POLITECNICO DI TORINO

Master's degree in Computer Engineering

Master's Degree Thesis

Extension of an enterprise web application for top-management reporting

A modular approach to Web Application development

Supervisor
dott. Riccardo COPPOLA

Candidate
Andrea TORREDIMARE

Company Tutors - SPRINT REPLY
dott. Emanuele DALL'OSPEDALE
dott. Andrea SEDINI

A.Y. 2023/24

Summary

In an increasingly competitive business landscape, effective procurement and supply chain management are critical to organizational success. As companies adopt data-driven decision-making, the demand for sophisticated reporting tools has grown significantly. This thesis examines the enhancement of SmartSpend, a modular enterprise web application designed to support top-management reporting in these crucial domains. Developed by Sprint Reply (a leader in digital transformation) in the course of the last three years, this project equips executives with the insights necessary to refine strategies and optimize resource allocation.

Built on the robust Spring Boot framework, SmartSpend ensures both scalability and reliability; it integrates with the Freemarker template engine to deliver dynamic content, presenting complex data through interactive dashboards and comprehensive reports. At its core, Microsoft SQL Server serves as a central repository, managing extensive procurement-related data with high performance and consistency. The application is further supported by the Smart Procurement API layer, a secondary Spring Boot integrated with Robotic Process Automation (RPA) charged solely with data import processes, enabling seamless data imports from external systems like SAP and iValua. This infrastructure ensures that management always operates with accurate, up-to-date information.

The outlined work emphasizes some critical contributions made to the overall system:

- **Cost Control module**: designed to streamline financial savings management, this new component replaces traditional, Excel-based workflows with an intuitive, spreadsheet-like interface powered by the Handsontable JavaScript library. Features such as real-time validation, automated calculations, and dynamic data linking simplify

the analysis of expenditure patterns, providing top management with actionable insights. The module enhances decision-making and improves resource allocation, offering a modernized approach that balances user familiarity with advanced functionality.

- **New API** for **Invoice** import: by leveraging Apache POI for data validation and transformation, and later introducing asynchronous processing to handle large datasets, the newly implemented API automates the import process by offering a dedicated interface to RPA agents, while also addressing scalability limitations. These refinements ensure the system can process high data volumes efficiently, maintaining uninterrupted operations while guaranteeing the availability of current and relevant information.

Moreover, significant performance optimizations were implemented for the **Sustainability Vendor Rating** (SVR) dashboard, which uses the Invoice table as one of its main data sources. By reducing query loads and improving responsiveness, these enhancements allow the system to effectively manage millions of records while maintaining an intuitive and efficient user experience.

The results of these enhancements underscore the transformative impact of the project: the modernization of workflows through the Cost Control module, coupled with the optimized API layer and SVR dashboard, has significantly improved system performance and usability; query response times have been drastically reduced, and operational efficiency has been enhanced across all the refined components, demonstrating the project's success in meeting its objectives and laying a solid foundation for future development. Through its thoughtfully designed architecture and strategic use of advanced technologies, SmartSpend exemplifies the principles of flexibility and effectiveness essential for contemporary enterprise solutions. This work not only underscores the application's ability to meet the demands of top management, but also demonstrates its adaptability for future enhancements and continuous optimization: by advancing operational efficiency and decision-making capabilities, the project highlights the indispensable role of technology in achieving business objectives in an ever-changing digital landscape.

Contents

List of Tables 7

List of Figures 8

1 Introduction 11

2 Background 15
- 2.1 Web applications: generalities, advantages and disadvantages 15
- 2.2 Spring Boot: a reliable backbone for robust applications . 20
- 2.3 RPA 24

3 Methodology 29
- 3.1 In-depth analysis of involved technologies 29
  - 3.1.1 Java MVC, Spring Data and JPA 30
  - 3.1.2 Freemarker Template Engine 33
  - 3.1.3 Microsoft SQL Server 35
- 3.2 System Architecture 38
- 3.3 SmartSpend: Cost Control module 41
  - 3.3.1 Processes description 41
  - 3.3.2 Implementation details 44
- 3.4 API Layer: RPA agents integration 51
  - 3.4.1 API implementation, InvoiceController and InvoiceService 51
  - 3.4.2 Async API for RPA agents integration 56
- 3.5 PSC: SVR dashboard analysis and performance boost . . . 62
  - 3.5.1 Dashboard overview 62
  - 3.5.2 Performance analysis and improvement 64

4 Results 67

5 Future works and open issues 71

Bibliography 75

List of Tables

2.1 Pros and cons of web applications: a summary 18
4.1 KPIs comparison before and after development (results obtained via Google Chrome developer tools' network tab) . 68

List of Figures

2.1 Example project structure for a Spring Boot application, following component-level foldering (taken from `https://tinyurl.com/sb-structure`) 21
2.2 Example workflow achievable via RPA (taken from `https://tinyurl.com/rpa-flow`) 24
3.1 Diagram showcasing the MVC pattern (taken from `https://tinyurl.com/mvc-diagr`) 30
3.2 Overall architecture of the system 38
3.3 BPMN diagram of the savings identification process 42
3.4 BPMN of the cost control process 43
3.5 Example instantiation of the Handsontable, to showcase the various features used, pt.1 45
3.6 Example instantiation of the Handsontable, to showcase the various features used, pt.2 46
3.7 Example events to toggle modes in the table and persist edits 47
3.8 Custom hook to manage automatic VAT calculation 48
3.9 Structure of the **Saving** entity 49
3.10 Structure of the **Edit** entity 49
3.11 Initial implementation of the **InvoiceController** 52
3.12 Implementation of the *processInvoiceFile* method 53
3.13 Implementation of the *processInvoiceFile* method (cont.) . 54
3.14 Structure of the **InvoiceImport** enum 55
3.15 Implementation of the *validateHeaderRow* method 55
3.16 Structure of the **ImportResponse** class 56
3.17 Async implementation of the **InvoiceController** 57
3.18 Structure of the **PollStatus** entity 58
3.19 Implementation of the updated *processInvoiceFile* method 59

3.20 Implementation of the updated *processInvoiceFile* method, cont. 60
3.21 Implementation of the *calculateStepsForPercentageUpdate* utility method 61
3.22 Mockup of the SVR dashboard 64
3.23 Structure of the **SupplierMonthlySummary** 66

Chapter 1

Introduction

In the rapidly evolving digital age, procurement and supply chain management have become central to the pursuit of operational efficiency. Organizations increasingly turn to technologies like Artificial Intelligence, Robotic Process Automation (RPA), and data analytics to streamline and facilitate decision-making processes by amplifying the capabilities that more traditional solutions provide. This thesis is positioned within such context, focusing on the development and enhancement of a web application aimed at supporting top-management reporting.

The work was carried out in collaboration with Sprint Reply[1], a specialized company within the Reply group that focuses on digital transformation and intelligent automation. Sprint Reply's expertise lies in leveraging advanced technologies to optimize business processes, particularly in procurement, where the integration of innovative approaches is critical for driving efficiency and improving performance.

At the heart of the project is SmartSpend, a web application built with Spring Boot and the Freemarker template engine, which functions as the primary reporting tool for procurement data. It serves top-management users by presenting coherent and comprehensive insights through a series of sophisticated dashboards and reports; the data it relies on is housed in the ProcurementServiceCatalogue, the core database that aggregates procurement-related information from various external systems. Through

[1] https://www.reply.com/sprint-reply/it/

this reporting interface, SmartSpend enables management to make informed, data-driven decisions, significantly enhancing visibility into procurement activities.

The application is supported by the Smart Procurement API Layer, a secondary Spring Boot application that plays a key role in the system's data flow by exposing a set of APIs for automatic or manual data import. It is mainly used by software robots that, after extracting useful data from systems like SAP and iValua, are able to package it into structured formats (like an Excel file for example) and upload it into the main database by using the interfaces provided.

Built upon Microsoft SQL Server, the ProcurementServiceCatalogue mentioned earlier serves as a centralized repository for all procurement data: it provides the necessary information to generate detailed reports and dashboards, ensuring accuracy and consistent updates by enabling a seamless flow between external sources, the API layer, and the main application itself, thus creating an efficient and cohesive system.

Although RPA robots play a crucial role in automating the data flow between external systems and the central database, their development lies outside the scope of this thesis. Instead, the focus is on two key aspects of SmartSpend's evolution: first, the design and implementation of a Cost Control module will be analyzed; developed to monitor and evaluate financial savings initiatives within company's processes, this addition enables top management to analyze expenditure patterns effectively, fostering data-driven decision-making and optimizing resource allocation. Second, the enhancement of the API Layer and ProcurementServiceCatalogue will be dissected, highlighting how it enables faster access to large dashboards and allows an automated data import process from external systems, like SAP and iValua. These core contributions exemplify how modular architecture can address the complex requirements of enterprise applications by balancing flexibility, scalability, and maintainability. To present this work systematically, the thesis is structured as follows:

- **Chapter 1: Introduction** – Sets the context of the project, outlining the growing importance of digital solutions in procurement

management. The chapter introduces SmartSpend and its core architecture, detailing its integration with the ProcurementServiceCatalogue database, the Smart Procurement API Layer, and external data sources via RPA agents. It also presents the main objectives and the modular approach used to enhance the application's functionality and maintainability.

- **Chapter 2: Background** – Provides a detailed overview of the technological landscape. It discusses the advantages and challenges of web applications, with a particular focus on the Spring Boot framework, and explores the role of RPA in automating data import processes. This chapter establishes the technical foundation for the work carried out.

- **Chapter 3: Methodology** – The core chapter, detailing the key contributions:

    1. **Cost Control module**: Designed to streamline savings identification and management processes, this module features an interactive Excel-like table built with Handsontable. It provides functionalities such as column resizing, real-time validation, smart linking, and robust data traceability through a dual-entity backend model (Savings and Edits). This design facilitates the transition from manual Excel workflows to an integrated, modern system.

    2. **API and SVR dashboard enhancements**: A new API for invoice data imports was implemented to handle large Excel files efficiently. Leveraging the Apache POI library, the API validates and processes incoming data, storing it in the ProcurementServiceCatalogue; asynchronous processing and a polling mechanism ensure reliability, addressing connection timeout issues. Additionally, significant performance optimizations were made to the Sustainability Vendor Rating (SVR) dashboard, reducing query loads and improving page responsiveness. These enhancements enable smooth handling of millions of data rows, delivering a faster and more user-friendly experience.

- **Chapter 4: Results** – Presents a thorough evaluation of the implemented solutions, highlighting both qualitative and quantitative outcomes: for the Cost Control module, it focuses on the modernization of workflows and enhanced user experience; for the API and SVR dashboard, it quantifies performance improvements, such as drastically reducing page load times in many different scenarios.

- **Chapter 5: Future works and open issues** – Concludes the thesis by outlining potential areas for improvement. Key proposals include transitioning to a Single Page Application (SPA) architecture using Angular for better frontend-backend separation and integrating monitoring tools like Grafana and Loki to enhance performance tracking and issue resolution.

In its development, this thesis is going to illustrate how a modular architecture can be used efficiently to provide integrated and comprehensive reporting from multiple sources, while also examining how advanced technologies can address complex enterprise challenges without forgetting efficiency, sustainability and innovation.

Chapter 2

Background

2.1 Web applications: generalities, advantages and disadvantages

Web applications have become an essential part of modern software development, offering flexibility, scalability, and high accessibility to both businesses and final users. As stated on the GeeksForGeeks website (a major tech-related learning platform) [1], they can be seen as software programs that run on remote servers and are accessed through browsers over the internet. Unlike traditional desktop ones, which require installation and regular updates on individual devices, web applications centralize processing and data management on servers, allowing users to interact with the system from any device or location. This model offers several important benefits that have driven the widespread adoption of web-based solutions in enterprise environments, while still bearing some drawbacks and limitations. In this section, we will explore both the strengths and challenges of this approach, and also introduce modern trends that are more and more popular in the field.

One of the standout advantages of web applications is their remarkable cross-platform compatibility. Unlike traditional software, which often operates within the confines of specific operating systems, web applications leverage the capabilities of web browsers, enabling seamless access across a diverse array of platforms. Whether users are on Windows, MacOS, Linux, or even mobile systems such as Android and iOS, they can easily engage with the application without any compatibility issues. This universality

is particularly beneficial for businesses, as it allows them to reach a wider and more varied audience without the complications and costs typically associated with developing separate versions tailored to each platform. Furthermore, because users can interact with web applications directly through their browsers, there is no need for cumbersome software installations (basically every new device has a pre-installed browser). This aspect not only simplifies the user experience, but also significantly lowers barriers to adoption, making web applications especially appealing in enterprise environments where ease of use and accessibility are crucial. In addition to their cross-platform advantages, web applications excel in terms of maintenance and updates: their centralized architecture means that the application logic resides on servers rather than individual user devices. As a result, all users automatically access the latest version of the software without having to engage in manual updates or installations. This streamlined approach to maintenance not only alleviates the burden on IT teams, but also ensures that users are less likely to operate outdated versions of the software: identified vulnerabilities can be addressed swiftly from one side only, enhancing overall security and reliability. Another compelling feature of web applications is their remote accessibility: users can engage with the system from virtually any location, as long as they have an internet connection. This flexibility is a significant advantage for both businesses and final customers, since it fosters a more dynamic and adaptable work environment. In today's increasingly mobile and interconnected world, the ability to access needed tools from anywhere enhances productivity and collaboration, therefore resulting in a increased appeal of this solution.

