INSTITUTO POLITÉCNICO DE BRAGANÇA
Escola Superior de Tecnologia e Gestão

Academic E-Voting Platform

Ricardo Luis Cadilhe Nunes

Project report presented to the School of Technology and Management of Bragança to obtain the Master's degree in Informatics.

Supervisors:

Prof. Doutor José Eduardo Moreira Fernandes

This document does not include the suggestions made by the board.

Bragança

2024

Dedication

I dedicate the conclusion of this important stage of my life to my family, who have always supported me unconditionally.

My deepest thanks go to my parents, Luis Nunes and Sandra Nunes, for their constant love, sacrifice, and motivation.

I'm grateful to my brother, Miguel Nunes, for always bringing joy and lightness to my days, even during the most challenging moments.

To my grandparents, João Cadilhe, Fernanda Cadilhe, Manuel Nunes, and Isabel Nunes, for their inspiration, affection, and the strong values of integrity and respect that have guided me throughout life.

I also wish to acknowledge the continued support and presence of my godparents, Ricardo Nunes and Isa Carvalho, as well as my cousin Bernardo Nunes.

A special thanks to my girlfriend, Mariana Vinhas, for her patience, understanding, and unwavering support that uplifted me every step of the way, and for making my days in Bragança brighter, more meaningful, and filled with a renewed sense of purpose.

Finally, to all my friends who shared this journey with me, thank you for the laughter, companionship, and unforgettable moments that made this adventure lighter and richer.

To all of you, my heartfelt thank you.

Acknowledgment

The completion of this work for the Master's Degree in Informatics would not have been possible without the collaboration and support of several individuals, to whom I express my deepest gratitude.

First, I would like to thank my supervisor, Professor José Eduardo Fernandes, for his guidance, availability, encouragement, and trust throughout this project. His support was instrumental to the successful completion of this project.

I am also grateful to the School of Technology and Management of the Polytechnic Institute of Bragança for providing the environment, resources, and conditions that made this project possible.

I extend my sincere thanks to all the professors of the Master's Degree in Informatics, whose teaching, dedication, and shared knowledge greatly contributed to my academic and personal growth throughout the course.

This project is a collective effort, and I am deeply appreciative of everyone who played a role in its completion.

Abstract

This dissertation presents the design and implementation of an electronic voting (e-voting) platform tailored to the needs of academic institutions. Motivated by the inefficiencies, errors, and lack of accessibility found in traditional voting systems, the project aims to create a secure, user-friendly, and transparent solution that enhances participation and decision making in academic governance.

The platform supports both informal polls and formal elections, offering flexibility to accommodate a variety of academic decision-making contexts while maintaining high standards of security and usability.

The system was developed using a modern technological stack, including React for the frontend, FastAPI for the backend, and PostgreSQL for data management. Special attention was given to accessibility, responsiveness (to mobile devices, tablets, and desktop), and data security to ensure a seamless and inclusive experience for all users.

Designed with scalability, usability, and security in mind, the platform fulfills core functional and non-functional requirements while delivering a responsive and intuitive user experience.

Keywords: electronic voting, academic governance, accessibility, responsiveness, security, online elections, FastAPI, React, PostgreSQL

Resumo

Esta dissertação apresenta o design e a implementação de uma plataforma de votação eletrónica (e-voting) desenvolvida para responder às necessidades específicas de instituições académicas. Motivado pelas ineficiências, erros e falta de acessibilidade nos métodos tradicionais de votação, o projeto visa criar uma solução segura, intuitiva e transparente que promova a participação e a tomada de decisão no contexto académico.

A plataforma suporta tanto votações informais (sondagens) como eleições formais, oferecendo flexibilidade para se adaptar a diversos cenários de decisão, mantendo sempre elevados padrões de segurança e usabilidade.

O sistema foi desenvolvido com uma stack tecnológica moderna, incluindo React para o frontend, FastAPI para o backend e PostgreSQL para a gestão dos dados. Durante o seu desenvolvimento, foram particularmente valorizadas a acessibilidade, a responsividade (permitindo o uso em telemóveis, tablets e computadores) e a segurança, garantindo uma experiência de utilização eficaz, segura e adaptada a diferentes dispositivos.

Concebida com foco na escalabilidade, usabilidade e segurança de dados, a plataforma cumpre os principais requisitos funcionais e não funcionais, proporcionando uma experiência de utilização fluida e acessível.

Palavras-chave: votação eletrónica, governação académica, acessibilidade, responsividade, segurança, eleições online, FastAPI, React, PostgreSQL

Table of Contents

List of Tables

List of Figures

Acronyms

ACID Atomicity Consistency Isolation Durability.

AES Advanced Encryption Standard.

API Application Programming Interface.

CSS Cascading Style Sheets.

DDoS Distributed Denial of Service.

DOM Document Object Model.

ESTiG Escola Superior de Tecnologia e Gestão.

GDPR General Data Protection Regulation.

HTTP Hypertext Transfer Protocol.

IPB Instituto Politécnico de Bragança.

ISO International Organization for Standardization.

IT Information Technology.

JWT JSON Web Token.

LDAP Lightweight Directory Access Protocol.

ORM Object-Relational Mapping.

OS Operating System.

OTP One Time Password.

PC Personal Computer.

RDBMS Relational Database Management System.

REST Representational State Transfer.

SMS Short Message Service.

SOC Service Organization Control.

SQL Structured Query Language.

UI User Interface.

URL Uniform Resource Locator.

UTC Coordinated Universal Time.

Chapter 1

Introduction

The Introduction chapter serves as the gateway to this dissertation, providing readers with a comprehensive overview of the research focus, objectives, and structure of the document. In this chapter, we start on a journey that explores the development and implementation of an electronic voting (e-voting) system customized to meet the particular needs of educational establishments.

1.1 Problem Statement

Within the context of academic governance, traditional voting methods have long struggled with fundamental limits and inefficiencies that limit the smooth operation of democratic processes in educational establishments. Although manual voting techniques have been around for a while, they often face logistical obstacles, time limits, and errors which can seriously compromise the voting process's legitimacy and inclusivity.

Conventional academic voting systems reliance on paper ballots and human tabulation is one of their main disadvantages. This dependency creates chances for mistakes throughout the counting process, which might compromise the legitimacy and integrity of election results, in addition to creating logistical challenges. Furthermore, the time-consuming nature of manual voting methods frequently results in inefficiencies, which in turn causes longer voting times and lower participation from stakeholders rates.

Furthermore, the inactive character of conventional voting procedures prevents voters from actively participating in the political process, which impedes the development of a dynamic democratic culture in academic communities. These systems struggle to promote widespread involvement among students, staff, and administrators and meaningful engagement because they lack individualized interactions and real-time feedback mechanisms.

In addition, the lack of strong mechanisms for gathering and analyzing data in traditional voting systems prevents informed choices and blocks efforts to improve the academic experience in general. The ability of academic governing organizations to implement significant changes and react to the changing requirements of their members is still restricted in the absence of thorough insights into voting trends, inclinations, and opportunities for enhancement.

Given these obstacles, putting in place an electronic voting infrastructure seems like a sensible way to transform academic voting. The proposed platform seeks to overcome the drawbacks of conventional voting systems and bring in a new era of efficiency, inclusivity, and transparency in academic governance by utilizing innovative tools to speed up the voting process, improve engagement, and promote decision-making based on data.

1.2 Importance within Academic Context

In the contemporary academic landscape, the integration of innovative technologies is imperative for encouraging efficiency, inclusivity, and active participation. The adoption of an E-voting platform represents a crucial advancement that holds significant implications for both the voting process and the overall academic experience.

Traditional voting systems in academic institutions often face challenges such as logistical complexities, time constraints, and potential errors in manual tabulation. The implementation of an E-voting platform changes the entire voting process, offering an efficient method for students, faculty, and administrators. Through the elimination of paper-based ballots and manual counting, the platform not only accelerates the voting

procedure but also minimizes the likelihood of errors, ensuring the integrity of the electoral outcomes.

An E-voting platform introduces a dynamic and interactive element to the voting process, transcending the passive nature of traditional methods. Through user-friendly interfaces, real-time feedback, and personalized notifications, the platform encourages active participation and engagement among students and faculty. This not only enhances the democratic spirit within the academic community but also cultivates a sense of involvement and responsibility among the participants.

By leveraging technology to simplify administrative processes, the platform contributes to the creation of a more efficient and student centered environment. Additionally, the data generated through the platform can be analyzed to gain insights into voting patterns, preferences, and areas for improvement, enabling informed decision-making for academic governance.

1.3 Goals

The project's main goal is to solve the inherent flaws and restrictions present in conventional voting systems by implementing an electronic voting platform that is specifically designed to meet the needs of academic institutions. This project is motivated by many overarching objectives aimed at completely changing the environment around academic voting.

The most important of these objectives is to increase voting process efficiency. The platform aims to speed the conduct of election events, reduce the possibility of mistakes, and improve administrative procedures by substituting an automated digital system for human tabulation and traditional paper-based ballots. By incorporating user-friendly interfaces and real-time feedback systems, the platform aims to facilitate quick and easy voting processes, which in turn maximizes resource consumption and administrative effectiveness. Alongside efficiency gains, the initiative places a strong emphasis on fostering inclusion in academic communities. Acknowledging the value of creating a participatory

atmosphere, the platform seeks to improve accessibility and participation from all parties involved. The platform strives to remove obstacles to participation by providing customized alerts, accessibility features, and multilingual assistance, guaranteeing fair representation and inclusion in the voting process.

