registered, see 7.7. Albeit, the primary influence of the final executable size is from the NodeJS environment. The scripts and assets might also influence it is not dealt with prudence and add all scripts to the executable.

Listing 7.7: Part of ”package.json” related to ”pkg” of the second ARMS solution.

```
"pkg": {
    "assets": [
      "./index.pug",
      "./docker/*",
      "./node_modules/@webfocus/app/views/**/*",
      "./node_modules/@webfocus/app/static/**/*",
      "./node_modules/@webfocus/util/**/*",
      "./node_modules/@webfocus/send-mail/**/*",
      "./node_modules/@webfocus/tray/**/*",
      "node_modules/ctray/build/**/*",
      "node_modules/drivelist/build/**/*"
    ],
    "scripts": [
      "node_modules/ctray/tray.js"
    ]
  },
```

7.4 Conclusions

Several concepts were implemented, showing the usefulness of the framework and successfully demonstrating that the use of external technologies (e.g. Docker) is possible. There are still features the framework is missing, such as generating an executable.

Chapter 8

Conclusions

The challenge of this project was to propose a reference architecture for the use of web technologies in the desktop domain. Precisely, it ought to improve workflows in an administrative environment of two case studies: digitising historical documents on ARMS and analysing legal documents on a top national court.

After introducing some basic concepts, aimed solution and the application characteristics were described according to the case studies. Afterwards, technologies that could help solve the problem were analysed, reaching a reasoned ground to formulate a solution and a proposed architecture in more detail. Architecture that was implemented afterwards alongside several use cases to demonstrate it.

8.1 Achievements

After analysing the current state of the art, a well-defined framework was achieved that extended the Express framework to create desktop applications that use the browser as the interface:

- For the developer:
  - It is simple to define an application reusing components or using local components. Requiring a few lines, and most of them could be copy-pasted into the application.
  - It is more challenging to create the component itself, as it requires more know-how of Express and the framework itself and ways to proxy to external technologies using NodeJS.
- For the administrator:
  - It is simple to install, requiring only the download of the executable.
- For the IRIS end-user:
  - There are no examples of applications for this scenario yet.
- For the ARMS end-user:
  - The application is usable but can still improve.

Regarding the use cases in section ”3. Requirements”, table 8.1 summarises the characteristics implemented and partially implemented.

Table 8.1: List of the use cases.
■- Implemented, ☒- Partially Implemented

□	Case Study	Use Case
	ARMS	
■		Select a TIFF files' location.
■		Define OCR settings.
■		Execute a bulk OCR on a location.
■		Notify the user.
☒		See TIFF files.
□		See OCR metadata.
☒		Edit OCR results.
□		Regenerate a PDF from manually edited OCR.
	IRIS	
□		Select a PDF document to use.
□		See the document and annotations.
□		Exectue OCR on demand.
□		Annotate the document.
□		Correct the OCR.
☒		Synchronise an external editor to the application.
□		Export an archived document.
□		Import an archived document.

8.2 Future Work

Regarding the framework itself, there are still several improvements it can receive:

- One idea would be to create an extra package “@webfocus/exe” to create executable files.
- The system tray package could still improve the menu handling and closing of the tray.

Regarding applications, some features implemented on the proof of concept for the case study ARMS were not introduced on the newer versions, such as the zip file containing the HOCR. However, the main problem is the current use of docker, which has a problem in the end user's machine. Ideally, it would be able to run the external technology without docker. Another idea regarding ARMS would be to enable an internal system that could distribute the file with other nodes inside the network to accelerate the OCR process. For the case study IRIS, there is still an ongoing development that might use this framework to

create the final application.

Bibliography

[1] P. Behrami. Implementation of a web application. Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2019. URL https://is.muni.cz/th/hy3q7/.

[2] R. Appel. Mobile web sites vs. native apps vs. hybrid apps. *MSDN Magazine*, 2014. URL https://docs.microsoft.com/en-us/archive/msdn-magazine/2014/november/modern-apps-mobile-web-sites-vs-native-apps-vs-hybrid-apps.