Although the benefits of web applications are substantial and positively impactful, they also come with notable downsides that need to be taken into consideration. While being one of their advantages, their dependence on internet connectivity can also be seen as a potential issue: in environments where stable and relatively high-speed connections are not guaranteed, users may face challenges such as latency or even complete inaccessibility if their internet access is interrupted. Such disruptions can lead to frustration and diminish the overall experience, particularly in time-sensitive scenarios. Moreover, the very nature of web applications makes them susceptible to various cyber threats: being accessible on the

open internet leaves them vulnerable to risks such as data breaches, denial-of-service (DoS) attacks and their distributed variants (DDoS), along with a range of other malicious activities. Implementing robust security measures (like data encryption, secure authentication protocols, and regular security audits) can significantly reduce these risks, but it's crucial to understand that such strategies can lead to considerable costs if not done in the right way: without proper planning and allocation, the investment required to establish adequate security countermeasures can strain the development process of a web application, potentially diverting attention and resources from other essential aspects of the project. Another crucial aspect to consider is performance, which plays a significant role in determining the overall user experience and effectiveness of a web application. Unlike desktop applications, that can fully harness the processing power of the user's machine, web applications rely heavily on server-side processing for handling complex tasks. In this model, the client — typically the user's browser — functions primarily as an interface, while the majority of the computational work takes place on the server. This kind of reliance introduces several challenges, particularly during periods of high traffic or when server resources are constrained: when the demand exceeds its capacity, users may experience slower response times, and consequently a significant reduction in terms of productivity. This issue is especially critical in scenarios where performance is a primary necessity, such as in real-time data processing or mission-critical applications, where even minor delays can have considerable negative impacts on operations. To mitigate these issues, organizations must invest in scalable infrastructures and adopt strategies like load balancing, distributed databases and cloud-based services, that allow for dynamic resources adjustment based on demand. However, implementing such solutions or buying already made ones can be costly, and careful planning is required to achieve an efficient and cost-effective result product. Additionally, these optimizations must be performed in a continuous process, involving regular monitoring, testing, and fine-tuning to ensure that the application can handle varying levels of usage without compromising the user experience.

Pros	**Cons**
Accessible on any device with a web browser, regardless of the operating system (Windows, macOS, Linux, Android, iOS).	Require a stable internet connection; interruptions cause inaccessibility.
Users can access directly from the browser without downloading or installing software.	Rely on server-side processing, which may cause latency under high traffic.
Centralized updates mean all users access the latest version, reducing IT overhead.	Require investment in infrastructure like load balancing and distributed databases.
Enable usage from any location with internet connectivity, fostering productivity and collaboration.	Exposed to network vulnerabilities, including risks of cyberattacks (e.g., DDoS, data breaches).
A single version serves all platforms, reducing the cost and complexity of separate versions.	Continuous monitoring and optimization required to maintain performance.

Table 2.1. Pros and cons of web applications: a summary

Beyond these core characteristics, recent advancements in web application development have introduced innovative models that address some of these traditional limitations while enhancing the user experience. In this sense, Single-Page Applications (SPAs)[2] represent a notable shift from traditional multi-page architectures: by loading a single HTML page that dynamically updates content as users interact with it, they are able to deliver a seamless and uninterrupted experience. JavaScript frameworks such as React, Angular, and Vue.js enable SPAs to provide almost instantaneous navigation within an app, bypassing the constant reloads that often slow down traditional applications. For users, this means fewer delays and a more cohesive interaction with the application, a benefit that is particularly valuable in data-driven environments where rapid feedback is essential; social networks, project management tools, and real-time collaboration platforms are just a few examples of applications leveraging this kind of architecture. Naturally, SPAs also pose certain challenges: since

they heavily rely on JavaScript for content rendering, they can hinder and limit SEO[1], as search engines have traditionally struggled to effectively crawls and index JS-based content; additionally, SPAs require extensive and careful state managment to mantain consistency across various components, especially in applications with complex and intricated data dependencies. To address this issues, techniques like server-side rendering (SSR)[2] and libraries like Redux[3] or Vuex[4] have been developed, ensuring proper scalability and consistency.

While SPAs prioritize smooth and responsive user interactions, Progressive Web Applications (PWAs)[3] take another direction by integrating features traditionally associated with native mobile applications, such as offline functionality, push notifications, and even home screen access. They achieve this versatility by using service workers—background scripts that cache assets and enable access even in environments with intermittent connectivity: e-commerce sites, news platforms, and financial applications benefit from this offline availability, as it helps maintain a seamless user experience despite potential network issues. By adding web app manifests[5], developers can enable users to "install" PWAs directly to their home screens, bypassing app stores entirely and facilitating instant access without downloads.

Together, SPAs and PWAs represent the growing emphasis on flexible, user-centric web application design: the formers facilitate highly interactive applications that prioritize responsiveness, while the latters enhance accessibility and resilience by offering offline capabilities and native-like functionality. Both approaches highlight the ongoing evolution of web applications toward solutions that cater to an increasingly mobile, on-demand, and interconnected user base: as organizations continue to pursue digital transformation, the demand for flexible, adaptable web applications that meet both present and future needs is expected to grow, underscoring the importance of selecting frameworks and technologies that align with long-term business objectives.

[1] Search Engine Optimization, https://tinyurl.com/seo-doc

[2] https://tinyurl.com/ssr-def

[3] https://redux.js.org/

[4] https://vuex.vuejs.org/

[5] https://tinyurl.com/webmanif

2.2 Spring Boot: a reliable backbone for robust applications

In this context, **Spring Boot**[6] has emerged as an essential framework. Built on the foundation of the Spring Framework [7], which has long been integral to Java development, it streamlines many of the challenges traditionally faced in enterprise scenarios. Its principle of *"convention over configuration"* allows developers to start a new project quickly, leveraging pre-configured defaults that reduce manual setup; this not only shortens development cycles, but also lowers the risk of configuration errors, making it an ideal choice in environments where speed and reliability are crucial. Such approach is supported by the extensive use of annotations [4], that are special markers or metadata within the code that instruct Spring Boot on how to configure, initialize, and manage components, replacing large configuration files with concise and declarative code. For example, **@SpringBootApplication** [5] combines several foundational annotations to enable automatic configuration and component scanning, simplifying the setup of a new project. As analyzed by Baeldung author Loredana Crusoveanu [6], this particular type of configuration relies heavily on a core principle known as Inversion of Control (IoC), which places the control of object creation and dependency management over to the Spring IoC container: by managing dependencies centrally, components are allowed to function independently, improving modularity and adaptability. Dependency Injection (DI), a practical application of IoC, further enables Spring Boot to automatically inject required dependencies into components via annotations like **@Autowired** [6]: this removes the need for manual instantiation within code, reducing coupling between components and improving testability.

[6] https://spring.io/projects/spring-boot

[7] https://spring.io/projects/spring-framework

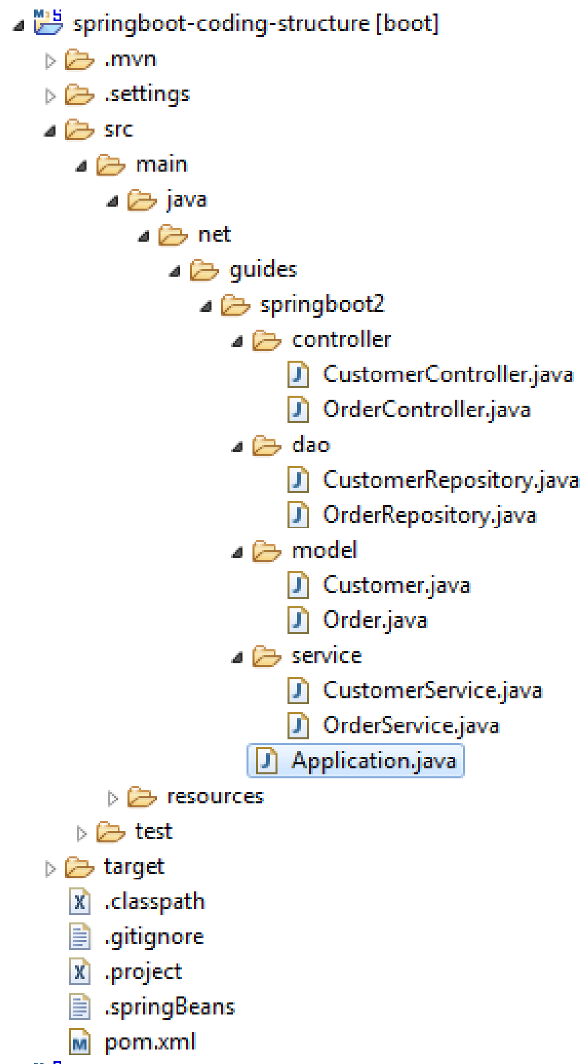

Figure 2.1. Example project structure for a Spring Boot application, following component-level foldering (taken from https://tinyurl.com/sb-structure)

Another Spring Boot's standout feature is its seamless integration with a broad spectrum of databases, both relational (such as Microsoft SQL Server, MySQL, and PostgreSQL) and non-relational (such as MongoDB and Cassandra) ones. This versatility is critical for enterprise applications that require efficient and flexible data storage solutions. By using abstractions like Spring Data JPA (Java Persistence API) for relational databases, and analogous solutions for non-relational ones, Spring Boot drastically reduces the complexity of data interactions, enabling developers to use

declarative code instead of intricate SQL or NoSQL queries. This simplification not only enhances development efficiency, but also mitigates errors, contributing to better long-term maintainability and scalability.

Additionally, Spring Boot's embedded web server functionality is another key advantage. By bundling the application with an embedded server like Tomcat or Jetty, there is need for external server configurations: this allows the entire application to be packaged as a self-contained unit, simplifying deployment across different environments; the uniformity obtained reduces the risk of errors due to discrepancies between development, testing, and production environments. Furthermore, Spring Boot includes production-ready features such as health monitoring, metrics, and optimized deployment settings, making it particularly suitable for enterprise-grade applications that demand operational reliability.

Although it is often associated with microservices architecture, it is important to note that Spring Boot is equally effective for monolithic applications. While microservices enable large-scale systems to be broken down into smaller, independently deployable services, not all enterprise applications require this level of decoupling. In many cases, traditional monolithic architectures, where all application components are tightly integrated, may still be the most appropriate solution, especially in scenarios where modularity and separation of concerns can be managed within a single codebase. Spring Boot's flexibility allows it to support both architectural styles, enabling developers to choose the approach that best fits their project requirements without forcing them into a specific paradigm.

While not being the focal point of this work, it is worth briefly discussing the role of microservices in modern software development to understand the broader capabilities that Spring Boot offers. According to Prakash Raj Ojha[7] in his article *"Spring Boot and Cloud-Native Architectures: Building Scalable and Resilient Applications"*, microservices represent an architectural shift that has allowed businesses to manage complexity in large systems more effectively. By breaking down an application into loosely coupled services, they enable independent development, scaling, and deployment, supporting the advancement of environments where different components of a system need to evolve at different rates. For example, Netflix, a pioneer in microservices, manages over 700 of them to handle more than 1 billion streaming hours weekly. This trend is further reflected in broader cloud adoption patterns: as we can see in *O'Reilly's*

Cloud Adoption Survey of 2021 [8], 90% of respondents indicated that their organizations were already using cloud technologies, with **Amazon Web Services** (AWS), **Microsoft Azure**, and **Google Cloud** being the most widely adopted platforms; the flexibility and scalability that they provide align with the principles of microservices architecture, which requires infrastructure capable of dynamically adjusting to workload demands.

Spring Boot's relevance in modern web application development is not only due to its technical strengths but also its alignment with broader trends. According to a survey by **JetBrains**[9] in 2023, over 70% of Java developers reported using it for their projects, underscoring its dominance in the enterprise web development space. Its robust ecosystem, combined with ease of integration with cloud platforms and flexible architectures, has made Spring Boot a go-to framework for businesses seeking to build scalable, maintainable, and flexible systems in an increasingly cloud-centric world. As enterprise software continues to grow in complexity, the need for solutions that not only simplify development, but also streamline operational processes, becomes more apparent. With the increasing volume of data and the repetitive nature of certain tasks, organizations are turning to advanced automation strategies to enhance efficiency and maintain agility. In this evolving landscape, technologies that complement web development frameworks are playing an ever more critical role in modern enterprise environments.

2.3 RPA

Robotic Process Automation (RPA) has emerged as one of the main solutions to support the digital transformation journeys across many different sectors: in an increasingly data-driven world, organizations must implement automated technologies to improve operational efficiency and preserve a competitive advantage. As Moreira, Mamede and Santos found in their study [10], it is a well known concept since the 2000s, with the first steps of development performed even in the 1990s; its definition can be grasped by closely analyzing the words that compose the acronym:

- *Robotic*: something that imitates human behaviour, with or without supervision;
- *Process*: series of events, tasks to be performed;
- *Automation*: perform a certain job with assistance, not manually.

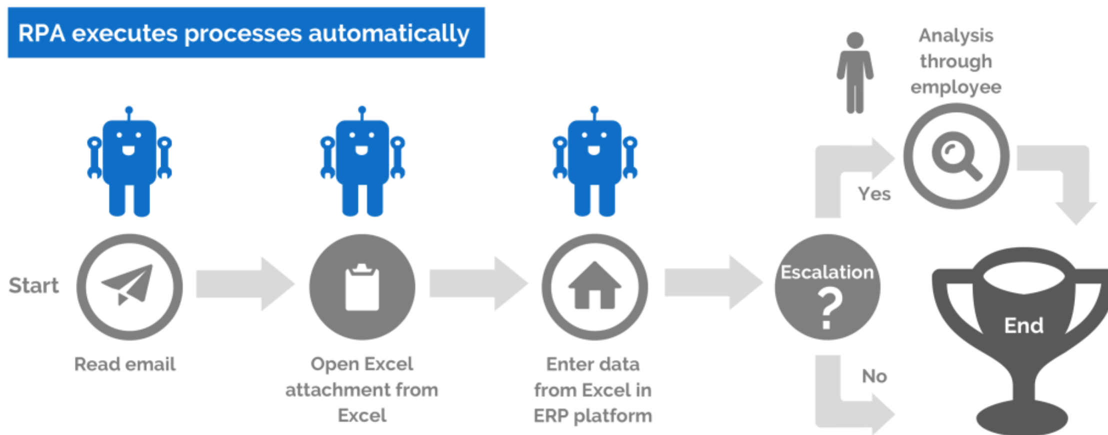

Figure 2.2. Example workflow achievable via RPA (taken from https://tinyurl.com/rpa-flow)

Essentially, RPA automates repetitive, structured, and rule-based operations by developing software bots that duplicate human actions; these bots may communicate with a variety of systems via user interfaces, just as humans do, without requiring deep integrations or large changes to current IT infrastructures. This non-intrusive approach has led to a valid

alternative for businesses looking to optimize operations, cut costs, and eliminate human error in manual procedures.