Encouraging active involvement and interaction among students, staff, and administrators is central to the project's goals. By utilizing interactive features like tailored alerts and real-time voting updates, the platform aims to cultivate a lively democratic culture within academic institutions by encouraging users to feel accountable and own their actions. The platform seeks to enable stakeholders to participate in the decision-making process actively, so augmenting the entire academic experience, through the facilitation of meaningful interactions and feedback systems.

Encouraging well-informed decision-making through extensive data analytics capabilities is one of the project's other main objectives. Academic governing organizations can obtain important insights into voting trends, preferences, and areas for development by utilizing the data produced by the platform. With the use of data, decision-making is made with more knowledge, allowing academic institutions to successfully implement significant changes and adapt to the changing demands of their constituent.

Lastly, maintaining the election process's integrity and openness is a priority for the initiative. By putting strong security mechanisms in place like audit trails and encryption protocols, the platform hopes to protect the privacy and integrity of vote results while encouraging participation and openness. The platform aims to improve confidence in academic governance systems and safeguard democratic norms by guaranteeing the integrity of the voting process.

1.4 Document Structure

This master's dissertation is organized into six chapters, each addressing a key component of the project. Chapter 1 introduces the research problem, its academic relevance, the goals pursued, and an outline of the document. Chapter 2 provides an analysis of existing

electronic voting solutions and the security criteria used to assess them. Chapter 3 focuses on software requirements engineering, presenting the functional and non-functional requirements, user stories, and the Figma UI mock-ups that guided the system's design. Chapter 4 describes the system architecture and the technologies used, namely React, FastAPI, and PostgreSQL, while also discussing scalability considerations. Chapter 5 details the implementation of each system layer, including the backend API, database schema and triggers, security mechanisms, and the React frontend components, establishing a direct link between these implementations and the requirements defined earlier. Finally, Chapter 6 concludes the dissertation by summarizing the work carried out, highlighting its limitations, and proposing future research directions. An appendix is included to provide auxiliary material and additional figures.

Chapter 2

State of Art

The literature review for this project was conducted to explore existing research and initiatives pertinent to the development and implementation of electronic voting (e-voting) systems. This review specifically targeted projects that share similarities with the current one, particularly in aspects related to the design, functionality, and security of e-voting platforms. By analyzing projects involved in the receipt, storage, and processing of e-votes, as well as those focusing on real-time information transmission within electoral processes, this review aimed to gain valuable insights to guide the advancement of the current e-voting project.

2.1 Introduction to Elections and the Rise of Electronic Platforms

Elections, whether conducted electronically or through traditional means, are fundamental processes for decision-making and representation across various domains. From national and local governance to corporate and community affairs, electronic voting (e-voting) systems play an essential role in ensuring fair, transparent, and efficient elections.

At the national level, elections involve voters electing representatives for legislative

bodies, the executive branch, and other administrative positions in government. Similarly, local elections elect members of local government bodies such as city councils and municipal governments. Corporate elections entail shareholder voting and board of directors elections, while professional bodies and trade unions conduct internal elections to elect officials.

In the academic sphere, electronic voting platforms are utilized to conduct various types of elections within educational institutions. These include elections for administrative positions such as members of pedagogical councils, course directors, and student council representatives. In addition, academic elections include the selection of class representatives, referendum votes on educational policies, and initiatives proposed by student groups. These elections empower stakeholders in the academic community to participate in the decision-making process and shape educational policies. Using e-voting technology, academic institutions simplify the electoral procedure, make it more transparent, and allow stakeholders to contribute to internal management.

In sum, just as in national governance and corporate governance, electronic voting systems facilitate democratic participation and ensure that individual voices are heard.

2.2 Electronic Voting

Electronic voting (e-voting) systems offer numerous advantages that make them an important tool for modern elections. These systems provide greater accessibility and convenience, allowing voters to cast their ballots from any location with internet access. This is particularly beneficial in increasing voter turnout, as it eliminates the need for physical presence at polling stations.

E-voting also enhances the efficiency and speed of the electoral process. The immediate tallying of votes and the reduction of human error in vote counting ensure quicker and more accurate election results. This is crucial in both large-scale national elections and smaller academic or corporate elections, where timely results are often necessary.

Security is a significant concern in any voting process, and e-voting systems are designed to address this through various measures such as end-to-end encryption, multi-factor authentication, and independent audits. These security features ensure the integrity and confidentiality of the voting process, preventing unauthorized access and manipulation of votes.

Furthermore, e-voting platforms provide greater transparency and auditability. The use of blockchain technology and other advanced security protocols allows for a verifiable and tamper-proof record of votes, which can be audited independently to ensure the election's fairness and integrity.

Overall, e-voting systems represent a significant advancement in the way elections are conducted, offering improved accessibility, efficiency, security, and transparency.

2.3 E-Voting Platforms

The idea of our study is to analyze several e-voting platforms to determine their strengths and weaknesses in the context of educational institutions. The criteria employed for evaluating these platforms include usability, features, and security. These criteria are vital for guaranteeing the efficacy, functionality, and integrity of the electronic voting system.

Usability ensures that the platform is user-friendly and easily navigable, encouraging broad adoption and interaction. It includes aspects such as interface design, accessibility features, and user support.

Features encompass the various functionalities the platform offers to accommodate a variety of voting needs and scenarios in educational settings. This includes customization options, support for different types of elections, and additional tools that enhance the voting experience.

Security involves measures to ensure the secrecy, integrity, and authenticity of the voting process. This includes encryption, authentication mechanisms, and compliance with security standards.

By evaluating platforms according to these criteria, we can gain a thorough understanding of their acceptability and efficacy. For this analysis we chose five prominent e-voting platforms - EVoting [1], ElectionBuddy [2], Voatz [3], Simply Voting [4] and X2Vote [5], we will analyze each chosen platform in detail to see how they measure up against these criteria.

2.3.1 Evoting

EVoting is a product developed by a company specializing in secure and efficient electronic voting solutions. The platform is designed to support a variety of election types, including those in educational institutions. EVoting aims to provide a user-friendly interface and robust security measures to ensure the integrity of the voting process. With a focus on transparency and ease of use, EVoting has been adopted by numerous organizations to facilitate their electoral processes.

Usability

Featuring a clean and organized interface available in multiple languages, EVoting provides an intuitive navigation menu for easy access to pertinent information. However, a comprehensive exploration of educational-specific usability features is warranted, as the current evaluation primarily focuses on the general interface and multilingual support.

Features

EVoting claims to adapt to various types of elections, including shareholder meetings, board meetings, and popular consultations. However, a critical evaluation reveals a lack of detailed insights into customization for educational-specific needs. The absence of information on elections such as class representatives or student committees raises concerns about its suitability for the comprehensive spectrum of educational processes.

Security

EVoting employs end-to-end encryption to protect votes, ensuring a baseline level of security. The mention of biometric authentication as an option further strengthens the platform's commitment to robust authentication processes, contributing to the overall security of the voting process.

2.3.2 ElectionBuddy

ElectionBuddy is a widely recognized e-voting platform known for its flexibility and extensive customization options. The platform is developed by ElectionBuddy, Inc., a company dedicated to providing reliable and accessible voting solutions for various types of organizations, including educational institutions. ElectionBuddy is designed to accommodate a wide range of electoral processes, from student council elections to board member votes, ensuring a secure and efficient voting experience.

Usability

ElectionBuddy distinguishes itself by offering extensive customization options, including themes, colors, logos, personalized instructions, and custom questions and answers. The platform's commitment to accessibility is notable, as it ensures compatibility with screen readers and other assistive technologies. This user-centric approach contributes to a tailored voting experience, accommodating the diverse needs of educational stakeholders.

Features

ElectionBuddy shines in terms of flexibility, adapting seamlessly to various elections, including leadership positions, student councils, associations, and clubs. The platform goes a step further by offering the option to create custom votes, showcasing a commendable commitment to tailoring the platform to the specific needs of each educational institution. This adaptability aligns with the dynamic nature of educational processes, ensuring a versatile and comprehensive e-voting solution.

Security

ElectionBuddy upholds high standards of security, employing end-to-end encryption to protect votes. The platform goes beyond the baseline by offering multi-factor authentication and providing independent audits of the voting process. The additional layer of security measures, coupled with SOC 2 Type 2 certification, positions ElectionBuddy as a platform committed to ensuring the security, availability, confidentiality, and integrity of the voting process.

2.3.3 Voatz

Voatz is an innovative e-voting platform developed by a technology company specializing in secure and accessible voting solutions. The platform leverages blockchain technology to enhance the security and transparency of the voting process. Voatz is known for its focus on ensuring voter integrity and preventing fraud, making it a suitable choice for high-stakes elections, including those in academic settings.

Usability

Voatz, though presenting a mobile platform with a modern interface, falls short by not supporting many languages like Portuguese, potentially limiting its usability in certain educational contexts. The absence of detailed insights into customization for educational institutions raises questions about its adaptability to the specific needs of schools and universities.

Features

Voatz, while asserting its adaptability to different types of elections, lacks specific insights into customization features tailored to the unique requirements of educational institutions. The absence of information regarding integration with academic systems or the creation of internal survey questionnaires raises concerns about its suitability for the intricate processes within schools and universities.