[3] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. WebApps'11, page 7, USA, 2011. USENIX Association.

[4] G. Kappel, B. Pröll, S. Reich, and W. Retschitzegger. *Web Engineering: The Discipline of Systematic Development of Web Applications*. Wiley, 2006. ISBN 9780470028933. URL https://books.google.pt/books?id=DZySwN2S1JIC.

[5] C. Sven, D. Florian, D. Peter, and M. Maristella. *Engineering Web Applications.* Data-centric Systems and Applications. Springer, 2009. ISBN 9783540922551. doi: 10.1007/978-3-540-92201-8.

[6] K. Liu, J. Jiang, X. Ding, and H. Sun. Design and development of management information system for research project process based on front-end and back-end separation. In *2017 International Conference on Computing Intelligence and Information System (CIIS)*, pages 338–342, 2017. doi: 10.1109/CIIS.2017.55.

[7] L. Teixeira, A. R. Xambre, H. Alvelos, N. Filipe, and A. L. Ramos. Selecting an open-source framework: A practical case based on software development for sensory analysis. *Procedia Computer Science*, 64:1057 – 1064, 2015. ISSN 1877-0509. doi: 10.1016/j.procs.2015.08.525.

[8] M. M. Moe and K. K. Oo. Evaluation of quality, productivity, and defect by applying test-driven development to perform unit tests. In *2020 IEEE 9th Global Conference on Consumer Electronics (GCCE)*, pages 435–436, 2020. doi: 10.1109/GCCE50665.2020.9291950.

[9] T. Yogesh and P. Vimala. Test-driven development of automotive software functionality. In *2020 Third International Conference on Smart Systems and Inventive Technology (ICSSIT)*, pages 1162–1165, 2020. doi: 10.1109/ICSSIT48917.2020.9214078.

[10] C. Augusto. Efficient test execution in end to end testing : Resource optimization in end to end testing through a smart resource characterization and orchestration. In *2020 IEEE/ACM 42nd*

International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pages 152–154, 2020.

[11] R. Paul. End-to-end integration testing. In *Proceedings Second Asia-Pacific Conference on Quality Software*, pages 211–220, 2001. doi: 10.1109/APAQS.2001.990022.

[12] P. Seebach. Develop web applications for local use. *IBM, developerWorks*, 2007, accessed October 2020. URL `https://www.ibm.com/developerworks/library/wa-localwebsrv/index.html`.

[13] I. G. Danil Demashov. Efficiency evaluation of node.js web-server frameworks. *ITMO University, Saint Petersburg, Russia*, 2019. URL `http://ceur-ws.org/Vol-2590/short14.pdf`.

[14] A. Adamanskiy and A. Denisov. Ejdb - embedded json database engine. In *2013 Fourth World Congress on Software Engineering*, pages 161–164, 2013. doi: 10.1109/WCSE.2013.29.

[15] L. Junyan, X. Shiguo, and L. Yijie. Application research of embedded database sqlite. In *2009 International Forum on Information Technology and Applications*, volume 2, pages 539–543, 2009. doi: 10.1109/IFITA.2009.408.

[16] L. P. Chitra and R. Satapathy. Performance comparison and evaluation of node.js and traditional web server (iis). In *2017 International Conference on Algorithms, Methodology, Models and Applications in Emerging Technologies (ICAMMAET)*, pages 1–4, 2017. doi: 10.1109/ICAMMAET.2017.8186633.

[17] Y. Yan, N. Jiang, H. Wang, Y. Wang, C. Liu, and Y. Wang. Multi-sql: An extensible multi-model data query language, 2020. URL `https://arxiv.org/abs/2011.08724`.

[18] S. Wang, L. Zhu, and M. Cheng. Docker-based web server instructional system. In *2019 IEEE/ACIS 18th International Conference on Computer and Information Science (ICIS)*, pages 285–289, 2019. doi: 10.1109/ICIS46139.2019.8940219.