Its increased importance can be attributed to a variety of interconnected causes. First of all, it enables corporations to optimize their processes while retaining their current systems. Since many businesses continue to rely on critical legacy pieces of software, RPA provides a feasible option to improve efficiency without facing the high costs and risks associated with large upgrades, making it an especially appealing alternative for large companies with existing IT frameworks. Moreover, as stated in the previously mentioned paper[10], the COVID-19 pandemic has served as a catalyst for its adoption, pushing organizations to reassess their operational capabilities. As remote work, labor shortages and operational disruptions became the norm, the need for reliable automation solutions became increasingly apparent: RPA not only helped in maintaining productivity during those challenging times, but also gave enterprises a greater agility to respond to future disruptions.

The current state of RPA reflects a technology that has matured significantly and is now considered an integral part of digital transformation strategies for organizations worldwide. Industry reports from 2020/21 indicated a rapid increase in its adoption across sectors, with heavy investments being at stake: according to Gartner[11], the global RPA market was projected to reach over $2 billion by 2022, underscoring the technology's widespread acceptance and implementation. Today those investments paid off, and, as businesses integrate it deeply into their methods, they are witnessing substantial improvements in efficiency, accuracy, and productivity; it has transitioned from a novel innovation to a mainstream tool, facilitating a more agile approach to business processes. The landscape of RPA is continually evolving, with advancements leading to more sophisticated applications and use cases: this includes the development of user-friendly platforms that allow organizations to deploy automation solutions more rapidly and with less reliance on IT departments.

Additionally, RPA operates around the clock, ensuring continuous workflows and enabling quicker turnaround times for tasks such as data processing, invoicing, and financial reporting: this 24/7 operational capability not only accelerates business processes but also improves accuracy. By reducing human error, particularly in data-sensitive sectors like finance or banking, it helps mitigate risks associated with data entry mistakes and

compliance issues, ensuring that tasks are performed reliably, consistently and without deviating from established protocols, while also granting a high level of transparency and accountability due to more or less extensive logging capabilities. Cost savings are another compelling advantage of RPA: by automating mundane tasks, companies can decrease their reliance on manual labor, leading to significant reductions in operational costs. This shift allows organizations to reallocate human resources toward more complex and strategic activities that add greater value to the business: the ability to refocus employee efforts on a higher level not only improves productivity, but also enhances job satisfaction, as employees engage in more meaningful and intellectually stimulating tasks

Despite its many advantages, RPA is not without its challenges. One of the primary limitations is its dependence on structured, rule-based processes: while it excels at automating tasks with clearly defined inputs and outputs, it struggles with processes that require judgment, intuition, or the handling of unstructured data. For instance, if a document's format changes unexpectedly, a bot may fail or produce inaccurate results, necessitating human intervention to resolve the issue; this limitation highlights the importance of carefully evaluating which processes are suitable for automation, and which ones still require a constant human intervention, from beginning to end. Additionally, while RPA is generally easier to implement than traditional automation solutions, it still requires ongoing maintenance and monitoring. As organizations evolve and their systems change, robots must be regularly updated to accommodate these alterations; failure to do so can lead to inefficiencies and potential disruptions in business operations. Businesses must also consider scalability: while small or medium scale tasks are more suited to this type of automation, deploying it across an entire organization or within complex processes can introduce new challenges that require careful design and governance. Without a structured approach, there is a risk that such initiatives could exacerbate existing inefficiencies rather than eliminate them. Another challenge is the potential resistance from employees: while RPA aims to reduce the burden of repetitive tasks, some workers may fear job displacement. To address these concerns, organizations should implement change management strategies that highlight its advantages, focusing on how it complements human efforts rather than replacing them. By investing in training and re-skilling programs, companies can help employees adapt

to new roles that leverage automation, fostering a partnership where human expertise guides higher-level decision-making while automated systems handle routine tasks.

Looking ahead, the future of RPA is closely tied to advancements in Artificial Intelligence (AI) and Machine Learning (ML). In its traditional version, it focuses on structured, rule-based tasks, but the integration of AI technologies can lead to more *"cognitive"* forms of automation. This emerging trend, often referred to as Intelligent Automation or Hyperautomation, combines RPA with AI capabilities, allowing bots to manage more complex processes that require decision-making, interpretation of unstructured data, and adaptability to new scenarios. As organizations seek to automate not only routine tasks, but also sophisticated workflows, the synergy between RPA and AI becomes increasingly crucial. By leveraging machine learning algorithms, performance can be continuously improved based on historical data and outcomes, enabling more intelligent decision-making; for example, AI-enhanced solutions can analyze customer interactions, assess sentiment, and adjust processes in real time to provide a more personalized experience. Moreover, the deeper integration between them allows to tackle unstructured data, such as emails or scanned documents: Natural Language Processing (NLP) can be utilized to interpret and extract valuable insights, further broadening the scope of automation; this capability is particularly beneficial for industries like healthcare and finance, where data is often complex and heterogeneous. As RPA evolves into a more comprehensive solution, it will play a vital role in reshaping the way organizations operate. Companies will increasingly rely on Intelligent Automation to enhance their operational capabilities, drive efficiency, and foster innovation. This transition is not just about replacing manual processes; it's about creating a new paradigm where human and machine collaboration leads to greater agility, responsiveness, and resilience in the face of rapidly changing business environments.

Chapter 3

Methodology

In this chapter, we are going to dive deeper into the details of the work done. We will start with additional and meaningful details of the technologies involved in the project, then we will outline the overall architecture of the system, ending with a thorough explanation of the three major interventions performed in the scope of the thesis.

3.1 In-depth analysis of involved technologies

This section delves into the core technological foundation of the project. It begins by exploring the Model-View-Controller (MVC) pattern, its implementation using Spring MVC and the integration of Spring Data and JPA for seamless database interactions, emphasizing their role in simplifying data access and persistence; it then transitions to an analysis of the Freemarker template engine, showcasing its contributions to dynamic content generation and separation of concerns in the presentation layer; finally, the capabilities of Microsoft SQL Server are discussed, focusing on its advanced features that support data management, performance optimization, and security in enterprise applications.

3.1.1 Java MVC, Spring Data and JPA

While working on large and complex codebases, such as in the context of this thesis, establishing a clear and well-structured architecture is essential for maintainability, scalability, and ease of development. One of the most widely adopted architectural patterns, particularly for modern web applications and GUIs, is the Model-View-Controller (MVC) paradigm. Introduced for the first time in 1979 [12], it conceptually divides an application into three interconnected and fundamental blocks:

- *Model*: it represents the application's data and business logic;
- *View*: it's the presentation layer, responsible for rendering the user interface after receiving data from the controller;
- *Controller*: acting as an intermediary, it is in charge of handling user inputs and directing interactions between the Model and the View.

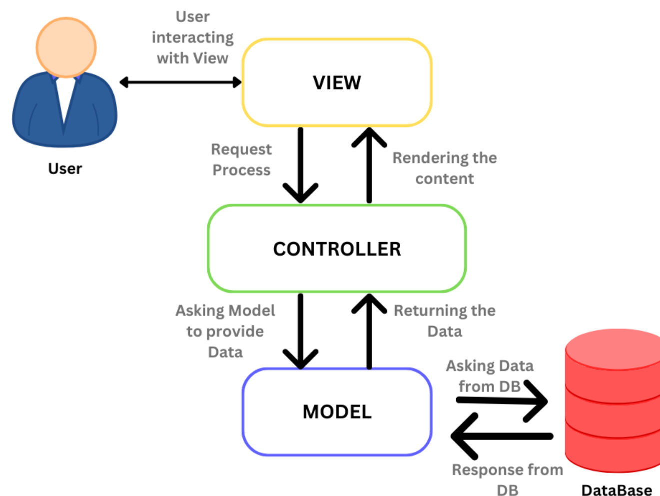

Figure 3.1. Diagram showcasing the MVC pattern (taken from https://tinyurl.com/mvc-diagr)

This separation of concerns ensures that changes in one part of the application (e.g., updating how data is rendered in the view) do not necessitate changes in other parts (e.g., the logic in the controller), thus enabling

a high degree of flexibility while extending already existing components or adding new functionalities.

Spring MVC, one of the many concrete implementations of the mentioned pattern, builds upon its foundation by implementing a well-defined structure through specific annotations that designate the roles and responsibilities of each component. Controllers, annotated with **@Controller** or **@RestController** [13], are classes in charge of handling HTTP requests, mapping them to the appropriate endpoints with methods further specified using **@RequestMapping** or its specific derived ones [14]; this approach simplifies request handling, allowing each controller to route and process incoming data seamlessly. They rely on classes known as Services, marked by the **@Service** [15] annotation, that normally encapsulate the business logic of the application while also coordinating all accesses to the data layer through dedicated Data Access Objects. These DAOs, usually annotated with **@Repository** [16], represent the persistence layer and serve as interfaces between the application and the database, offering CRUD functionality and exception handling through Spring's persistence exception translation mechanism, thereby further promoting consistency and reliability. Together, this layered, annotation-driven approach ensures that each component has a well-defined role, enabling clear pathways for communication and data flow across layers, enhancing overall readability and maintainability of the code and also facilitating rapid development in complex scenarios (as said before, developers are able to focus on discrete parts of the application without compromising other layers).

Beyond this foundational MVC-like structure, Spring Boot introduces further abstraction and automation through Spring Data, a core component of the broader Spring ecosystem that enables a consistent approach to data access across different database types. The interfaces it provides are able to abstract away much of the boilerplate code typically involved in such operations; by supporting supports multiple databases (both relational and noSQL), they are able to provide enough flexibility to switch between different solutions without major code refactoring. This is particularly important in enterprise scenarios, that require the ability to scale and adapt to changing business requirements: its modular nature ensures that developers can integrate new database technologies as needed, all while maintaining a consistent code structure[17].

A key aspect of Spring Data is its repository abstraction, that enables the definition of common data access methods without needing to manually implement them: for example, by specifying methods like *findByLastName* within a repository interface, it is possible to rely on automatic query derivation to generate the necessary database-specific queries, simply based on the method name. This declarative programming approach enables developers to focus on what they want to achieve, not on the management of internal details, thus not only accelerating development and enhancing code readability, but also minimizing the risk of errors, as the framework ensures consistency and precision in query creation[18].

Building on Spring Data, Spring Data JPA extends its functionality by integrating with the Java Persistence API (JPA), a specification for Object-Relational Mapping (ORM) that maps Java objects to relational database tables simplifying record management. To enable this mapping, developers mark classes that represent database entities with the **@Entity** [19] annotation: they act as templates for database tables, with each instance representing a row in the table; the fields within these classes map to columns, allowing for seamless interaction between Java objects and database records. Each **@Entity** class requires an **@Id** [20] annotation to specify a primary key, which uniquely identifies a record within the table, and additional annotations like **@Column** [21] can be used to customize the column names, data types, and other properties as required. Moreover, Spring JPA adds to this foundational mapping by offering built-in support for auditing and pagination, two essential features in environments where large datasets are handled, requiring comprehensive tracking. Auditing allows for the automatic recording of entity lifecycle events, such as creation and modification timestamps: this is invaluable for enterprise applications, where compliance and traceability are often essential. Additionally, pagination is critical for handling substantial amounts of data by retrieving smaller, more manageable chunks, preventing performance bottlenecks and ensuring proper responsiveness even with high volumes of data.

Tightly integrated with Spring Data JPA, Hibernate[1] is the most widely used implementation of these specifications and serves as a powerful ORM (Object-Relational Mapping) framework. As a JPA provider, it further

[1] https://hibernate.org/

optimizes data interactions with several advanced features, such as:

- Lazy loading: enabled by specifying proper join attributes within an Entity (with annotations like **@OneToMany** [23] or **@ManyToMany** [24], that contain a property to set the preferred fetch mode for linked entities), it enables the retrieval of the necessary data when only when it is explicitly required, reducing unnecessary database calls;
- Caching: used to store frequently accessed data in memory, minimizing repetitive queries and improving performance;
- Automatic dirty checking: Hibernate is able to keep track of changes to objects, ensuring that only modified data is persisted and optimizing database interactions.

3.1.2 Freemarker Template Engine

FreeMarker[2] is a comprehensive, Java-friendly templating engine, widely used to produce dynamic content for web applications while maintaining a clear separation between business logic and presentation. Known for its compatibility with frameworks like Spring Boot, FreeMarker allows developers to define reusable and modular templates (characterized by the .ftl extension, meaning FreeMarker Template Language) that generate HTML, XML, JSON, and other text-based formats with minimal configuration. By isolating view logic from backend functionality, FreeMarker ensures that designers and developers can work independently, a practice that enhances maintainability, scalability, and modularity in many different scenarios.

At its core, FreeMarker's syntax is designed to be both accessible and highly functional, enabling the implementation of complex data-driven logic directly within templates: it's possible to interact seamlessly with Java objects, supporting diverse data types such as JavaBeans, lists, and maps. This capability is reinforced by a set of directives and expressions that allow the usage of conditionals, loops, and complex calculations directly inside the template. The conditional logic, for example, enables to

[2] https://freemarker.apache.org/

dynamically display content based on the application's data, while looping structures can iterate over collections of data with minimal additional code. For repetitive content, FreeMarker's macro feature is invaluable, as it enables the encapsulation of components into reusable, callable modules that simplify maintenance and improve consistency across templates.