Security

Similar to EVoting, Voatz claims to utilize end-to-end encryption for vote protection. The platform also offers biometric authentication as an option, reflecting a commitment to advanced authentication methods for heightened security. The inclusion of independent audits further contributes to the overall security framework of the platform.

2.3.4 Simply Voting

Simply Voting is a well-established e-voting platform developed by Simply Voting Inc., a company dedicated to providing secure and straightforward voting solutions. The platform is designed to cater to a wide range of election types, including those in educational institutions. Simply Voting is known for its emphasis on security and compliance with industry standards, making it a reliable choice for conducting elections.

Usability

Simply Voting, in its commitment to user experience, provides a highly intuitive and easy-to-navigate platform for both administrators and voters. With customization features extending to themes, colors, logos, personalized instructions, and custom questions and answers, the platform ensures a positive and efficient voting experience. Moreover, its compatibility with screen readers and other assistive technologies further reinforces its commitment to accessibility.

Features

Simply Voting emerges as a robust solution in the Features dimension, exhibiting exceptional flexibility and customization options. The platform seamlessly adapts to different types of elections, including leadership positions, student councils, associations, clubs, student feedback, academic comments, surveys, budget allocation, award distribution, candidate selection for programs, and performance evaluation of teachers and staff. This

extensive coverage ensures that Simply Voting caters to the diverse needs of educational institutions, offering a comprehensive and adaptable e-voting solution.

Security

Simply Voting stands out in the Security dimension, employing end-to-end encryption with the AES-256 standard, considered one of the most secure in the world. The platform offers multi-factor authentication, providing users with a choice between different authentication methods, including password authentication, one-time passcode (OTP) authentication, and biometric authentication. Furthermore, Simply Voting undergoes independent audits conducted by cybersecurity-specialized companies, ensuring the integrity and confidentiality of the voting process. Compliance with SOC 2 Type 2 and the General Data Protection Regulation (GDPR) further solidifies the platform's commitment to

2.3.5 X2Vote

X2Vote is an emerging e-voting platform designed to offer flexible and secure voting solutions for various types of elections. The platform is developed with a focus on adaptability, making it suitable for both simple and complex voting scenarios. X2Vote aims to provide a comprehensive and user-friendly voting experience, with particular emphasis on security and transparency.

Usability

X2Vote offers a flexible platform for creating various types of online voting, ranging from simple to complex. Users can benefit from the convenience and accessibility of electronic voting, accessing the platform from any device with internet access, including PCs, Macs, iOS, and Android devices. The platform is compatible with all major browsers, ensuring accessibility for users regardless of their browser preferences. Moreover, X2Vote allows voting from anywhere in the world, providing participants with the flexibility to cast their

votes remotely. The voting process is efficient, delivering accurate and updated results in real-time immediately after the close of voting. Additionally, the platform ensures a secure and confidential voting experience, generating and securely sending electronic voting credentials via email, SMS, or traditional mail.

Features

X2Vote offers a wide range of features for both elections and assemblies. For elections, the platform supports secure and anonymous voting, with the ability to use multiple and sequential ballots. It allows for the segmentation of voting by regions, voter typology, and lists, among other criteria. X2Vote also offers hybrid voting options, combining in-person and remote voting methods, as well as encryption of votes and generation of decryption keys for enhanced security. The platform is auditable, providing transparency into the voting process, and calculates results in real-time immediately after voting closes. For assemblies, X2Vote enables various types of voting during meetings, with the option to add new voting topics during the session. It provides immediate results of votes, supports both anonymous and identified voting, and allows for differentiation of the number of votes per voter, if applicable.

Security

X2Vote prioritizes robust security measures to ensure the integrity and confidentiality of the voting process. The platform offers strong encryption methods, ensuring the security of votes and the protection of sensitive voter information. It is scalable to accommodate varying levels of voting activity and is designed to deliver precise and accurate results. X2Vote maintains a commitment to convenience without compromising security, offering transparency and auditability throughout the voting process. The platform prioritizes accessibility and usability, ensuring that all participants can easily navigate the voting interface. Additionally, X2Vote prioritizes voter privacy, safeguarding the confidentiality of each voter's choices.

2.4 Security Considerations in Electronic Voting

As we verified on every platform studied, security is critical in an e-voting platform.

A secure e-voting system must ensure that only eligible voters can access the voting process. This involves implementing strong authentication mechanisms that verify the identity of each voter effectively. The system should require more than just a username and password and should have multiple authentication factors, providing a higher level of security and reducing the risk of unauthorized access.

The platform must ensure the integrity of the voting process. This means that once you vote, it cannot be changed. Each vote should be securely logged and time-stamped, and the system should support auditing capabilities. This allows election officials and auditors to verify the correctness of the voting process and the count without compromising voter privacy.

The platform must maintain the confidentiality of voter choices, ensuring that votes cannot be traced back to individual voters. At the same time, the system must guarantee that each vote is anonymous yet verifiable. Voters should have confidence that their selections are private and that the system itself cannot link votes back to their identity, except for verifying eligibility and preventing multiple entries.

The platform should be designed to be highly available and resilient against various forms of cyber threats and technical failures. This includes the ability to resist high traffic volumes, potential cyberattacks such as DDoS, and hardware or software failures. Effective failover mechanisms and redundancy are essential to maintain service availability and to ensure that voting can continue smoothly in the face of disruptions.

In line with data protection regulations such as the GDPR, the e-voting platform must only collect and retain the minimal necessary personal data. The system should be designed to protect personal information with strict data handling and storage protocols and should provide voters with clear information about what data is collected, how it is used, and how long it is retained.

2.5 Conclusion

In conclusion, the overview of various types of elections, whether conducted electronically or through traditional means, underscores the fundamental role of e-voting systems in decision-making and representation across different domains. From national and local governance to corporate and community affairs, electronic voting platforms facilitate fair, transparent, and efficient elections, empowering stakeholders to participate in democratic processes.

Transitioning to the academic sphere, electronic voting platforms play a crucial role in conducting various types of elections within educational institutions. These include elections for administrative positions such as members of pedagogical councils, course directors, and student council representatives, as well as the selection of class representatives, referendum votes on educational policies, and initiatives proposed by student groups. By leveraging e-voting technology, academic institutions streamline the electoral process, promote transparency, and empower stakeholders to actively engage in governance and decision-making.

Turning our attention to the technical aspects, the comparative analysis of five prominent e-voting platforms provides valuable insights into their usability, features, and security dimensions. Each platform offers unique strengths and capabilities tailored to the diverse needs of educational institutions.

From the comparison table 2.1, it is evident that while all platforms prioritize usability, features, and security, there are notable differences in their specific offerings. For instance, EVoting and Simply Voting excel in usability with clean interfaces and accessibility features, while X2Vote stands out for its flexible platform and real-time result calculation. In terms of features, ElectionBuddy and Simply Voting offer extensive customization options, catering to various types of elections, while X2Vote provides support for secure and anonymous voting with hybrid voting options. Regarding security, all platforms employ robust measures such as end-to-end encryption and multi-factor authentication, with Simply Voting additionally complying with SOC 2 Type 2 and GDPR.

	Usability	**Features**	**Security**
EVoting	- Clean interface - Multiple languages supported	- Adaptable to various elections - Lack of educational-specific customization	- End-to-end encryption - Biometric authentication
ElectionBuddy	- Extensive customization - Accessibility features	- Adaptable to various elections - Extensive customization options	- End-to-end encryption - Multi-factor authentication - Independent audits
Voatz	- Modern interface - Limited language support	- Claims adaptability - Lack of educational-specific features	- End-to-end encryption - Biometric authentication - Independent audits
Simply Voting	- Intuitive interface - Accessibility features	- Adaptable to various elections - Extensive customization options	- End-to-end encryption - Multi-factor authentication - Independent audits - Compliance with SOC 2 Type 2 and GDPR
X2Vote	- Flexible platform - Convenient and accessible voting experience	- Support for various types of elections - Secure and anonymous voting - Hybrid voting options - Encryption of votes - Real-time result calculation	- Robust security measures - Strong encryption methods - Transparent and auditable process - Prioritization of accessibility and usability - Protection of voter privacy

Table 2.1: Comparison of E-voting Platforms

Chapter 3

Software Requirements Engineering

Before diving into the detailed specifications and design of the e-voting platform, it is crucial to establish a clear understanding of the system's requirements.

This chapter delves into the processes and methodologies utilized to gather, analyze, and define the requirements for the e-voting platform, ensuring that it meets the needs and expectations of all stakeholders involved.

3.1 Software Engineering Process

The development of the e-voting platform was managed through a structured software engineering process. The foundational understanding of software engineering principles was derived from the book by João M. Fernandes and Ricardo J. Machado [6], which provided valuable insights into the essential phases of the process. In my understanding, and as emphasized in the book, the phases of Inception, Elicitation, and Negotiation are particularly crucial. These phases were prioritized and focused on in our software engineering process, as they form the backbone of successful requirements engineering. For a more comprehensive exploration of these methodologies, the book by Karl Wiegers and Joy Beatty[7] was also referenced. While this book uses different nomenclature for the phases, it provides deeper insights and detailed guidance, reinforcing the importance of these phases by using alternative terminology. Additionally, it offers practical techniques

and best practices for applying them effectively. By concentrating on these three phases, we aim to ensure that our requirements engineering process is robust, detailed, and aligned with stakeholders' expectations, setting a strong foundation for the successful development of the e-voting platform.