In addition to basic structures, FreeMarker provides a range of built-in functions for manipulating strings, numbers, dates, and even locale-specific data: these allow to fine-tune how data is presented, which is especially useful in applications requiring extensive formatting or internationalization. Its support for locale-sensitive formatting is particularly powerful, allowing to adapt content based on the user's language or region settings: for instance, date and number formats automatically matching local standards and ensuring that applications deliver a user-friendly experience across multiple languages and regions. Moreover, there is also a robust support for custom directives and user-defined functions, that enables the possibility to extend the templating language with project-specific functions when additional customization is required.

Strictly linked to this last feature is the support for custom tags and plugins, which allow further expansion of template capabilities beyond standard functions and macros. By creating custom tags, developers can introduce specialized template components that can encapsulate complex behaviors, such as dynamic data retrieval, condition-based transformations, or custom HTML generation. By standardizing these operations once, it's possible to reuse them across multiple projects, further enhancing FreeMarker's suitability for large-scale applications with complex presentation needs. This extensibility also supports FreeMarker's utility in hybrid systems, where it can serve as the frontend for applications that leverage multiple backend systems or microservices.

Another distinctive feature of FreeMarker is its flexible error-handling capabilities: with built-in directives to handle null values, missing variables, and data inconsistencies, it allows templates to degrade gracefully when data is incomplete or unexpected. This capability is essential for environments where data quality may vary, ensuring that the user experience remains unaffected even if some data is missing. By using the *default* operator, developers can specify fallback values for variables, and the *attempt* directive enables recovery mechanisms within templates, allowing complex data processing without introducing code-breaking errors. These

features improved drastically the general resilience and fault-tolerance of the project, reducing the risk of disruptions in production environments.

Integrating FreeMarker with Spring Boot offers additional advantages, as the latter provides out-of-the-box support for many template engines within its MVC structure. By placing templates within the designated *resources/templates* directory, developers benefit from Spring Boot's auto-configuration, which simplifies the rendering process by automatically loading them at runtime. This integration not only minimizes configuration but also aligns with Spring Boot's embedded server model, ensuring consistency across development, testing, and production environments without requiring additional setup for deployment.

Lastly, FreeMarker is compatible with a variety of caching strategies, including integration with external caching mechanisms such as Redis or Ehcache, which is critical for performance in high-traffic applications: by caching pre-rendered templates or data, it's possible to reduce response times and optimize server resources, especially in scenarios where templates are frequently reused with similar data. This caching integration also complements Spring Boot's performance-oriented configurations, creating an efficient, scalable environment ideal for enterprise deployment.

3.1.3 Microsoft SQL Server

Microsoft SQL Server[3] is as an ideal backbone for enterprise-grade applications, offering a comprehensive suite of features that fit the needs of modern data processing, security, and integration: as a relational database management system (RDBMS), it provides a stable foundation with robust support for both Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) through advanced in-memory capabilities. Specifically, its In-Memory OLTP accelerates high-volume transactions by keeping track of frequently accessed tables, significantly reducing latency, while columnstore indexes further enhance performance by enabling efficient storage and retrieval of data for analytical workloads, allowing applications to perform complex queries on large datasets while maintaining fast response times. To improve query efficiency, SQL Server includes an Intelligent Query Processing (IQP) suite, which dynamically optimizes

[3] https://www.microsoft.com/it-it/sql-server/sql-server-2019

queries based on workload patterns. Features like table variable deferred compilation, batch mode on rowstore, and adaptive memory feedback allow the database engine to adjust to different workload conditions, reducing processing time and resource consumption. This is particularly beneficial in applications with fluctuating query demands, where SQL Server can adapt to provide consistent performance; moreover, the presence of special metadata (called memory-optimized tempdb) further boosts performance in scenarios with high levels of concurrency, minimizing bottlenecks during intense operations. Another standout feature of SQL Server is its Big Data Clusters, which enables the integration with Apache Spark[4] and Hadoop Distributed File System (HDFS)[5] to manage structured and unstructured data in a single platform, allowing organizations to store large amounts of unstructured data while simultaneously making it accessible for machine learning and analytics purposes. PolyBase, a complementary feature, extends such capabilities by enabling external queries without data movement, allowing SQL Server to interface with various sources, such as Oracle, MongoDB, and Teradata: this facilitates unified reporting and analytics across disparate systems, making it an ideal choice for applications that require efficient data handling across heterogeneous sources. Being security a major concern, especially in scenarios with sensitive and personal data, SQL Server offers features like Always Encrypted metadata, which enables data to remain encrypted throughout processing, allowing secure operations on such information; secure enclaves extend this capability by providing a secure area for decryption, permitting operations on encrypted data without exposing it. Transparent database encryption (TDE) safeguards data at rest by encrypting database files, while dynamic data masking and row-level security add layers of protection, controlling access to sensitive fields and specific rows based on user permissions. This strict level of security and privacy standards allows organizations to build effective and secure tools that achieve compliance with industry regulations (like GDPR in Europe, or HIPAA in the USA).

[4] https://spark.apache.org/

[5] https://hadoop.apache.org/

Building on the capabilities of Microsoft SQL Server, SQL Server Management Studio (SSMS) serves as a crucial interface for managing and optimizing database operations; this integrated environment brings together a range of tools for streamlined administration, enabling users to effectively monitor, tune, and manage SQL Server databases. Among them, we can find:

- Query Store, that allows the capture query performance history to identify bottlenecks and make data-driven decisions to enhance the overall responsiveness of the system;
- Activity Monitor, which provides a live overview of health and performance metrics, assisting in troubleshooting and ensuring optimal operation;
- Database Tuning Advisor, that analyzes usage patterns and provides recommendations for indexing and partitioning strategies, significantly enhancing query performance by aligning the database structure with workload demands

Lastly, SQL Server can also integrate with SQL Server Data Tools (SSDT) within Visual Studio, enabling developers to build, test, and deploy SQL Server applications seamlessly. This integration supports continuous integration and deployment (CI/CD) processes, promoting an agile development. By combining SQL Server's high-performance data processing, advanced security measures, and extensive integration options with the flexibility and modularity of frameworks like Spring Boot, SQL Server offers a scalable, secure, and efficient database solution that supports the stability and operational demands of modern enterprise applications.

3.2 System Architecture

To provide a clear overview of the system's architecture, this section schematically defines the roles and main key functions of each core module, describing how they collectively support procurement reporting and data management; each module has been assigned a number for reference, which is consistent with the system diagram below.

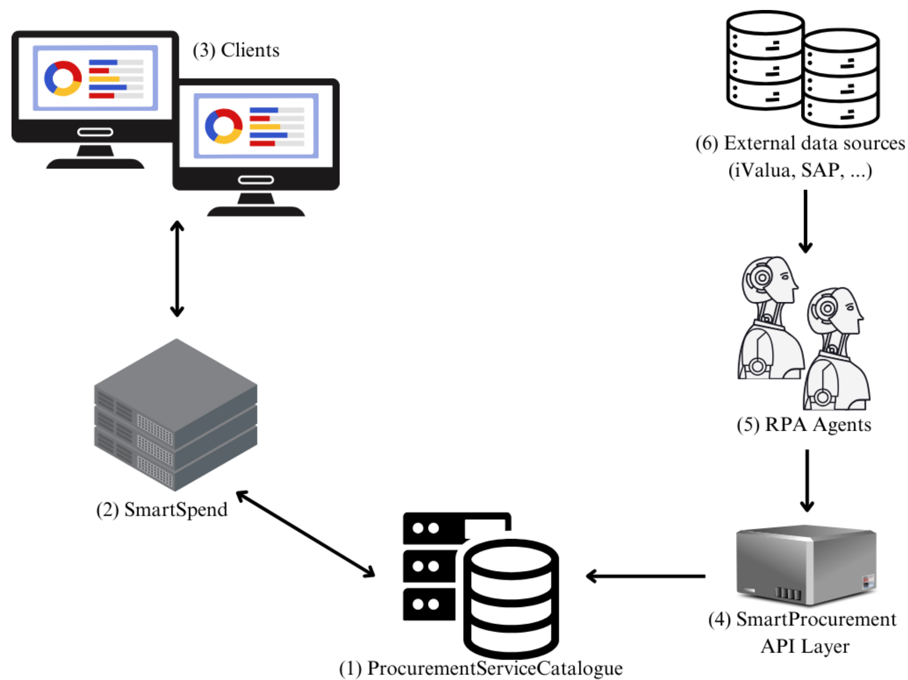

Figure 3.2. Overall architecture of the system

1. **ProcurementServiceCatalogue**

    - Centralized repository built on Microsoft SQL Server, serving as the foundation for all procurement-related data;
    - Employs a normalized schema to organize key entities such as suppliers, invoices, and purchase orders;
    - Includes optimized indexing and stored procedures to enable high-performance queries for complex, multi-join reporting;

- Provides SmartSpend with clean and verified data, making it the core source of structured insights for top-management decision-making;

2. **SmartSpend**

   - Primary web interface, developed with Spring Boot and Freemarker for dynamic dashboards and reports;
   - Presents data-driven insights to enable informed choices based on well structured procurement data;
   - Uses SAML2 for secure user authentication, integrating with almost any identity provider;
   - Protected by a Web Application Firewall (WAF) to secure internet exposure, minimizing external threats;
   - Deployed as a standalone Spring Boot application, with embedded server capabilities for consistent functionality across environments (development, test, production);

3. **Clients**

   - Represents end-users (primarily top-management), who access the SmartSpend application for procurement insights.
   - Connects via a secure, browser-based interface, allowing the interaction with intuitive dashboards and reports;
   - Enables data-driven decision-making by providing insights derived from the ProcurementServiceCatalogue;

4. **SmartProcurement API Layer**

   - Spring Boot application that facilitates communication between data sources and the ProcurementServiceCatalogue;
   - Exposes RESTful APIs for data imports, supporting both manual and automated processes;
   - Validates incoming data to align with the ProcurementServiceCatalogue schema and performs batch processing for large data volumes;

  - Supports data formats like JSON and Excel, allowing compatibility with various data extraction sources;
  - Enforces security through secret API keys for RPA agents, maintaining controlled access and ensuring data integrity;

5. **RPA Agents**
   - Bridges external data sources with the Smart Procurement API Layer by automating data extraction;
   - Standardizes extracted data into formats like Excel, facilitating smooth import processes into the ProcurementServiceCatalogue;
   - Communicates securely with the API Layer using secret API keys, ensuring secure data transfer;
   - Reduces manual intervention in data import processes, providing a consistent, automated feed of procurement data;

6. **External data sources**
   - Include primary procurement systems such as SAP and iValua, where raw data on suppliers, invoices, and purchase orders originates;
   - Serves as the starting point in the data pipeline, with extracted data passed through RPA agents and processed before integration;
   - Provides essential data that forms the basis of the systems' structured insights;

The deployment of this architecture is automated using Terraform[6], enabling an Infrastructure as Code (IaC) process to streamline the setup and management of resources. The entire system is hosted on Google Cloud (specifically, on two separate instances for the two Spring Boot applications), where it benefits from cloud-based scalability and resilience. Leveraging the available smart load balancers, the architecture supports optimized resource distribution, ensuring reliability even with high data volumes and concurrent access; this combination ensures a secure, flexible and high-performance environment that aligns with the overall requirements of the project, while keeping a good level of maintainability and ease of configuration.

[6] https://www.terraform.io/

3.3 SmartSpend: Cost Control module

Effective cost control plays a crucial role in procurement management, by allowing to identify and implement savings opportunities while maintaining financial oversight: to support this need, the Cost Control module of the SmartSpend web application was developed to provide a fundamental tool to analyze such initiatives, track their progress, and evaluate outcomes with precision. This section explores both the conceptual and technical foundations of the module: first, we will outline the processes of savings identification and cost control, emphasizing their importance in driving procurement efficiency and highlighting the workflows that support these objectives; we will then examine the technical implementation of the module, focusing on the dynamic Excel-like table at its core (built with the Handsontable JS[7] library) while also discussing key backend mechanisms, such as the storage of savings entities and edit entities in separate tables and how these are combined to enable the advanced formatting and functionality of the overall table.