3.1.1 Inception

The inception phase involves identifying the project's scope, defining the goals, and understanding the high-level requirements. It sets the groundwork for the project by establishing a clear understanding of what needs to be achieved. In our project, the objectives of developing an e-voting platform were clearly outlined in the first introductory chapter of this report.

During inception, we focused on:

- **Defining the Project Scope:** This included understanding the boundaries of what the e-voting platform should achieve within the academic setting.
- **Setting Goals:** Establishing what the platform aims to accomplish, such as ensuring secure and transparent elections.
- **High-Level Requirements:** Identifying the major functionalities and constraints of the system.

This phase is crucial because it lays the foundation for all subsequent activities in the project, ensuring that all stakeholders have a common understanding of the project's objectives, that in this case it's me, the professors that are following the development and the project coordinator.

3.1.2 Elicitation

The elicitation phase involves gathering detailed requirements from stakeholders through various techniques. Given the importance of this phase, several methods were employed:

- **Interviews and Meetings:** Conducting interviews and meetings with professors from different areas such as security, programming, and directors of IT departments to gather their insights and requirements.
- **Brainstorming Sessions:** Engaging in brainstorming sessions with the project coordinator and other stakeholders to generate ideas and refine requirements.

3.1.3 Negotiation

The negotiation phase focuses on resolving conflicts and prioritizing requirements to ensure that the final set of requirements is agreed upon by all. Activities in this phase included:

- **Priorization:** Ranking requirements based on their importance and feasibility, using the Pairwise Comparison method to systematically compare the relative importance of each requirement.
- **Conflict Resolution:** Addressing any conflicting requirements or stakeholder concerns through discussions and compromise.
- **Requirement Validation:** Ensuring that the gathered requirements accurately reflect the main objectives and are achievable within the project constraints.

3.1.4 Documentation

Documentation is a critical aspect of the software engineering process, serving as the primary means of communication among stakeholders and providing a reference for future maintenance and development. Proper documentation ensures clarity, continuity, and consistency throughout the project lifecycle.

To develop the project was adopted several methods to ensure thorough and effective documentation.

Requirements Documentation

- **Functional Requirements:** We utilized the Sequence Number technique to organize and manage the functional requirements presented in Table A.1. This method assigns each requirement a unique identifier (FR1–FR10), simplifying tracking, referencing, and validation throughout the development process. Each functional requirement is described concisely and categorized by priority to guide implementation and testing efforts.

- **Non-Functional Requirements:** As shown in Table **??**, non-functional requirements are documented separately to emphasize the system's quality attributes. These include performance, security, reliability, usability, and legal compliance. Like functional requirements, each non-functional item is uniquely identified (NFR1–NFR6) to support traceability and quality assurance.

Project Documentation

- **Version Control:** Using version control systems (e.g., Git[8]) to maintain a history of all changes made to the documentation. This allows for tracking modifications and reverting to previous versions if necessary.

3.2 Functional Requirements

The platform was designed to support secure and flexible ballot-based voting, covering both functional needs like authentication, ballot creation, and vote integrity. Functional requirements were identified through iterative planning and aligned with both user roles and expected platform behavior. Key features include multi-method participant management, OTP-based voting confirmation, and secure result publication.

(For the full table of Functional Requirements, see Appendix A.1.)

3.3 Non-Functional Requirements

Non-functional requirements were established to ensure the system remains performant, secure, reliable, and accessible under real-world usage. These include scalability expectations, data protection mechanisms, accessibility standards, and legal compliance such as GDPR.

(For the full table of Non-Functional Requirements, see Appendix A.2.)

3.4 User Stories

To ensure that development remained user-centered, a series of user stories were written and categorized according to three roles: user, creator, and voter. These stories helped guide the implementation of authentication, ballot management, and voting flows, aligning with real-world expectations for usability and security.

(For the full table of User Stories and Acceptance Criteria, see Appendix A.3.)

3.5 UI Mockups

To design the user interface of the e-voting platform, the collaborative tool Figma was used. The mockups served as a foundation for the frontend implementation in React,

aligning design decisions with the usability principles and functional requirements defined in Section A.1.

The UI was structured with a strong focus on simplicity, accessibility, and clarity. Interfaces were designed to ensure that users, regardless of technical proficiency, could securely participate in the voting process with minimal friction.

Figure 3.1 shows an overview of the most critical screens and their layout in Figma. These screens served as visual references and communication tools throughout the front-end development process.

Figure 3.1: Figma - Key UI Screens

Chapter 4

System Architecture and Technologies

The high-level architecture and structural structure of the e-voting platform are explored in detail in the Architecture section. It offers a thorough rundown of the architectural elements of the system, how they work together, and the underlying technological stack. This part provides insights into how the e-voting system will be constructed and organized to satisfy the functional, usability, and security criteria mentioned in the previous criteria chapter by outlining the architectural design.

The Architecture section seeks to build the foundation for the construction and deployment of the e-voting platform by drawing on industry best practices and known architectural patterns. To guarantee the stability and dependability of the voting system in the dynamic setting of academic institutions, it takes into account important factors such as system scalability, performance optimization, and security measures.

This part offers a process for turning the needs listed in the needs chapter into a logical and scalable architectural solution through careful planning and in-depth research. This section provides an overview of the structural structure of the e-voting platform, helping readers better understand how the various parts of the system will work together to provide a safe and easy voting experience.

4.1 Architecture Overview

The e-voting platform has been designed to ensure solid performance and to allow for security as well as scalability. It consists of many inter-connected parts, each of which plays an important role about how it works and its security features.

The platform supports both desktop and mobile users, allowing voters to access the system through various devices. The interfaces are built using React[9], a popular JavaScript library for building user interfaces. The frontend is developed with React, providing a responsive and dynamic user experience. React allows for efficient updates and rendering of components, making it ideal for handling the interactive elements of the e-voting system. The frontend communicates with the backend through HTTP requests and receives responses accordingly, ensuring seamless interaction between the user and the server-side logic.

The backend is powered by FastAPI[10], a modern, fast (high-performance) web framework for building APIs with Python. FastAPI is chosen for its speed and ease of development, allowing for quick implementation of secure and scalable APIs. The backend processes requests from the frontend, performs the necessary business logic, and returns the appropriate responses.

For data storage, the platform uses PostgreSQL, a robust and feature-rich relational database management system (RDBMS) known for its reliability, ACID compliance, and support for complex queries. It manages all data related to user accounts, voting records, and system logs, ensuring data is stored securely, efficiently, and with strong consistency guarantees.

Security is a critical aspect of the e-voting platform. While the login process relies on standard password-based authentication, user credentials are never stored in plain text. Instead, passwords are securely hashed using the bcrypt algorithm, which adds a salt and is computationally expensive, effectively protecting against brute-force and rainbow table attacks. The platform is also thought to integrate with an LDAP server to authenticate users within the institution's existing identity infrastructure, ensuring that only verified

academic members can access the system.

In addition to secure login, a second layer of protection is applied specifically during vote submission. Each voter is required to enter a One-Time Password, a 6-digit code sent to their registered contact to confirm and finalize their vote. This mechanism guarantees that only the intended voter can cast a ballot, reinforcing the system's integrity and preventing unauthorized submissions.

The e-voting system is secure, scalable, and efficient because of this construction. It makes the system ready to deal with pressures of an electronic voting context and secure users' data by utilizing state-of-the-art technologies such as FastAPI and React together with strong security measures.

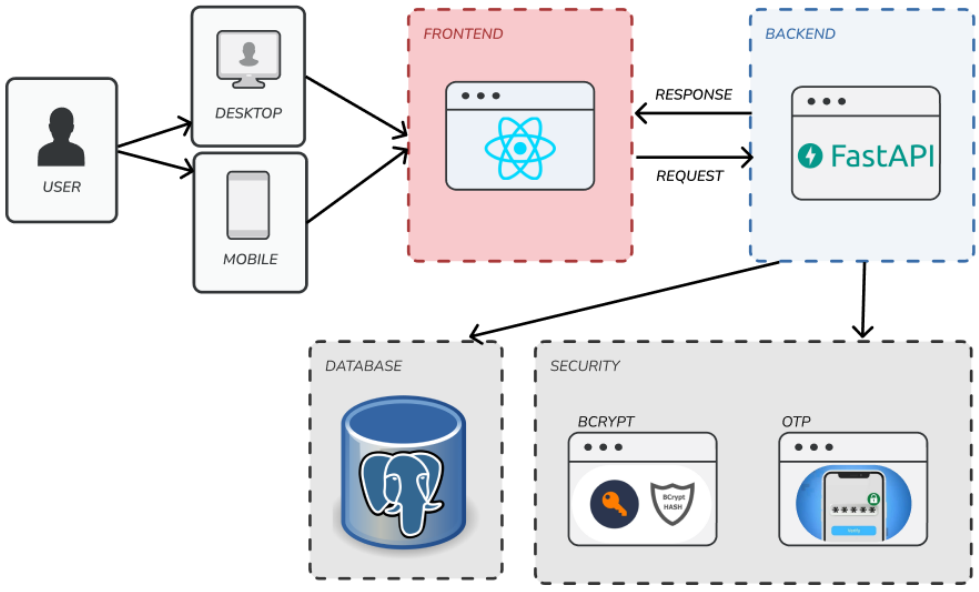

Figure 4.1: Architecture Overview

4.2 Technologies

The choice of technologies was significantly influenced by the existing technological infrastructure and the best practices already in place within our institution. In alignment with the established IPB technologies, it's easier to ensure integration and ease of maintenance.

The decision was to use React and FastAPI, which are technologies that have a proven track record at IPB, where they have been successfully implemented in various academic and administrative projects. Using these technologies, our goal is to create a robust, secure, and scalable e-voting platform that meets the high standards of our institution.

4.2.1 React

React is an excellent choice for both mobile and desktop development in the context of our e-voting platform due to its versatility and efficiency. React's component-based architecture allows for reusable UI components, which can be leveraged across different platforms, ensuring a consistent and cohesive user experience. This is particularly beneficial for an e-voting system, where a seamless interface is crucial for user engagement and accessibility.

Moreover, React's strong ecosystem includes libraries like React Native, which enables the development of native mobile applications for iOS and Android using the same React principles and components. This capability allows for the creation of a mobile app that complements the desktop version, ensuring that users can participate in the voting process from any device with ease.

Additionally, React's virtual DOM efficiently updates only the necessary parts of the user interface, resulting in fast rendering and a responsive experience, even under heavy user load. This performance is vital to maintain a smooth and reliable voting process, particularly during peak times. The extensive community support and robust tooling available for React further enhance its suitability for building a high-quality, cross-platform e-voting application.

4.2.2 FastAPI

FastAPI is an ideal choice for the backend of our e-voting platform due to its high performance, ease of use, and modern features. Built on Python, FastAPI leverages the latest asynchronous capabilities, which allow it to handle a high number of concurrent requests efficiently. This performance is crucial for an e-voting system, where timely and reliable processing of votes is essential to ensure a smooth and trustworthy election process.

FastAPI's strong support for data validation and serialization helps ensure that the data flowing through the system is accurate and secure. This is critical for an e-voting platform, where data integrity and security have big importance.

Additionally, FastAPI's compatibility with modern databases and its ability to integrate with various frontend technologies, including React, make it a versatile and powerful choice for our e-voting platform. The combination of these features positions FastAPI as a robust backend framework that can support the complex requirements of a secure, efficient, and user-friendly e-voting system.

4.2.3 PostgreSQL

PostgreSQL was selected as the database for this e-voting platform due to its proven reliability, strong support for relational data modeling, and extensive feature set tailored to transactional applications. Its robustness ensures data consistency and integrity, both of which are fundamental for maintaining trust in an electronic voting system.

The platform takes advantage of PostgreSQL's advanced constraint mechanisms to enforce rules such as preventing multiple votes by the same user and disallowing vote modification.

Altogether, PostgreSQL provides a secure, performant, and auditable foundation for managing critical election data.

Chapter 5

Implementation

This chapter details the technical implementation of the e-voting platform, providing insights into the frontend, backend, database design, and security measures adopted to ensure integrity and reliability.

5.1 Backend Implementation (FastAPI)

The backend of the e-voting platform was developed using FastAPI, a modern web framework for building APIs with Python. FastAPI was selected due to its high performance, automatic OpenAPI documentation generation, dependency injection system, and support for asynchronous operations, which are essential for modern RESTful services.

5.1.1 Structure and Organization

The backend code is structured into several logical modules to promote clarity, scalability, and separation of concerns:

- `main.py` – Entry point of the application. It initializes the FastAPI app, declares title, version, and tag metadata so that the automatically generated Swagger UI is grouped in the same three sections visible in the codebase (Users, Ballots and Notifications). Whitelists the React front-end origins and mounts /media so uploaded

files (profile pictures and ballot attachments) are served by the same process.

- **routers/:** The `routers` package is the public face of the backend. Each module groups together all endpoints that belong to one business-domain, making the API easy to navigate and to secure with FastAPI's dependency-injection system.

  - **`users.py`** – Identity and access-management.
  - **`ballots.py`** – The largest router; it implements the complete ballot life-cycle.
  - **`notifications.py`** – Lightweight wrapper around the `notifications` table.

  Each router is registered in `main.py` with a meaningful tag(*Users, Ballots, Notifications*) so that the generated Swagger UI is grouped exactly like the source code. All endpoints share common dependencies—database session via `get_db` and authenticated user via `get_current_user`—which keeps the handlers concise and testable.

- **`core/`** – Private service layer (helpers that are *imported* by routers but never exposed as HTTP endpoints).

  - **`security.py`** – Authentication and authorization.
    - Bcrypt password hashing (`CryptContext`).
    - `create_access_token` issues HS256 JWTs that expire after 60 minutes.
    - `get_current_user` decodes a token, fetches the `User` from the DB, and is injected as a dependency in every protected route.
  - **`utils.py`** – Miscellaneous but security-relevant utilities.
    - Robust `str_to_bool` that accepts `True/1/yes` etc.
    - OTP life-cycle: `generate_otp`, `generate_salt`, `hash_otp`, `send_email`.
    - `generate_voter_hash` produces a deterministic SHA-256 of `user_id || ballot_id || SECRET_SALT`; this keeps the voter anonymous while still preventing double voting.

  - **audit.py** – Single function `verify_chain_hash_for_ballot` that iterates over all votes of a ballot, recomputes the SHA-256 chain, and returns `False` on the first mismatch (algorithm explained in Section 5.3).
  - **tasks.py** – Two lightweight background loops using `fastapi_utils.tasks.repeat_every`; they share the same SQLAlchemy session pattern used by the routers, keeping the project free of external workers.

- **db/:** Centralizes the database layer. It includes the ORM models (`models.py`), data validation schemas (`schemas.py`), and database session and engine configuration (`database.py`) needed to manage persistent data with PostgreSQL.
  - **database.py** – Builds the PostgreSQL engine, defines a scoped `SessionLocal`, and provides the ubiquitous `get_db` generator used in Dependency Injection.
  - **models.py** – ORM mapping for every table (`users`, `ballots`, `votes`, ...) with bidirectional relationships, ON-DELETE cascades, and the extra integrity columns (`chain_hash`, `integrity_failed`).
  - **schemas.py** – Pydantic models that validate request bodies and serialise responses; `ConfigDict(from_attributes=True)` allows direct conversion from SQLAlchemy rows.

5.1.2 Endpoint List

To define the access points of the REST API, a URL structure was designed to clearly reflect the intent of each request. The structure begins with the entity associated with the functionality being performed by the request. Following this, the URL includes the action the request is meant to carry out in relation to that entity, and finally, a parameter identifying the specific resource when applicable.

Users

Method	Endpoint	Description
POST	/token	Login with email and password. Returns access & refresh tokens.
POST	/token/refresh	Refresh an expired access token using refresh token.
POST	/users/	Create/register a new user. Supports profile image upload.
GET	/users/me	Get information about the currently authenticated user.
GET	/users/me/statistics	Get statistics about the user's created and active ballots.
GET	/users/search	Search users by name, group, or course. Supports special commands like @everyone.

Table 5.1: User and Authentication API Endpoints

Notifications

Method	Endpoint	Description
GET	/notifications	Get all notifications for the current user. Supports filtering and sorting.
GET	/notifications/unread_count	Get the count of unread notifications.
POST	/notifications/mark_as_read	Mark all unread notifications as read.

Table 5.2: Notification-related API Endpoints

Ballots

Method	Endpoint	Description
POST	/ballots/	Create a new ballot with options and participants. Supports file attachment.
POST	/ballots/{ballot_id}/vote	Cast a vote in a ballot. Supports OTP verification.
GET	/ballots/{ballot_id}/results	Get the results of a ballot, including votes and stats.
GET	/ballots/mine	Get ballots created by the authenticated user.
GET	/ballots/visible	Get public ballots and private ballots user can access.
GET	/ballots/{ballot_id}	Get detailed info of a specific ballot.
DELETE	/ballots/{ballot_id}	Delete a ballot (only by the creator).
POST	/ballots/{ballot_id}/publish_results	Publish results after chain verification.
GET	/ballots/{ballot_id}/join	Join a ballot via QR code. Adds current user.

Table 5.3: Ballot-related API Endpoints

Figure 5.1 presents the automatically generated Swagger UI documentation provided by FastAPI. It offers an interactive interface for exploring and testing all available endpoints, showing details about request methods, parameters, responses, and authentication requirements.