3.3.1 Processes description

The process of savings identification within SmartSpend is the critical first step in an ongoing cycle of cost management. It ensures that potential opportunities are systematically identified, validated, and integrated into the company's broader cost control strategy. The diagram below outlines the overall process, detailing the roles of local managers and regional managers as they collaborate to capture and evaluate savings initiatives:

[7] https://handsontable.com/

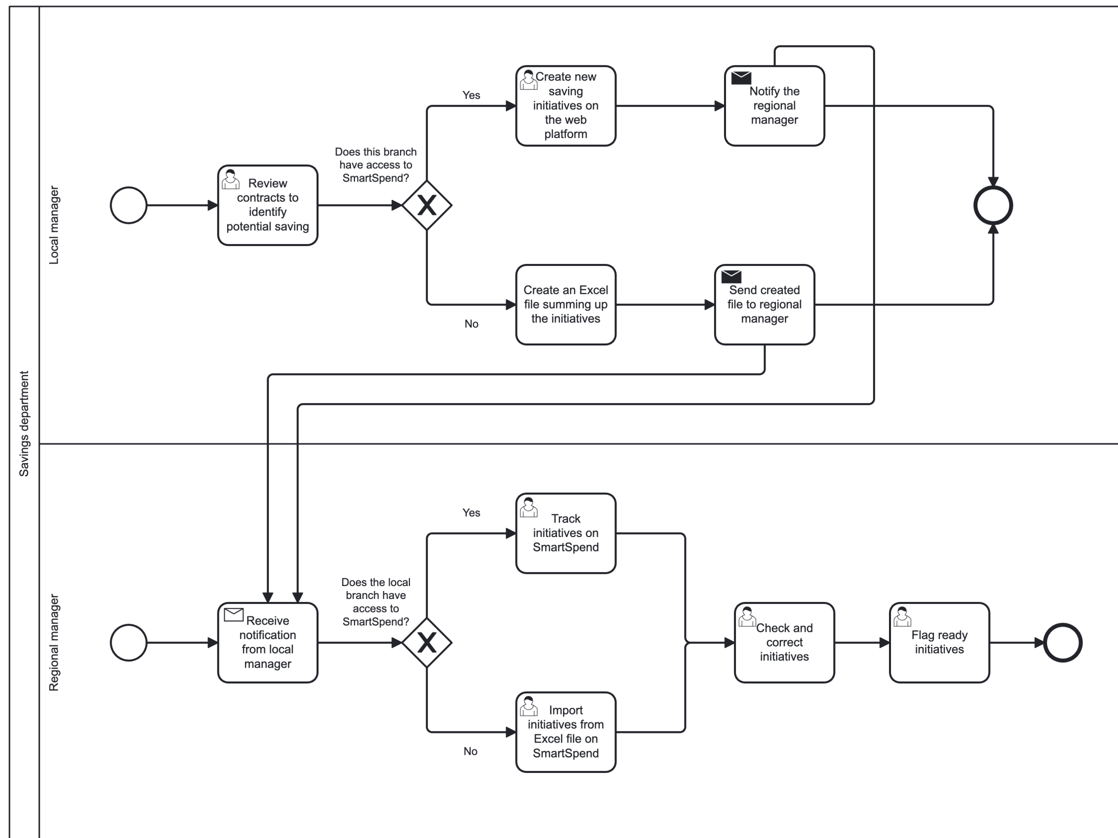

Figure 3.3. BPMN diagram of the savings identification process

The process begins with the local manager, who is responsible for reviewing the currently active contracts to identify potential savings: this step is vital for ensuring that cost reduction opportunities are spotted early, setting the stage for further analysis and action; once a saving is identified, the next steps depend on whether the local branch has access to SmartSpend.

For branches with access to the application, the local manager is able to directly creates new saving initiatives by using the appropriate form, allowing them to be stored in the ProcurementServiceCatalogue and available for further review; for other branches, the local manager manually creates an Excel file with the saving initiatives identified. In either case, the regional manager is emailed directly by each local counterpart to be notified of new initiatives that need attention and, in the second scenario, is in charge of manually uploading each row of the file received into the

platform. This sub-optimal workaround is necessary but temporary, as almost all branches are in the process of being enrolled into the system: the migration from a manual, Excel-based approach to a more modern, integrated solution is a key step of the platform's evolution, and the design of the frontend table, resembling the interface the users are accustomed to, was specifically chosen to facilitate a smoother transition; nonetheless, this duality of the process allows to accommodate varying levels of technological access, ensuring that all savings initiatives are captured and reviewed. The central savings department, lead by the regional manager, is in charge of the final validation, to ensure accuracy and compliance of the proposed initiatives with organizational guidelines; any necessary corrections are made, ensuring that only verified savings are flagged as ready for the next step. This workflow is an essential first step in managing and controlling costs across the organization: by systematically capturing savings opportunities and ensuring their accuracy and completeness, the process lays the groundwork for the next critical phase, and the transition from identifying savings to actively controlling costs is seamless, as validated savings are integrated into the broader cost management framework.

Following the identification of savings, the next phase involves cost control, where these identified savings are actively monitored and adjusted over time. The following BPMN diagram illustrates this process in detail:

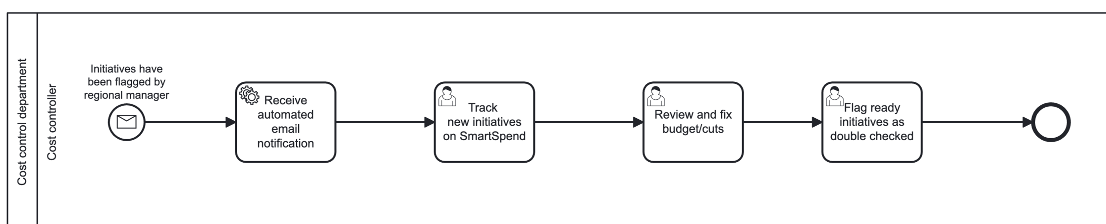

Figure 3.4. BPMN of the cost control process

While being quite linear, this process acts as a final gateway for actual initiatives implementation. By double checking the validation made from the regional manager, and setting an definitive threshold for avaliable budget and cuts to be made, it allows a finer tuning of savings strategies: centralizing these decisions under a single cost controller (one for

each country) ensures proper realization and easier manintainance. The new SmartSpend module significantly enhances the efficiency of these two processes by consolidating all relevant data into a unified interface: by integrating savings initiatives and cost control mechanisms within the same environment, it streamlines decision-making, reduces redundancies, and provides stakeholders with a comprehensive overview necessary for a higher operational effectiveness.

3.3.2 Implementation details

At the heart of the Cost Control Module is a dynamic, Excel-like table implemented using the Handsontable JavaScript library, mainly selected for its ability to replicate the intuitive and versatile experience of working with spreadsheets, making it more accessible to users already accustomed to such tools: this approach simplifies the transition from manual Excel workflows to the integrated SmartSpend platform, ensuring a smoother adjustment phase while benefiting from enhanced functionality and data integration. Handsontable offers a rich set of features that closely emulate Excel, contributing to its usability and flexibility:

- Resizable columns, to allow dynamic and tailored widths for better content visibility;
- Header filters, to narrow down visible data without needing additional measures (like pre-filtering the dataset from the backend, slowing down the initial page load);
- Collapsible columns, to temporarily add less relevant fields and keep the focus on specific sections;
- Custom hooks to manage cells, for example to highlight edited cells to provide immediate feedback;
- Strict validation, either via pre-defined validators for common types or by allowing custom ones, to accommodate more or less complex needs;
- Fine grained edit control, to prevent accidental edits by locking the entire table into read mode, and to protect important columns by keeping them read-only even in edit mode (based on user permissions);

- Custom and configurable smart pairs, to link columns and automatically propagate edits when necessary.

The following code snippet demonstrates how the overall table is instantiated, highlighting the advanced features mentioned above:

```
1   const container = document.getElementById('savingsTable');
2   const hot = new Handsontable(container, {
3     data: [], // Placeholder for dynamic data fetched from backend
4     colHeaders: ['Initiative Name', 'Department', 'Budget (VAT Included)', 'Budget (VAT Excluded)'],
5     columns: [
6       {
7         data: 'initiativeName',
8         type: 'text',
9         readOnly: false,
10        validator: function (value, callback) {
11          callback(!!value && value.trim().length > 0); // Must not be empty
12        }
13      },
```

Figure 3.5. Example instantiation of the Handsontable, to showcase the various features used, pt.1

```
14      {
15        data: 'department',
16        type: 'dropdown',
17        source: ['Finance', 'HR', 'Operations'],
18        allowEmpty: false
19      },
20      {
21        data: 'budgetWithVAT',
22        type: 'numeric',
23        format: '0,0.00',
24        allowEmpty: false
25      },
26      {
27        data: 'budget',
28        type: 'numeric',
29        format: '0,0.00',
30        allowEmpty: false
31      },
32    ],
33    stretchH: 'all', // Automatically adjust column widths to fit the container
34    filters: true, // Enable header filters
35    dropdownMenu: true, // Add filtering options in headers
36    collapsibleColumns: [1], // Enable column collapsing for the 'Department' column
37    manualColumnResize: true, // Allow columns to be resized
38    invalidCellClassName: 'htInvalid', // Apply class to invalid cells
39    cells: function (row, col, prop) {
40      const cellProperties = {};
41      const rowData = this.instance.getSourceDataAtRow(row);
42      if (rowData && rowData.edited === 1) {
43        cellProperties.className = 'edited-cell'; // Highlight edited rows with orange background
44      }
45      return cellProperties;
46    },
47  });
```

Figure 3.6. Example instantiation of the Handsontable, to showcase the various features used, pt.2

The table's configuration ensures that each column is directly linked to a field within the DTO returned by the backend, which represents the saving initiatives found: for example, the "*Initiative Name*" column corresponds to the *initiativeName* field, while the budget columns map to *budgetWithVAT* and *budget* (this is just a reduced example to give the idea of the overall functionality implemented, the actual code is quite a bit more complicated and extensive). Such direct mapping guarantees synchronization between the table content and the underlying model, streamlining data management, and, to maintain accuracy, validation rules can be applied to critical fields: initiative names for example are required to be non-empty, while department selections must adhere to a predefined set of valid options. Additionally, dynamic styling is used to highlight rows where the

edited field in the DTO is set to 1 (true), visually signaling modified entries with an orange background. To complement this design, the table alternates between two operational modes, which are toggled programmatically through button click events: by default, the table is accessed as immutable, allowing users to freely consult its content while protecting from accidental modifications; by clicking on a specific button, the table switches to its editable version, allowing updates to all the columns that aren't statically marked as read-only through the apposite attribute. Once edits are finalized, they are submitted to the backend via another button click and the table reinstates its safeguards. The following snippet illustrates how these transitions are achieved:

```
1  // Enable edit mode
2  document.getElementById('edit_mode').addEventListener('click', function() {
3    hot.updateSettings({ readOnly: false });
4    document.getElementById('edit_mode').style.display = 'none'; // Hide edit button
5    document.getElementById('save_button').style.display = 'inline'; // Show save button
6  });
7
8  // Save edits and disable edit mode
9  document.getElementById('save_button').addEventListener('click', function() {
10   const updatedData = hot.getData();
11   fetch('/save-edits', {
12     method: 'POST',
13     headers: { 'Content-Type': 'application/json' },
14     body: JSON.stringify({ updatedData }),
15   })
16   .then(response => {
17     if (response.ok) {
18       hot.updateSettings({ readOnly: true });
19       alert('Changes saved successfully!');
20     } else {
21       alert('Failed to save changes.');
22     }
23     document.getElementById('edit_mode').style.display = 'inline'; // Show edit button
24     document.getElementById('save_button').style.display = 'none'; // Hide save button
25   })
26   .catch(error => console.error('Error:', error));
27 });
```

Figure 3.7. Example events to toggle modes in the table and persist edits

Another essential feature of the table, which was mentioned before and can enabled progammatically, is its support for linked columns, or smart pairs, which ensures consistency between related fields. For example, the "Budget (VAT Included)" and "Budget (VAT Excluded)" columns are dynamically connected: when a user updates one of these fields, the other is automatically recalculated based on a predefined VAT rate. This behavior

is achieved through Handsontable's afterChange hook, as demonstrated in the following code block:

```
1  hot.addHook('afterChange', function(changes, source) {
2    if (source === 'edit') {
3      changes.forEach(([row, prop, oldValue, newValue]) => {
4        const vatRate = 0.2; // Example VAT rate of 20%, but can be retrieved from BE based on country
5        if (prop === 'budgetWithVAT') {
6          const withoutVAT = newValue / (1 + vatRate);
7          hot.setDataAtCell(row, 3, withoutVAT.toFixed(2)); // Update 'Budget Without VAT'
8        } else if (prop === 'budget') {
9          const withVAT = newValue * (1 + vatRate);
10         hot.setDataAtCell(row, 2, withVAT.toFixed(2)); // Update 'Budget With VAT'
11       }
12     });
13   }
14 });
```

Figure 3.8. Custom hook to manage automatic VAT calculation

Beyond the interactive front-end of the table, its seamless integration with the back-end is crucial to ensure consistency and traceability: the architecture is designed to complement visual features by providing efficient data retrieval, robust filtering, and persistent update storage employing a very classical Controller-Service-Repository structure, with the first charged with properly routing incoming requests, the second containing all business logic (including proper data filtering based on users' geographical and team permissions) and the latter managing all interactions with the underlying database. To go into further details, there are actually two repositories that contribute to all data interactions, properly reflecting the data model supporting the whole module: structured to facilitate both real-time interactions and historical tracking, each saving initiative is represented in the database by a **Saving** entity, while modifications are captured through the **Edit** entity, which maintains a many-to-one relationship with the **Saving**.

The following snippets give an idea about how these entities are structured:

```
1  @Entity
2  public class Saving {
3      @Id
4      @GeneratedValue(strategy = GenerationType.IDENTITY)
5      private Long id;
6
7      private String initiativeName;
8      private String department;
9      private Double budgetWithVAT;
10     private Double budget;
11
12     @OneToMany(mappedBy = "saving", cascade = CascadeType.ALL, orphanRemoval = true)
13     private List<Edit> edits = new ArrayList<>();
14
15     // Getters and setters
16 }
```

Figure 3.9. Structure of the **Saving** entity

```
1  @Entity
2  public class Edit {
3      @Id
4      @GeneratedValue(strategy = GenerationType.IDENTITY)
5      private Long id;
6
7      private String fieldEdited;
8      private String previousValue;
9      private String newValue;
10     private Integer userId;
11     private LocalDateTime timestamp;
12
13     @ManyToOne
14     @JoinColumn(name = "saving_id")
15     private Saving saving;
16
17     // Getters and setters
18 }
```

Figure 3.10. Structure of the **Edit** entity

This structure not only enables efficient data retrieval, but also allows each row in the table to include a "*View older edits*" button: when clicked, it opens a modal displaying a detailed history of saved edits for the selected saving initiative, providing users with a full audit trail of changes stored in the database and effectively complementing the orange-highlighted cells that indicate unsaved modifications directly within the table. Together, these elements ensure the Cost Control module provides both real-time interactivity and comprehensive traceability, meeting the demands of accurate and transparent savings management.

3.4 API Layer: RPA agents integration

A robust and efficient data import process is vital for maintaining the integrity and accuracy of the centralized ProcurementServiceCatalogue database. This section details the development of a new API designed to streamline the import of invoice data: once the RPA agents extract the necessary information from the external sources, they structure the data into a static Excel file and send it to the newly created endpoint, providing additional details such as the invoice reference date and the country of purchase. The system then processes these files using the Apache POI[8] library, efficiently reading and transforming each row into a Spring Boot entity for storage. We will also address some challenges encountered during the initial implementation and present a refined asynchronous approach to this task, developed to improve both flexibility and efficiency while handling large-scale imports.

3.4.1 API implementation, InvoiceController and InvoiceService

Apache POI (Poor Obfuscation Implementation) is a robust, open-source Java library that enables reading and writing of Microsoft Office files, particularly Excel files (.xls and .xlsx). It allows Java applications to interact with both the older binary Excel format (HSSF) and the XML-based format (XSSF), providing useful methods for parsing, manipulating, and extracting data from Excel cells, rows, and sheets; in out context, it enables us to parse the structured data provided by the robots, validate it, and convert each row into a corresponding entity ready for database storage. The new API, created within the **InvoiceController**, receives the Excel file directly from the RPA agents together with the date of extraction and the country where the invoices were generated; these parameters help maintain data consistency and support downstream reporting and analysis. The *uploadInvoiceData* method associated with the */import* endpoint enforces strict validation to ensure data integrity: all input parameters are required, with annotations enforcing these constraints at the parameter level (**@NotNull** for the file and **@NotBlank** for extraction date

[8] https://poi.apache.org/

and country), preventing incomplete data submissions and ensuring that essential information is always provided. Upon receiving the request, the controller first checks if the file is empty; if it is, a **BAD_REQUEST** response is returned, notifying the client (in this case, the RPA agents) of the missing file. Otherwise, the main processing of the file is delegated to the **InvoiceService**, and the correct HTTP status code is returned to notify the final result of the computation (**INTERNAL_SERVER_ERROR** in case of an exception occoured during the import, or **OK** if the operation was successful).

The following code snippet illustrates the setup of **InvoiceController**, showcasing the design and initial validation measures:

```
1  @RestController
2  @RequestMapping("/api/spend")
3  public class InvoiceController {
4
5      @Autowired
6      private InvoiceService invoiceService;
7
8      @PostMapping("/import")
9      public ResponseEntity<String> uploadInvoiceData(
10             @RequestParam("file") @NotNull MultipartFile file,
11             @RequestParam("extractionDate") @NotBlank String extractionDate,
12             @RequestParam("country") @NotBlank String country) {
13
14         if (file.isEmpty()) {
15             return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Provided empty file!");
16         }
17
18         try {
19             invoiceService.processInvoiceFile(file, extractionDate, country);
20             return ResponseEntity.ok("File uploaded and processed successfully.");
21         } catch (Exception e) {
22             return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
23                                      .body("Error processing file: " + e.getMessage());
24         }
25     }
26 }
```

Figure 3.11. Initial implementation of the **InvoiceController**

The main processing phase of the data import is carried out by the **InvoiceService**. This class ensures that the data adheres to a strict format and is consistently processed, validated, and stored in the ProcurementServiceCatalogue database: by implementing rigorous validation mechanisms, logging key events, and summarizing the results, it guarantees data integrity while providing actionable feedback on the whole process. At the core of this service there is the *processInvoiceFile* method, which orchestrates the entire workflow. Starting with header validation, it ensures that the file is coherent with the predefined format expected by the system; each data row is then parsed, keeping track of invalid rows and converting valid rows into **Invoice** entities for later persistence. A structured summary of the process is returned at the end by leveraging the **ImportResponse** class, detailing the results of the operation. Below we can find a summarized implementation of this method:

```
1  @Service
2  public class InvoiceService {
3      private static final Logger log = LoggerFactory.getLogger(InvoiceService.class);
4
5      public ImportResponse processInvoiceFile(MultipartFile file,
6          String extractionDate,
7          String country) throws Exception {
8
9          List<Invoice> invoices = new ArrayList<>();
10         List<Integer> discardedRows = new ArrayList<>();
11         try {
12             InputStream inputStream = file.getInputStream();
13             Workbook workbook = WorkbookFactory.create(inputStream);
14             Sheet sheet = workbook.getSheetAt(0);
15             Iterator<Row> rowIterator = sheet.rowIterator();
16             log.info("Number of rows (header excluded): " + (sheet.getPhysicalNumberOfRows() - 1));
17
18             if (rowIterator.hasNext()) {
19                 Row headerRow = rowIterator.next();
20                 validateHeaderRow(sheet, headerRow);
21             } else {
22                 throw new IllegalArgumentException("Invalid Excel file: no data found.");
23             }
24
25             int rowIndex = 1; // Header row already parsed
26             while (rowIterator.hasNext()) {
27                 Row row = rowIterator.next();
28                 boolean isRowValid = true;
29
30                 String invoiceId = getCellValueAsString(row.getCell(InvoiceImport.INVOICE_ID.getColumnIndex()));
31                 if (invoiceId == null) {
```

Figure 3.12. Implementation of the *processInvoiceFile* method

```
32                      log.error("Row " + rowIndex + ": Missing mandatory field 'invoiceId'.");
33                      discardedRows.add(rowIndex);
34                      isRowValid = false;
35                      continue;
36                  }
37
38                  // Other mandatory fields validation...
39
40                  String purchaseNotes = getCellValueAsString(row.getCell(InvoiceImport.PURCHASE_NOTES.getColumnIndex()));
41                  if (purchaseNotes == null) {
42                      log.warn("Row " + rowIndex + ": Missing optional field "
43                          + InvoiceImport.PURCHASE_NOTES.getColumnName() ".");
44                  }
45
46                  if (isRowValid) {
47                      Invoice invoice = new Invoice(invoiceId, ..., purchaseNotes);
48                      invoices.add(invoice);
49                      log.info("Row " + rowIndex + ": Successfully converted to Invoice entity.");
50                  }
51                  rowIndex++;
52              }
53
54              saveInvoices(invoices);
55              log.info("Successfully saved " + invoices.size() + " invoices.");
56              return new ImportResponse(sheet.getLastRowNum(), invoices.size(), discardedRows.size(), discardedRows);
57          } catch (Exception e) {
58              log.error("Error processing Excel file: " + e.getMessage(), e);
59              throw new Exception("Error processing Excel file: " + e.getMessage(), e);
60          }
61      }
62  }
```

Figure 3.13. Implementation of the *processInvoiceFile* method (cont.)

The flow begin begins by validating the file's structure, particularly its header row: by leveraging the **InvoiceImport** enum, which defines the expected Excel schema, this step ensures that the file contains the correct columns, in the correct order, and with the expected names. Each column is identified by its name and position, as shown in the following snippet:

```java
public enum InvoiceImport {
    INVOICE_ID("invoiceId", 0),
    INVOICE_DATE("invoiceDate", 1),
    SUPPLIER_NAME("supplierName", 2),
    INVOICE_AMOUNT_VAT_INCL("invoiceAmountVatIncl", 3),
    INVOICE_AMOUNT_VAT_EXCL("invoiceAmountVatExcl", 4),
    CURRENCY("currency", 5),
    PURCHASE_NOTES("purchaseNotes", 6);

    private final String columnName;
    private final int columnIndex;

    InvoiceImport(String columnName, int columnIndex) {
        this.columnName = columnName;
        this.columnIndex = columnIndex;
    }
    //getters
}
```

Figure 3.14. Structure of the **InvoiceImport** enum

To validate the header, the *validateHeaderRow* method checks both the column count and their exact names and positions. If the header does not match the schema, an exception is raised, preventing further processing, otherwise the import flow is able to resume. This ensures that only properly formatted files are processed:

```java
private void validateHeaderRow(Sheet sheet, Row headerRow) {
    if (headerRow.getPhysicalNumberOfCells() != InvoiceImport.values().length) {
        throw new IllegalArgumentException("Invalid Excel format: unexpected number of columns.");
    }

    for (InvoiceImport column : InvoiceImport.values()) {
        Cell cell = headerRow.getCell(column.getColumnIndex());
        if (cell == null || !column.getColumnName().equalsIgnoreCase(cell.getStringCellValue().trim())) {
            throw new IllegalArgumentException("Invalid Excel format: expected column "
                    + column.getColumnName() + " at position " + column.getColumnIndex()
                    + ", but found " + (cell != null ? cell.getStringCellValue() : "null"));
        }
    }
    log.info("Header row validation successful.");
}
```

Figure 3.15. Implementation of the *validateHeaderRow* method

Once the header is validated, the method iterates through each row, extracting individual fields. If any required one (in this case intended as per buisness logic) is missing, the row is discarded, logged as an error, and its index is added to a list of discarded rows for reporting; on the other hand, the optional ones, like the purchase notes in the example, do not invalidate

the row if missing, but are logged as warnings to aid in troubleshooting. Successfully validated rows are converted into **Invoice** entities and logged with an informational message. At the end of the process, all valid entities are saved, and the method returns an **ImportResponse** object summarizing the operation: by including key information like the total number of rows processed, the number of successfully saved rows, the count of discarded rows, and the indices of invalid rows, it effectively provides a useful output for the overall flow.

```
1  public class ImportResponse {
2      private int totalRows;
3      private int savedRows;
4      private int errorRows;
5      private List<Integer> discardedRowIndices;
6
7      public ImportResponse(int totalRows, int savedRows, int errorRows, List<Integer> discardedRowIndices) {
8          this.totalRows = totalRows;
9          this.savedRows = savedRows;
10         this.errorRows = errorRows;
11         this.discardedRowIndices = discardedRowIndices;
12     }
13
14     // Getters and setters
15 }
```

Figure 3.16. Structure of the **ImportResponse** class

3.4.2 Async API for RPA agents integration

The initial implementation analyzed above, while functional, revealed significant limitations when handling larger datasets: on average, the files processed contained quite a few thousands rows (starting from 10000 and going up to 50000+), requiring considerable time for parsing, validating, and saving the successfully converted data. Although the server was able to manage the computation in a few minutes (up to 15 for larger files), the firewall governing the network used by the RPA agents enforced a strict 90-seconds timeout on active connections: this constraint often translated into errors, even when the server-side process had completed successfully. As a result, the agents were unable to confirm whether the operation had succeeded, introducing ambiguity and inefficiencies into the workflow. One potential solution considered was splitting large files into smaller ones, each containing fewer rows, and processing these in separate

and consecutive API calls; however, determining an optimal and universal file size proved challenging due to variables such as concurrent API calls and fluctuating server loads. Moreover, requiring the RPA agents to manage file splitting and multiple uploads would have added complexity to their development, making the approach impractical.

The adopted solution instead focused on making the whole process asynchronous: by spawning a new thread on the server to handle the import, this approach decouples the API call from the actual file processing. The request is now acknowledged immediately, providing the RPA agents with a confirmation that the operation has started successfully (or with a proper error message if that's not the case), and, to allow agents to monitor the progress of the import, an additional polling mechanism was introduced, enabling them to query the server for status updates via a different and new endpoint. So, while eliminating timeout errors, this design still allows to maintain a robust and user-friendly workflow. To understand the technical differences introduced by this asynchronous model, we first examine the changes made to the **InvoiceController**. Below is the updated implementation:

```
1  @RestController
2  @RequestMapping("/api/spend")
3  public class InvoiceController {
4
5      @Autowired
6      private InvoiceService invoiceService;
7      @Autowired
8      private PollService pollService;
9
10     @PostMapping("/import")
11     public ResponseEntity<?> uploadInvoiceData(
12             @RequestParam("file") @NotNull MultipartFile file,
13             @RequestParam("extractionDate") @NotBlank String extractionDate,
14             @RequestParam("country") @NotBlank String country) {
15
16         if (file.isEmpty()) {
17             return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Provided empty file!");
18         }
19
20         PollStatus poll = pollService.initPoll(LocalDateTime.now(), file.getOriginalFilename(), "Invoice");
21         if (poll == null) {
22             return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Error initializing poll object.");
23         }
24
25         invoiceService.processInvoiceFile(file, extractionDate, country, poll);
26         return ResponseEntity.ok(poll.getId());
27     }
28 }
```

Figure 3.17. Async implementation of the **InvoiceController**

This updated controller reflects the asynchronous design by introducing several key changes. Unlike the previous synchronous implementation, the *uploadInvoiceData* method now initializes a **PollStatus** object via the **PollService**: this allows to track the progress of the import operation, storing metadata such as the start time, the file name, and the type of import. The identifier of the object is then returned as part of the response, allowing the RPA agents monitor the process through a separate polling mechanism. Like before, the validation and processing phase is delegated to the **InvoiceService**, by calling the updated version of the *processInvoiceFile* method, which now operates in a separate thread. Additionally, if the initialization of the **PollStatus** fails, an **INTERNAL_SERVER_ERROR** is returned, signaling a critical issue on the server side. For completeness, here is a definition of the newly introduced polling entity, which will be useful also to explain some new additions to the **InvoiceService**:

```
1  @Entity
2  @Table(name = "poll", schema = "[config]")
3  public class PollStatus {
4      @Id
5      @GeneratedValue()
6      @Type(type = "uuid-char")
7      @Column(name = "id", columnDefinition = "uniqueidentifier")
8      private UUID id;
9      @Column(name = "imported_entity")
10     public String importedEntity;
11     @Column(name = "start_timestamp", columnDefinition = "TIMESTAMP")
12     @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd HH:mm:ss")
13     private LocalDateTime startTimestamp;
14     @Column(name = "completed_percentage")
15     public Integer percentageStatus;
16     @Column(name = "serialized_import_response") // populated only if operation is finished (either correctly or not)
17     public String serializedImportResponse;
18     @Column(name = "filename")
19     public String filename;
20
21     //constructors, getters, setters, ...
22 }
```

Figure 3.18. Structure of the **PollStatus** entity

As we can see, this class is used to map rows from the polling table in the ProcurementServiceCatalogue, which are often updated during the parsing phase of the original Excel file in order to reflect the completion percentage of the operation; RPA agents can retrieve this information simply by calling the endpoint */getPollStatus* exposed by the **PollController** (not included for briefness) providing the previously returned identifier,

effectively retrieving useful information about the whole status of the process.