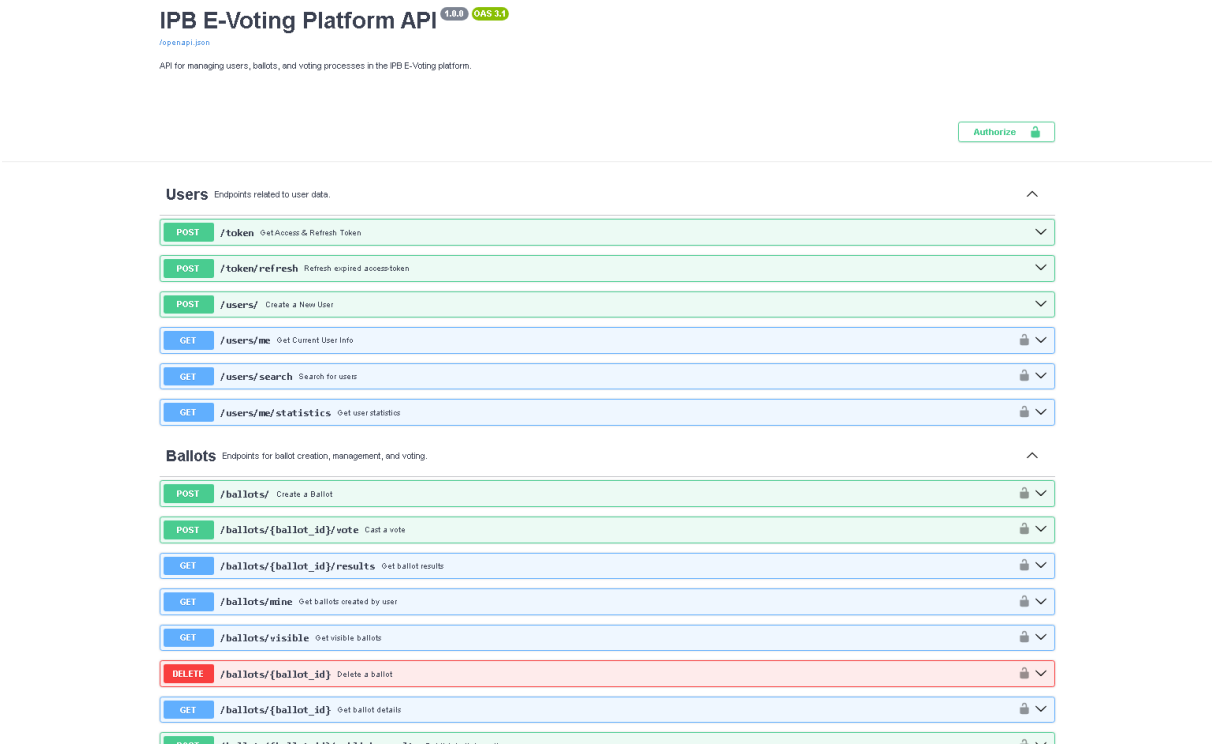

Figure 5.1: Swagger UI by FastAPI

5.1.3 End-to-End Backend Workflow

The following sequence describes the interaction of the main backend modules throughout a full voting workflow from user enrolment to final publication of results, highlighting how each component collaborates to ensure correctness, security, and traceability.

1. User Registration and Login *(development-only flow)*

During prototyping, we expose `POST /users/` so it is easier to create local test accounts. In production, this endpoint will be *disabled*: authentication will be delegated to the Instituto Politécnico de Bragança single sign-on service (LDAP/Active Directory), meaning only pre-existing institutional users will be able to access the platform.

A temporary account that *is* created through the REST API is handled securely — passwords are bcrypt-hashed and never written in clear text:

Chunk of code — user creation

```
hashed_password = get_password_hash(password)       # bcrypt
new_user = models.User(
    name=name, email=email,
    hashed_password=hashed_password,
    group_name=group_name, ipb_number=ipb_number,
)
db.add(new_user); db.commit()
```

Listing 1: Create a new user and persist it to the database

Once LDAP is connected, the table will keep only the minimal profile data retrieved from the directory, while credential storage and verification will be performed exclusively by IPB's central identity service.

Authentication uses OAuth2 "password → token" flow. Upon successful login the backend issues a short-lived JWT, signed with the server's `SECRET_KEY` and returned by

`POST /token`. Access tokens conform to the JSON Web Token standard [11], while user passwords are stored using the adaptive bcrypt hash proposed by Provos & Mazieres [12]:

Chunk of code — JWT creation

```
access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
access_token = jwt.encode(
    {"sub": str(user.user_id),
    "exp": datetime.now(timezone.utc) + access_token_expires},
    SECRET_KEY, algorithm="HS256"
)
```

Listing 2: Generate a signed JWT access token

All subsequent requests simply attach `Authorization: Bearer token`.

2. Ballot Creation

Authenticated users create a ballot through `POST /ballots/`. The endpoint accepts multipart-form data (text + optional file attachment) and JSON-encoded lists for `options` and `participants`. Timestamps are normalized to UTC on arrival.

Chunk of code — ballot creation

```
new_ballot = models.Ballot(
    created_by=current_user.user_id,
    title=title,
    description=description,
    start_time=start_time_dt,
    end_time=end_time_dt,
    visibility=visibility,
    is_anonymous=is_anonymous_bool,
    num_choices_allowed=num_choices_allowed,
```

```
    allow_blank_votes=allow_blank_votes_bool,
    results_publication=results_publication,
    file_attachment=file_attachment_path,
    allow_join_via_qr=allow_join_via_qr_bool,
    ballot_type=ballot_type,
)
db.add(new_ballot); db.commit(); db.refresh(new_ballot)
```

Listing 3: Insert a new ballot record and refresh its ID

Ballot-level configuration options

- **`ballot_type`** — choose between a simple *poll* (plain-text options) and an *election* in which each option may embed one or more `candidates` (`name`, `user_id`), enabling multi-seat contests.
- **`visibility` -** *public ballots* appear in the "Visible Ballots" list for all authenticated users; *private ballots* are hidden unless the user is on the participant list or joins via QR code.
- **`is_anonymous`** — when enabled, the vote row stores `NULL` in place of the voter's `user_id`, guaranteeing secrecy at the database level.
- **`num_choices_allowed` -** Limit on how many options a voter may select. Single-choice polls set this to 1; multi-select surveys can raise it to any positive integer.
- **`allow_blank_votes` -** Permits an explicit blank vote. Such ballots are recorded with `option_id = NULL` and are included in the final percentage calculations.
- **`results_publication` -** *auto* – results are published automatically when the voting period ends (background task). *manual* – only the creator can trigger `POST /ballots/{id}/publish_results` after an integrity check (see Section 5.3).

- **`allow_join_via_qr`** – If **`true`**, the organizer can generate a QR code that calls `/ballots/{id}/join`; scanning the code appends the current user to the participant list in real time.

After choosing these parameters the organiser selects the eligible voters. Each invited participant immediately receives a *participant_added* notification row that the frontend can poll or stream.

Chunk of code — add participant + notify

```
participant = models.Participant(
    ballot_id = new_ballot.ballot_id,
    user_id   = user_id,
)
db.add(participant)

if user_id != current_user.user_id:
    notification = models.Notification(
        user_id           = user_id,
        notification_type  = 'participant_added',
        data = {
            'ballot_id'   : new_ballot.ballot_id,
            'ballot_title': new_ballot.title,
            'start_time'  : new_ballot.start_time.isoformat(),
            'creator_name': current_user.name,
        },
    )
    db.add(notification)
```

Listing 4: Add a participant and generate a notification

These notifications are later retrieved through `GET /notifications` and rendered in

the client's inbox or pushed via WebSocket, ensuring that every eligible voter is informed of a new ballot the moment it is created.

3. Joining via QR Code

If the ballot organiser enables the flag `allow_join_via_qr = true`, the client application (outside the scope of this chapter) shows a QR code whose payload is the plain HTTP link `/ballots/id/join`.

When that URL is requested, the same JWT that protects every other API call is automatically attached in the `Authorization` header. The only backend responsibility is therefore to validate the request and update the database, as captured below.

Chunk of code — join via QR

```
@router.get("/ballots/{ballot_id}/join",
summary="Join ballot via QR code")
def join_ballot_via_qr(
    ballot_id: int,
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
    ballot = db.query(models.Ballot).get(ballot_id)
    if not ballot:
        raise HTTPException(404, "Ballot not found")

    already = db.query(models.Participant).filter_by(
        ballot_id=ballot_id, user_id=current_user.user_id
    ).first()
    if already:
        return {"message": "You are already a participant.",
                "ballot_title": ballot.title}
```

```
    db.add(models.Participant(ballot_id=ballot_id,
                              user_id=current_user.user_id))


    db.add(models.Notification(
        user_id          = current_user.user_id,
        notification_type= "participant_added_via_qr",
        data             = {
            "ballot_id"   : ballot.ballot_id,
            "ballot_title": ballot.title,
        },
    ))
    db.commit()
    return {"message": "You have been added to the ballot.",
            "ballot_title": ballot.title}
```

Listing 5: Endpoint for joining a ballot through a QR-code link

All standard middleware (JWT authentication, rate-limiting, structured logging) runs unchanged, so the QR workflow introduces no additional attack surface while providing a one-tap onboarding experience.

4. Casting a Vote with OTP

Voting is a two-step process:

1. **Request OTP** – client calls `POST /ballots/{id}/vote` without an `otp_code`. The backend generates a 6-digit code, salts and hashes it, stores the hash in the *otps* table, and e-mails the raw code to the voter.

2. **Submit Vote** – client resends the call, now including `otp_code`. The backend verifies the hash, deletes the OTP row, and records the vote(s). Each vote row

immediately receives its `chain_hash` before being committed.

Chunk of code — OTP verification & vote insert

```
provided_hash = hash_otp(vote_data.otp_code, otp_entry.salt)
if provided_hash != otp_entry.otp_hash:
    raise HTTPException(status_code=400, detail="Invalid OTP")

vote = models.Vote(
    ballot_id=ballot_id,
    option_id=option_id,
    is_blank=False,
    user_id=None if ballot.is_anonymous else current_user.user_id,
)
db.add(vote)
db.commit()
```

Listing 6: Validate OTP and record the vote

A voter hash (SHA-256 of user +ballot +secret-salt) is stored in *voters* so the same person cannot vote twice.

5. Automatic and Manual Result Publication

When `results_publication = auto`, a background task (`tasks.py`) polls every hour and sets `results_published = True` as soon as `end_time < now`.

For the manual mode the creator calls `POST /ballots/{id}/publish_results`. Immediately before publishing, the backend performs a *full chain-hash audit* (Algorithm 5.1.3) to guarantee the vote ledger is intact; if verification fails the ballot is flagged `integrity_failed = True` and publication is aborted.