Moving to the **InvoiceService**, the updated implementation of the *processInvoiceFile* method includes several key enhancements. Firstly, the use of the **@Async** annotation enables the execution in a separate thread, allowing the API to acknowledge the operation immediately while continuing to process the file in the background. The method is now declared as *void* and no longer returns any value or propagates exceptions: instead, errors and results are logged and encapsulated within the enhanced **ImportResponse** object (it now includes a String field for detailed feedback, improving the clarity of error messages and process summaries), which is serialized and stored in the specific **PollStatus** field. This updated design allows for better traceability and eliminates the need for immediate responses, addressing the previous timeout issues:

```
@Service
public class InvoiceService {
    private static final Logger log = LoggerFactory.getLogger(InvoiceService.class);
    @Autowired
    PollService pollService;

    @Async
    public void processInvoiceFile(MultipartFile file, String extractionDate, String country) throws Exception {
        List<Invoice> invoices = new ArrayList<>();
        List<Integer> discardedRows = new ArrayList<>();

        try {
            InputStream inputStream = file.getInputStream();
            Workbook workbook = WorkbookFactory.create(inputStream);
            Sheet sheet = workbook.getSheetAt(0);
            Integer totalRows = sheet.getPhysicalNumberOfRows();
            Iterator<Row> rowIterator = sheet.rowIterator();
            log.info("Number of rows (header excluded): " + ( totalRows- 1));

            if (rowIterator.hasNext() || totalRows > 1) {
                Row headerRow = rowIterator.next();
                validateHeaderRow(sheet, headerRow);
            } else {
                ImportResponse response = new ImportResponse(0, 0, 0, Collections.EMPTY_LIST, "File is empty.");
                pollService.updatePollStatus(poll, 0, JsonUtils.toJson(response));
            }

            int rowIndex = 1; // Header row already parsed
            List<Integer> stepsForPercUpdate = calculateStepsForPercUpdate(sheet.getPhysicalNumberOfRows());
            while (rowIterator.hasNext()) {
                Row row = rowIterator.next();
                boolean isRowValid = true;
```

Figure 3.19. Implementation of the updated *processInvoiceFile* method

```
33
34                  //parse all fields as before...
35
36                  if (stepsForPercUpdate.contains(rowIndex)) {
37                      int percentage = (stepsForPercUpdate.indexOf(rowIndex) + 1) * 5;
38                      pollService.updatePollStatus(poll, percentage, "");
39                  }
40                  rowIndex++;
41              }
42
43              try {
44                  saveInvoices(invoices);
45                  String responseMsg = String.format("Successfully saved %d invoices.", invoices.size());
46                  ImportResponse response = new ImportResponse(totalRows, invoices.size(),
47                          totalRows, discardedRows, responseMsg);
48                  pollService.updatePollStatus(poll, 100, JsonUtils.toJson(response));
49              } catch (Exception e) {
50                  String errMsg = String.format("Error saving invoices: %s", e.getMessage());
51                  log.error(errMsg, e);
52                  ImportResponse response = new ImportResponse(totalRows, 0, totalRows,
53                  discardedRows, "DB error, save failed.");
54                  pollService.updatePollStatus(poll, 99, JsonUtils.toJson(response));
55              }
56          } catch (Exception e) {
57              String generalErrMsg = String.format("Error processing Excel file: %s", e.getMessage());
58              log.error(generalErrMsg, e);
59              ImportResponse response = new ImportResponse(0, 0, 0, Collections.EMPTY_LIST, generalErrMsg);
60              pollService.updatePollStatus(poll, 0, "Error processing Excel file.");
61          }
62      }
```

Figure 3.20. Implementation of the updated *processInvoiceFile* method, cont.

The handling of the **PollStatus** object has a very important role in this asynchronous design. When the process begins, a new entry is created and initialized to track the operation's progress. The state of this object is updated at key points during execution, such as upon unsuccessful validation of the file's structure, at regular intervals during row processing, and after correct or failed save into the database. These updates include the completion percentage, interim messages, and, upon conclusion, a final summary of the operation's outcome. By disassociating the operation's state from the API response, the design ensures that RPA agents can query the progress through dedicated endpoints without relying on synchronous interactions.

To manage percentage updates efficiently, a utility method is used to calculate the milestones at which progress updates are triggered. The method divides the total number of rows into evenly spaced intervals, enabling consistent and predictable updates:

```
1 private List<Integer> calculateStepsForPercUpdate(int totalRows) {
2     int stepPercentage = 5;
3     int numberOfSteps = 100 / stepPercentage;
4     return IntStream.range(1, numberOfSteps)
5             .map(ind -> ind * (totalRows / numberOfSteps)).boxed().toList();
6 }
```

Figure 3.21. Implementation of the *calculateStepsForPercentage-Update* utility method

This logic ensures that updates occur only at designated points, minimizing redundant computations while providing sufficient granularity for near real-time progress tracking; during the row-processing loop, the current row index is checked against these milestones, and the percentage is updated accordingly. The **PollService** is responsible for persisting these updates, ensuring that the the operation's current state is promptly refreshed.

This updated approach not only resolves the limitations of the previous synchronous implementation, but also introduces a flexible and scalable solution tailored to handle the complexities of large data imports: by decoupling the central logic, ensuring timely progress updates and providing detailed feedback (either through the **PollStatus** object or via detailed logs), the system delivers both reliability and clarity. These changes make the integration process seamless for RPA agents, while the granular tracking and robust error handling ensure a transparent and efficient workflow, that aligns effectively with the goals of the application.

3.5 PSC: SVR dashboard analysis and performance boost

The **Sustainability Vendor Rating** (from here on, **SVR**) page is a central feature of the SmartSpend application, developed to enable the company to monitor how well its suppliers align with its core social and environmental values. By consolidating procurement data with sustainability metrics, it is able to provide actionable insights for fostering a responsible and sustainable supply chain. This section is divided into two parts: in the first one, we will present an overview of the SVR dashboard, describing its functionality and how it integrates data to assess suppliers' adherence to sustainability principles; the second one, on the other hand, will focus on the performance challenges encountered during its development, particularly due to the large datasets involved, and the optimization measures implemented to enhance the overall responsiveness of the page, drastically improving the final user experience.

3.5.1 Dashboard overview

The SVR dashboard was designed to align procurement practices with the company's commitment to corporate social responsibility, thus providing a comprehensive tool for evaluating how closely suppliers adhere to key **E**nvironmental, **S**ocial, and **G**overnance (**ESG**) principles. These assessments are vital to ensure that main business partners share values such as environmental stewardship, ethical labor practices, and strong corporate governance, reinforcing its broader commitment to sustainability and responsible practices. Evaluation is performed through the IntegrityNext[9] platform, where, upon supplier registration, a detailed ESG survey is filled out at the start of every new provisioning contract. This procedure must be completed within a predefined timeframe, and the results are used to generate scores for each supplier: an overall rating is assigned, together with category-specific ones like environmental protection, human rights and labor, health and safety, and supply chain responsibility.

The assessment data is retrieved from IntegrityNext and stored in the

[9] https://www.integritynext.com/

ProcurementServiceCatalogue in order to be combined with the procurement information contained in the Invoice table (populated with the API analyzed in the previous section), the foundation for spend analysis. By joining these two datasets (supplier information extracted from the invoices and the evaluation results imported), the dashboard bridges financial and sustainability data, enabling decisions that are both economically and ethically aligned with long term company's goals.

The interface combines clarity with interactivity, ensuring users can easily interpret and act on complex datasets. A donut chart on the left provides a high-level overview of spend distribution by suppliers' overall ESG scores; suppliers are categorized into five performance groups: green for those meeting sustainability standards, yellow for sufficient performance, red for non-compliant vendors, light gray for suppliers with pending surveys, and dark gray for those who have not yet completed the registration process on the dedicated platform. To complement this first diagram, a bar graph on the right breaks down spend data by the four ESG categories: this visualization allows users to identify strengths and weaknesses in supplier performance, such as suppliers excelling in environmental protection but underachieving in human rights or supply chain responsibility.Beneath these visualizations, an interactive table provides granular details about suppliers: selecting a segment of the donut chart or a bar in the graph dynamically loads specific supplier attributes, such as their name, code, total invoiced amount, and individual ESG scores; this layered approach allows to achieve both an at-a-glance summary and the ability to drill down into more specific details, if needed. By consolidating procurement and ESG metrics into a cohesive, visually compelling format, the SVR dashboard enhances the company's ability to align its supply chain operations with its corporate social responsibility objectives.

The mockup below is close to the actual developed tool, and, while lacking some details, it effectively illustrates the overall layout of this indispensable page:

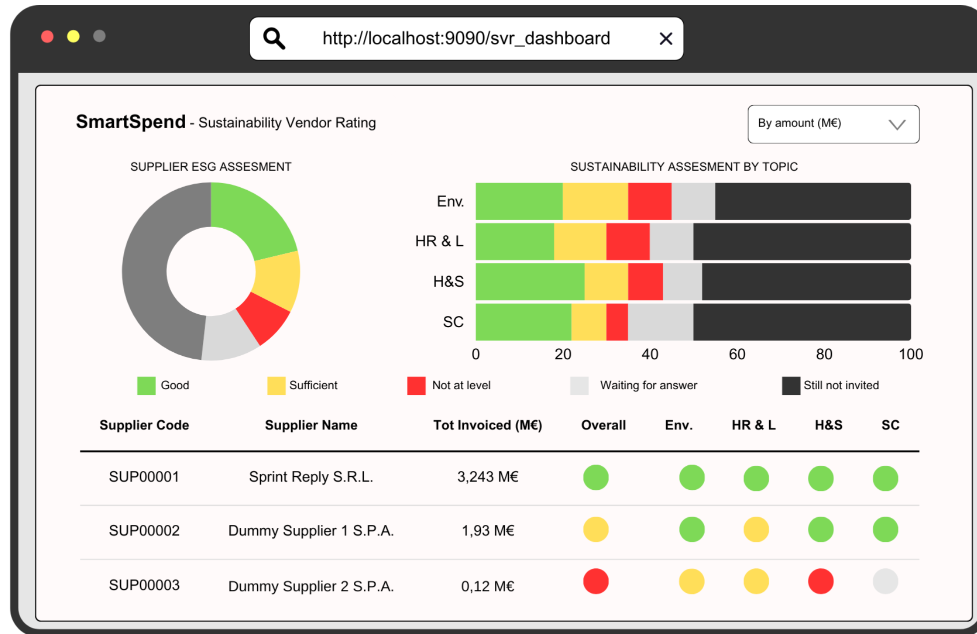

Figure 3.22. Mockup of the SVR dashboard

3.5.2 Performance analysis and improvement

The extensive size of the Invoice table constituted a severe bottleneck for the overall dashboard's performance, particularly affecting its responsiveness and overall loading times. Storing over 4.7 million rows at the time of development (Apr-May 2024), each one spanning over 67 columns, this table stores essential procurement data (invoice dates, item descriptions, amount spent, and so on) for around 76.000 suppliers. While indispensable for analysis, the sheer volume of the data makes queries computationally expensive: as mentioned in the previous section, the construction of the graphs requires multi-level aggregations and joins, resulting in significant delays and negatively impact the user experience; addressing these issues has been crucial to ensure the dashboard's continued usability and scalability.

Two potential solutions were evaluated to mitigate the problem identified. The first involved SQL Server's indexed views, which materialize query results by persisting them as a physical structure for efficient retrieval; this is particularly advantageous when queries involve straightforward aggregations or joins and when real-time synchronization with the underlying data is required: by automatically updating whenever base data changes, they ensure that query results remain current without the need of manual intervention. However, their strict schema constraints and limitations (such as restrictions on non-deterministic functions and lack of support for certain types of joins, like left joins) made them unsuitable for the SVR dashboard's complex aggregation logic. Additionally, maintaining real-time updates introduced unnecessary overhead, since the Invoice table data changes infrequently, with updates occurring only during quarterly or semi-annually imports (the timeframe changes based on the branch location, each geographical area has its own). The second solution focused on creating a dedicated summary table, here called **SupplierMonthly-Summary**, designed specifically for the dashboard's use case and populated with pre-aggregated data (such as total supplier spend grouped by month, along with fields such as supplier identifiers, iValua codes, and reference periods) in order to remove a big part of the computation from the load phase. Moreover, a dedicated stored procedure was developed to manage the content of this table, practically deleting all the rows and then repopulating it with freshly aggregated data derived from the Invoice table, aligning with the periodic data import cycle. Both the Invoice load process (from the RPA agents) and the stored procedure (from the ProcurementServiceCatalogue) were thought to be executed outside business hours, ensuring no interference with the company's operational activities: this deliberate scheduling eliminates the risk of performance bottlenecks or even failures during peek work hours, and ensures that the newly imported data is immediately available in the summary table for dashboard creation.

After comparing the two approaches, the second one clearly emerged as the optimal solution: indexed views, being effective for scenarios requiring real-time synchronization, were unnecessary for the infrequent update schedule of the Invoice table. The flexibility of the stored procedure allowed for the implementation of sophisticated aggregation and filtering

logic beyond the capabilities of indexed views: by isolating computationally intensive tasks to controlled, off-peak periods, the summary table provides a sustainable solution that meets both performance and operational requirements.

The following code snippet illustrates the **SupplierMonthlySummary** entity in Spring Boot, mapping the structure of the summary table:

```
1  @Entity
2  @Table(name = "SupplierMonthlySummary", schema = ["summary"])
3  public class SupplierMonthlySummary {
4      @Id
5      @GeneratedValue(strategy = GenerationType.IDENTITY)
6      private Long id;
7      @Column(name = "supplier_id")
8      private String supplierId;
9      @Column(name = "ivalua_code")
10     private String ivaluaCode;
11     @Column(name = "monthly_spend")
12     private Double monthlySpend;
13     @Column(name = "reference_year")
14     private Integer referenceYear;
15     @Column(name = "reference_month")
16     private Integer referenceMonth;
17     @Column(name = "spend_category_1")
18     private String spendCategory1;
19     @Column(name = "spend_category_2")
20     private String spendCategory2;
21     @Column(name = "geographical_area")
22     private String geoArea;
23
24     //getters and setter
25 }
```

Figure 3.23. Structure of the **SupplierMonthlySummary**

By implementing this table table and its associated maintenance process, even with the trade-off of occupying some more space into the database, the dashboard has received a significant performance boost: as we will see in the upcoming results chapter, query execution times are now reduced to a matter of very few seconds, ensuring that the dashboard now handles increasing data volumes while maintaining efficiency and delivering a pleasant user experience in the mean time.

Chapter 4

Results

In this section, we will present the results of the developments discussed in the previous chapters, highlighting their impact on the SmartSpend application and its role in modernizing procurement reporting. The outcomes of the work carried out can be grouped into two categories: qualitative results, as in the case of the Cost Control module, and quantitative results, which pertain to the enhancements of the SVR dashboard and the asynchronous API implemented for the Invoice import; this distinction reflects the nature of the data available for evaluation and the specific characteristics of each module's contributions.

The introduction of the Cost Control module represents a decisive advancement in how savings initiatives are monitored and managed across the organization. Previously, these processes were characterized by a very fragmented and largely manual nature: local branches relied heavily on Excel files to document savings initiatives, which were then emailed to regional managers for review; this approach, while functional, introduced inefficiencies such as inconsistent formatting and potential errors during manual data consolidation. The newly implemented module transforms this landscape by providing an online, centralized, and always-available tool that streamlines savings management. Fully integrated into the SmartSpend application, it delivers a unified interface that caters to the needs of all users, from local managers responsible for identifying potential opportunities to country-level controllers charged with validating and finalizing initiatives, significantly enhancing both visibility and control over the savings lifecycle. A critical design consideration was ensuring

a seamless transition for users accustomed to spreadsheet-based workflows, and the final result being an overall Excel-like table successfully allowed to tackle this potential issue; by blending the familiarity of traditionally used tools with the advantages of an integrated digital solution, the module reduces resistance to adoption while improving data consistency and accuracy. While lacking a quantitative measure of the module's impact, its qualitative contributions are evident: the module centralizes and modernizes the savings management process, making it more transparent and efficient; furthermore, as additional branches adopt the SmartSpend application, the system will fully eliminate the need for manual file uploads, ensuring that all savings initiatives are captured and reviewed within a single, cohesive platform.

Moving from the qualitative improvements brought by the Cost Control module, the SVR dashboard optimizations showcase measurable, quantitative advancements, aimed at addressing the critical performance bottlenecks that previously hindered the dashboard's responsiveness, particularly when working with large datasets. The table below quickly summarizes the changes of the KPIs considered, mainly consisting of load times and quantity of moved rows:

KPIs	**Before Development**	**After Development**
Page load time (worst case, no filtering applied)	16.4 s	1.7 s
Supplier breakdown table (dark grey) load time	32.4 s	3.8 s
Page load time (for Italy)	7 s	0.685 s
# of rows moved	4.7 million rows (Invoice table)	398,000 rows (Summary table)
# of suppliers represented	76,000	- (no change)

Table 4.1. KPIs comparison before and after development (results obtained via Google Chrome developer tools' network tab)

Before optimization, this section encountered significant delays when rendering data-heavy views. For example, in the worst-case scenario (loading the page without applying any data filters) the response time was over 16 seconds, causing frustration for end-users and impeding workflows; following the integration of the pre-aggregated table, this load time dropped dramatically to 1.7 seconds, representing a nearly tenfold improvement. Similarly, operations involving specific data segments saw substantial gains: the loading time of the supplier details table, located at the bottom of the page, initially required 32 seconds to load (still considering the worst case, the click on the dark gray portions of the graphs, representing the *still not invited* suppliers), and was subsequently reduced to just 4 seconds, ensuring smoother navigation and quicker access to critical insights. Filtered views also benefited, as demonstrated by the page load time for Italian suppliers, which improved from 7 seconds to an impressive 0.685 seconds. These improvements are closely tied to the reduced volume of data being processed: previously, queries pulled 4.7 million rows directly from the Invoice table, straining system resources and slowing response times; with the introduction of the summary table, this was reduced to 398,000 rows, significantly lowering the computational load while maintaining the integrity of the information displayed. Finally, the number of suppliers represented—a key metric of the page's fidelity—remained unchanged at 76,000 suppliers, obtaining the same level of data accuracy while constructing the overall dashboard. These quantitative results achieved underscore the transformative potential of efficient data handling and pre-aggregation techniques: together, these improvements are able to deliver a faster, more responsive tool, empowering users to analyze supplier data with the same level of precision but with far more efficiency.

Complementing these advancements, the asynchronous API designed for Invoice imports has introduced a robust and efficient solution to handle large scale data integration: by enabling smooth communication between RPA agents and the central ProcurementServiceCatalogue database, it automates the previously tedious process of data extraction and upload. It has demonstrated the capacity to handle files containing up to 70,000 rows within approximately 15 minutes, ensuring timely data processing, and its asynchronous nature eliminates timeout errors, a frequent issue with the synchronous implementation developed initially: by decoupling the data upload initiation from the processing phase, RPA agents can

now initiate imports, receive immediate confirmation of successful operation initiation, and monitor progress via a dedicated polling mechanism. Equally important is the API's rigorous error-handling mechanism: by enforcing strict header validation, it ensures that only properly formatted files are processed, drastically reducing data inconsistencies, while cell-level validation is able to detect missing or incorrect data, log non-critical issues as warnings and reject rows with critical errors. This approach guarantees that only valid invoices are persisted in the database while providing clear feedback on issues encountered, effectively ensuring data quality, enahncing reliability, and simplifying the integration workflow.

The combined results of these developments further solidify the overall platform's role as a comprehensive tool for procurement reporting, showcasing the transformative impact of qualitative and quantitative enhancements: from streamlining savings management to optimizing dashboard performance and automating data integration, each contribution has played an impactful role, demonstrating the potential of thoughtful design and innovative technology in driving operational efficiency.

Chapter 5

Future works and open issues

After having presented the accomplishments of the work done, this final section outlines opportunities for future enhancements to SmartSpend, ensuring its continued relevance in addressing the needs of top-management reporting and procurement processes. While significant advancements were made in optimizing performance and usability, either by introducing a new component or by improving data integration, the project also highlighted areas where further development can enhance both user experience and system maintainability.

One key area for improvement is the current reliance on the Freemarker template engine for server-side rendering; while being instrumental in delivering dynamic, data-driven content, it presents notable limitations in performance and scalability, particularly for data-intensive workflows: issues such as the need to reload templates for every page transition or when applying multiple filters create inefficiencies and hinder responsiveness. A move toward a Single Page Application architecture, potentially using Angular (already used in other projects among Sprint Reply), offers a promising solution: this would allow to shift rendering to the client side, improving page transitions, reducing server load, and significantly boosting the overall user experience. Additionally, separating the frontend from the backend by using a dedicated SPA would simplify development and maintenance, enabling the latter to focus exclusively on API delivery, while allowing the first to improve and progress independently. This shift would align SmartSpend with modern web development practices,

fostering scalability and long-term adaptability. It's worth noting that, while SPA architectures have known disadvantages (particularly in terms of Search Engine Optimization, mentioned in the previous sections), this is irrelevant for this platform: as a private enterprise application designed for internal use, it does not depend on public visibility or user engagement driven by SEO, making the SPA model particularly suitable, as it circumvents this common limitation without compromising functionality or usability.

Another critical area needing improvement would be the need of dedicated systems for performance and reliability monitoring. Currently, the identification of potential bottlenecks, such as slow APIs or database inefficiencies, relies primarily on user feedback, which delays resolutions and surely has a negative impact on the overall experience. A proactive approach could be achieved by introducing tools such as Grafana[1], Loki[2], and Prometheus[3] for system monitoring and analysis. The latter would serve as the primary tool for collecting and storing time-series data, offering robust support for querying and alerting based on performance metrics; Grafana would complement this by providing visual dashboards for real-time monitoring of API response times, database query performance, and server resource utilization, while Loki would centralize and streamline log management, making it easier to trace issues, identify trends, and debug failures efficiently. Together, these tools would allow the system to anticipate potential bottlenecks and address them before they affect users, ensuring reliability and a seamless experience.

These proposed improvements are not merely technical refinements, but rather strategic and impactful steps: by transitioning to an SPA, SmartSpend could embrace a modern, flexible architecture that delivers faster, more dynamic interactions while improving separation of concerns; likewise, integrating a comprehensive monitoring stack with Prometheus, Grafana, and Loki would significantly enhance the system's reliability, enabling the application to operate at peak performance even under growing data volumes and increasing user demands.

Looking to the future, these enhancements would elevate SmartSpend's

[1] https://grafana.com/

[2] https://grafana.com/oss/loki/

[3] https://prometheus.io/

capabilities, reinforcing its role as a vital tool for procurement decision-making: as the application continues to evolve, it must remain responsive to technological advancements and organizational needs, supporting both operational efficiency and strategic goals. The work presented in this thesis demonstrates the value of modular, user-centric development and lays a clear foundation for ongoing innovation, ensuring SmartSpend remains a leader in digital transformation for procurement processes.

Bibliography

[1] GeeksForGeeks, *What is a Web App?*, GeeksForGeeks, available at https://tinyurl.com/web-app-def [retrieved in Nov, 2024]

[2] Adobe Experience Cloud Team, *Single-page applications (SPAs) — what they are and how they work*, Adobe, 2023, available at https://business.adobe.com/blog/basics/learn-the-benefits-of-single-page-apps-spa [retrieved in Oct, 2024]

[3] Mozilla Development Team, *What is a progressive web app?*, MDN Web Docs, available at https://tinyurl.com/pwa-def

[4] Spring, *Annotation-based Container Configuration*, Spring Framework official documentation, available at https://tinyurl.com/annbasedconfig [retrieved in Oct, 2024]

[5] Spring, *Using the @SpringBootApplication Annotation*, Spring framework doc reference, available at https://tinyurl.com/sbappdoc [retrieved in Oct, 2024]

[6] Loredana Crusoveanu, *Intro to Inversion of Control and Dependency Injection with Spring*, Baeldung, 2024, available at https://tinyurl.com/ioc-di-intro

[7] Prakash Raj Ojha, *Spring Boot and Cloud-Native Architectures: Building Scalable and Resilient Applications*, International Journal of Computer Engineering and Technology, 2024

[8] O'Reilly News, *Cloud Adoption Steadily Rising Across Industries, but Managing Cost Remains a Concern, New O'Reilly Research Reveals*, O'Reilly, 2021, available at https://tinyurl.com/oreilly-cloud-survey [retrieved in Oct, 2024]

[9] JetBrains, *JetBrains State of Developer Ecosystem 2023*, 2023, available at https://tinyurl.com/jbsurv

[10] Sílvia Moreiraa, Henrique S. Mamedeb, Arnaldo Santosc, *Process automation using RPA – a literature review*, Procedia Computer Science 219, 2023, 244–254

[11] Gartner, *Gartner Says Worldwide Robotic Process Automation Software Revenue to Reach Nearly $2 Billion in 2021*, Gartner, 2020, available at https://tinyurl.com/rpagart

[12] Robert Eckstein, *Java SE Application Design With MVC*, 2007, https://tinyurl.com/mvcpatt

[13] Spring, *Annotated Controllers*, Spring Framework Javadoc API documentation, available at https://tinyurl.com/anncontrollers [retrieved in Oct, 2024]

[14] Spring, *Mapping Requests*, Spring Framework documentation, available at https://tinyurl.com/reqdocum [retrieved in Oct, 2024]

[15] Spring, *Annotation Interface Service*, Spring Framework Javadoc API documentation, available at https://tinyurl.com/servdocum [retrieved in Oct, 2024]

[16] Spring, *Core concepts*, Spring Framework documentation, available at https://tinyurl.com/coreconceptsrepo [retrieved in Oct, 2024]

[17] Rod Johnson, Juergen Hoeller, *Expert one-on-one J2EE development without EJB*, Wrox Press, 2004, available at https://tinyurl.com/j2eedevel

[18] Iuliana Cosmina, Rob Harrop, Chris Schaefer, Clarence Ho, *Pro Spring 6*, Springer, 2023, available at https://tinyurl.com/prospring6

[19] Oracle, *Annotation Type Column*, Javax Persistence official documentation, available at https://tinyurl.com/entitydocum [retrieved in Oct, 2024]

[20] Oracle, *Annotation Type Id*, Javax Persistence official documentation, available at https://tinyurl.com/iddocum [retrieved in Oct, 2024]

[21] Oracle, *Annotation Type Column*, Javax Persistence official documentation, available at https://tinyurl.com/columndoc [retrieved in Oct, 2024]

[22] Christian Bauer, Gary Gregory, *Java persistence with Hibernate (2nd ed.)*, O'Reilly Media, 2015, available at https://tinyurl.com/pershibern

[23] Oracle, *Annotation Type OneToMany*, Javax Persistence official documentation, available at https://tinyurl.com/onetomanydoc [retrieved in Oct, 2024]
[24] Oracle, *Annotation Type ManyToMany*, Javax Persistence official documentation, available at https://tinyurl.com/manytomanydoc [retrieved in Oct, 2024]