Chunk of code — Verify chain hash

```
for vote in votes_in_order:
    payload = canonicalize(vote) + prev_hash
    if sha256(payload) != vote.chain_hash: return False
    prev_hash = vote.chain_hash
return True
```

Listing 7: Iteratively verify the vote hash-chain

6. Notification Pipeline

Two repeating background jobs complete the workflow:

- **check_ballot_statuses_task** — every 5s emits *ballot_started* and *ballot_ended* notifications and, for `auto` mode, flips `results_published`.
- **cleanup_expired_otps_task** — every hour deletes expired OTP rows.

Both tasks run in-process using `fastapi_utils.tasks.repeat_every`, leveraging the same SQLAlchemy session factory as the HTTP handlers.

Together these steps showcase how FastAPI, SQLAlchemy and PostgreSQL collaborate to deliver a secure, auditable voting backend while keeping the codebase modular and easy to reason about.

5.2 Database Implementation (PostgreSQL)

The database layer is critical for securely managing election data. PostgreSQL was chosen for its robustness, scalability, open-source nature, and its support for advanced features such as triggers and constraints, which were extensively used in this project. Schema objects and triggers are written according to the PostgreSQL 17 grammar and trigger interface [13].

5.2.1 Data Model

The entities managed within the database are ballots, candidates, notifications, options, otps, participants, users, voters, and votes. The database schema was carefully designed using SQLAlchemy, an Object Relational Mapping (ORM) tool that simplified database interactions and maintained consistency across application layers.

Explaining briefly each one:

- **ballots:** Represents individual voting events. Each ballot contains details such as the ID, title, description, voting period, visibility settings, and options available for selection.

- **candidates:** Represents individuals associated with specific ballot type *Elections*. Candidates link to users in the system.

- **notifications:** Stores system-generated notifications sent to users, informing them of important events such as voting start/end, invitations to ballots, or other alerts.

- **options:** Represents the choices available to voters within each ballot. Options can have a defined order and, in election-type *Election*, may link to candidates.

- **otps (One-Time Passwords):** Manages secure one-time-use codes sent to users, typically for authenticating sensitive actions like voting.

- **participants:** Defines which users are eligible to participate in specific ballots, establishing a direct association between users and ballots.

- **users:** Represents individuals registered in the platform, storing personal information, authentication credentials, and permissions.

- **voters:** Records participation in ballots by tracking voter hashes, ensuring voter privacy while verifying participation.

- **votes:** Represents each individual vote cast in a ballot, capturing essential details such as the selected option(s), timestamp, voter identity (if applicable), and cryptographic hashes ensuring data integrity.

Data Model

Figure 5.2 shows the relational schema used to support the core functionalities of the voting platform. It highlights the main entities (such as `users`, `ballots`, `votes`, `voters`, `options`, `otps`, `candidates`, `notifications` and `participants`) and their relationships, including foreign key constraints.

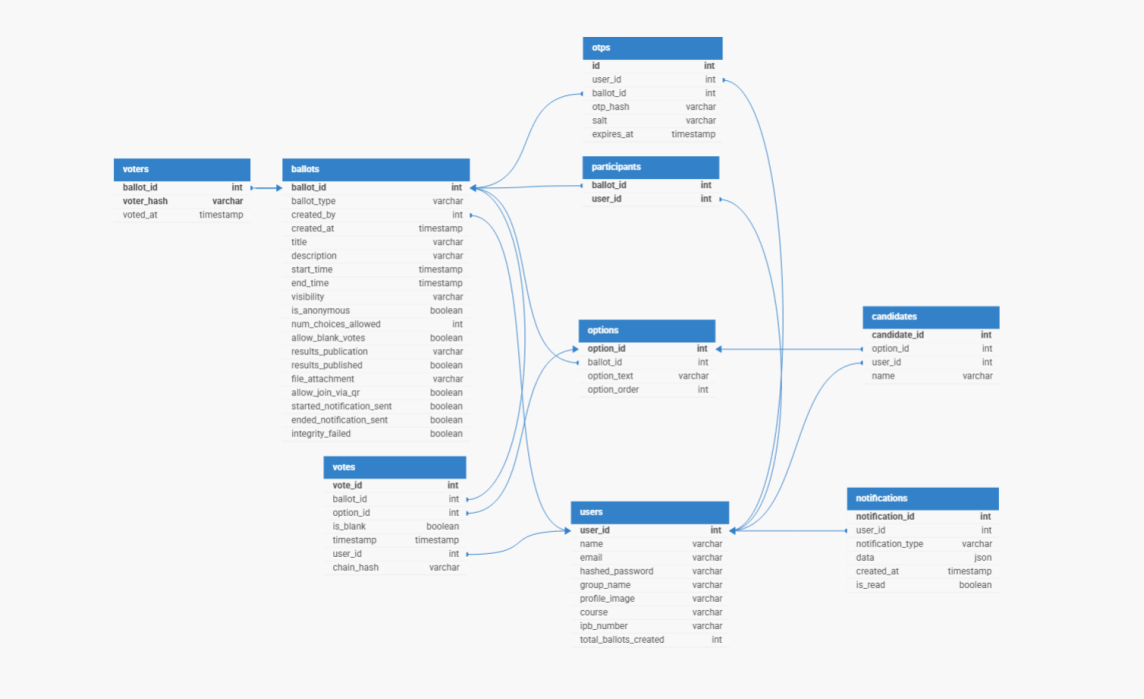

Figure 5.2: Relational database diagram of the platform

5.2.2 Database Constraints and Triggers

Constraints help maintain data consistency and accuracy, while triggers automatically enforce integrity rules at the database level. To ensure votes remain unmodifiable after insertion, PostgreSQL triggers were implemented to explicitly block both update and

delete operations on the *votes* table. These are enforced using a shared trigger function that immediately aborts any modification attempt.

Trigger Function:

Trigger Function: fn_block_update_delete

```
CREATE OR REPLACE FUNCTION fn_block_update_delete()
    RETURNS trigger
    LANGUAGE plpgsql
AS $$
BEGIN
    RAISE EXCEPTION 'Operation not allowed: votes are immutable.';
END;
$$;
```

Listing 8: PL/pgSQL trigger function that blocks UPDATE and DELETE

Trigger for DELETE:

Trigger: trg_block_delete

```
CREATE OR REPLACE TRIGGER trg_block_delete
    BEFORE DELETE
    ON public.votes
    FOR EACH ROW
    EXECUTE FUNCTION public.fn_block_update_delete();
```

Listing 9: Trigger to block DELETE operations on `votes`

Trigger for UPDATE:

Trigger: trg_block_update

```
CREATE OR REPLACE TRIGGER trg_block_update
    BEFORE UPDATE
    ON public.votes
    FOR EACH ROW
    EXECUTE FUNCTION public.fn_block_update_delete();
```

Listing 10: Trigger to block UPDATE operations on `votes`

These triggers actively enforce immutability at the database layer, ensuring that once a vote is recorded, it cannot be changed or removed. This layer of protection complements the passive integrity verification provided by the cryptographic hash-chain mechanism described in Section 5.3.

5.3 Security Implementation

Security is paramount in electronic voting systems, especially regarding data integrity and immutability. Due to the sensitivity of election data and the importance of maintaining public trust, robust mechanisms were implemented to prevent and detect unauthorized data alterations.

5.3.1 Data Integrity and Immutability

During the early stages of this project, the Bluefin extension for PostgreSQL was considered for enforcing database immutability. Unfortunately, Bluefin was only packaged for Ubuntu and, although we mounted an Ubuntu/PostgreSQL instance inside Docker, it became clear that the Bluefin project was no longer publicly maintained, forcing to explore alternative solutions.

Given these limitations, a custom-built approach was adopted, employing a cryptographic hash-chain mechanism to ensure vote data integrity directly within PostgreSQL.

5.3.2 Cryptographic Hash Chain

Given the constraints with external tools, we implemented a custom cryptographic hash-chain approach directly within PostgreSQL. This method involved storing a SHA-256 hash for each vote, chaining each vote's data with the previous vote's hash. This approach ensures that even a slight change in the data breaks the integrity of the entire chain, making unauthorized changes immediately detectable.

The *votes* table schema was modified to this hash-chain method as follows:

SQLAlchemy Vote Model

```
class Vote(Base):
    __tablename__ = 'votes'
    vote_id = Column(Integer, primary_key=True)
    ballot_id = Column(
        Integer,
        ForeignKey('ballots.ballot_id', ondelete='CASCADE')
    )
    option_id = Column(
        Integer,
        ForeignKey('options.option_id', ondelete='SET NULL')
    )
    is_blank = Column(Boolean, nullable=False)
    timestamp = Column(DateTime(timezone=True), default=func.now())

    user_id = Column(
        Integer,
```

```
        ForeignKey('users.user_id', ondelete='SET NULL')
    )
    chain_hash = Column(String, nullable=False)  # Stores SHA-256 hash
```

Listing 11: SQLAlchemy ORM model for the `votes` table

5.3.3 Vote Integrity Verification Process

The integrity verification involves recalculating the hash for each vote and comparing it with the stored hash. A critical challenge in this process was ensuring consistent data representation between PostgreSQL and Python.

Timestamp Precision:

PostgreSQL triggers and Python scripts represented timestamps differently due to variations in microsecond precision and timezone handling. This subtle difference broke hash consistency. To solve this, we standardized both sides converting timestamps explicitly to UTC and formatting timestamps with exactly six-digit microseconds (%f).

For example, the trigger-side implementation was standardized as:

Trigger-side Implementation

```
NEW.ts_utc := (NEW.timestamp AT TIME ZONE 'UTC')::timestamp(6);
```

Listing 12: Ensure timestamps are stored in UTC within triggers

And the Python-side implementation:

Python-side Implementation

```
ts = vote.timestamp.astimezone(timezone.utc).
strftime("%Y-%m-%d %H:%M:%S.%f")
```

Listing 13: Convert timestamp to ISO-8601 with microseconds in Python

Boolean Representation (*is_blank*):

Initial inconsistencies also arose from how Boolean values (is_blank) were represented between SQL ('t'/'f') and Python ('True'/'False'). These inconsistencies led to hash mismatches. The solution involved explicitly converting both sides to consistent literal strings:

In Python:

```
bool_str = "True" if vote.is_blank else "False"
```

Listing 14: Boolean serialization of blank vote flag in Python

In PostgreSQL triggers:

```
bool_str := CASE WHEN NEW.is_blank THEN 'True' ELSE 'False' END;
```

Listing 15: Boolean serialization of blank vote flag in PL/pgSQL

The finalized Python routine for verifying the hash chain became:

```
import hashlib
from datetime import timezone

def verify_chain_hash_for_ballot(db: Session, ballot_id: int) -> bool:
    votes = (
        db.query(Vote)
          .filter(Vote.ballot_id == ballot_id)
          .order_by(Vote.vote_id)
          .all()
    )
    last_hash = "GENESIS"

    for vote in votes:
```

```
            ts = vote.timestamp.astimezone(
                timezone.utc
            ).strftime("%Y-%m-%d %H:%M:%S.%f")
            bool_str = "True" if vote.is_blank else "False"
            data_str = (
                f"{vote.ballot_id}|"
                f"{vote.option_id or 'NULL'}|"
                f"{bool_str}|"
                f"{ts}|"
                f"{vote.user_id or 'NULL'}|"
                f"{last_hash}"
            )
            computed_hash = hashlib.sha256(data_str.encode()).hexdigest()
            if computed_hash.lower() != (vote.chain_hash or "").lower():
                return False
            last_hash = computed_hash
        return True
```

Listing 16: Complete Python function to verify the full vote hash-chain

This ensures that any minor data alteration immediately results in a chain mismatch, ensuring that vote tampering cannot go unnoticed.

5.4 Frontend Implementation (React)

This chapter presents the final appearance of the graphical interface of the application. Each of the developed screens will be analyzed in more detail, explaining the approach taken in their implementation and illustrating their design and functionality.

The layout adopted for the application was guided by the mockups previously defined

in the Section 3.5. The main concern behind the chosen layout was to ensure ease of use, with a user-centered design that favors intuitive and straightforward navigation.

To maintain visual coherence and familiarity for users within the institution, the platform adopts color tones already used in other official IPB platforms. This choice reinforces the identity of the institution while also providing a familiar and trustworthy environment for users. The interface follows a consistent design across all views, ensuring a uniform and predictable experience.

5.4.1 Register

The user registration page was developed to initially support user onboarding by allowing new users to create an account within the IPBvoting system. However, it is crucial to note that the goal for the production phase is the integration with the Instituto Politécnico de Bragança's LDAP authentication system. This means that the registration page described here primarily served testing and development purposes and will not be available to users in the final deployment.

A detailed overview of the fields, their expected behavior, and validation rules is provided in Appendix B.1.

The registration interface (illustrated in Figure 5.3) was designed with an emphasis on clarity, simplicity, and user-friendliness. The layout follows the same guidelines established previously in the mockups phase (described in Section 3.5), and uses color tones already familiar to IPB users from other institutional platforms, ensuring a visually coherent and trustworthy experience.

From a technical perspective, the Form state and validation logic are managed through the `useState` hook, allowing the component to dynamically update field values and error messages. Each form field is a controlled input, meaning its value is linked to the component's state, making it possible to implement real-time validation and feedback.

The form submission logic relies on the `axios` library to perform an asynchronous HTTP POST request to the backend API. A `FormData` object is used to properly handle

multipart form data, including the optional profile image. The following snippet illustrates the asynchronous submission logic:

HTTP POST request for user registration:

```
await axios.post('http://localhost:8000/users/', submitData, {
  headers: { 'Content-Type': 'multipart/form-data' },
});
```

Listing 17: Axios call to create a new user with multipart form data

Upon submission, the application validates all required fields locally. If the data is valid, a toast notification is displayed using the `react-toastify` library, and the user is redirected to the login screen via the `useNavigate` hook from `react-router-dom`. In case of errors, messages are presented both inline under the corresponding input fields and as global toasts from `React Toastify` for better visibility.

It is important to emphasize once again that, despite the thorough development and robust handling of user registrations detailed here, the final deployment intends to omit this feature entirely. In production, the IPBvoting system will exclusively utilize LDAP authentication, limiting access only to verified IPB users, thereby enhancing both security and integration consistency.

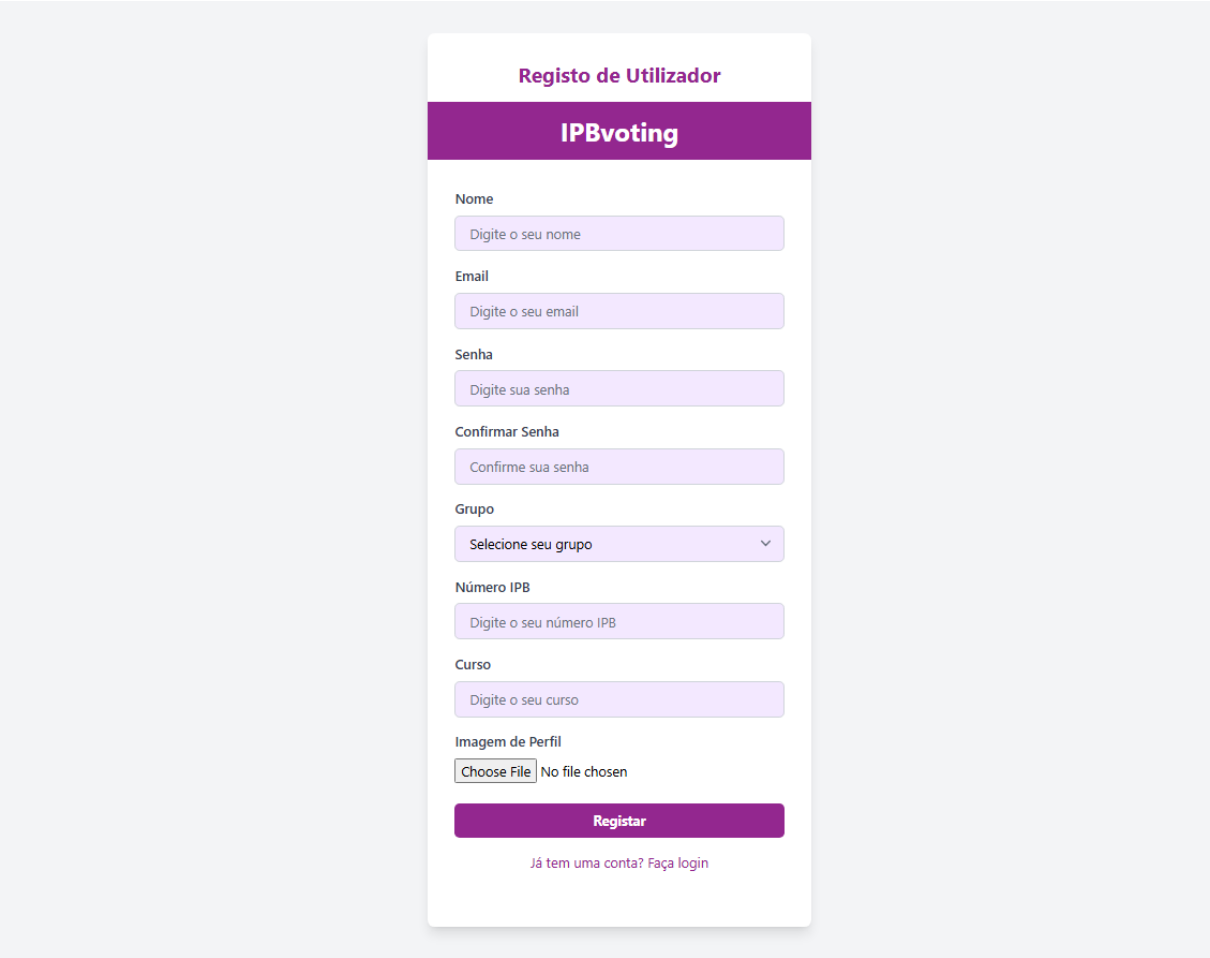

Figure 5.3: Register Page

5.4.2 Login

The login screen is a fundamental part of the IPBvoting system, serving as the secure entry point for all users. It was designed to be simple, responsive, and visually aligned with IPB's existing digital identity. Just like other screens in the application, it adopts a layout based on the previously defined mockups (refer to Section 3.5) and uses the institutional color palette to reinforce trust and familiarity.

The interface (Figure 5.4) focuses on clarity and usability. It contains two input fields — email and password — both clearly labeled and styled with sufficient spacing and contrast. The password field uses masked input to preserve privacy, while the email field

enables keyboard optimizations for mobile users, such as showing the "@" symbol.

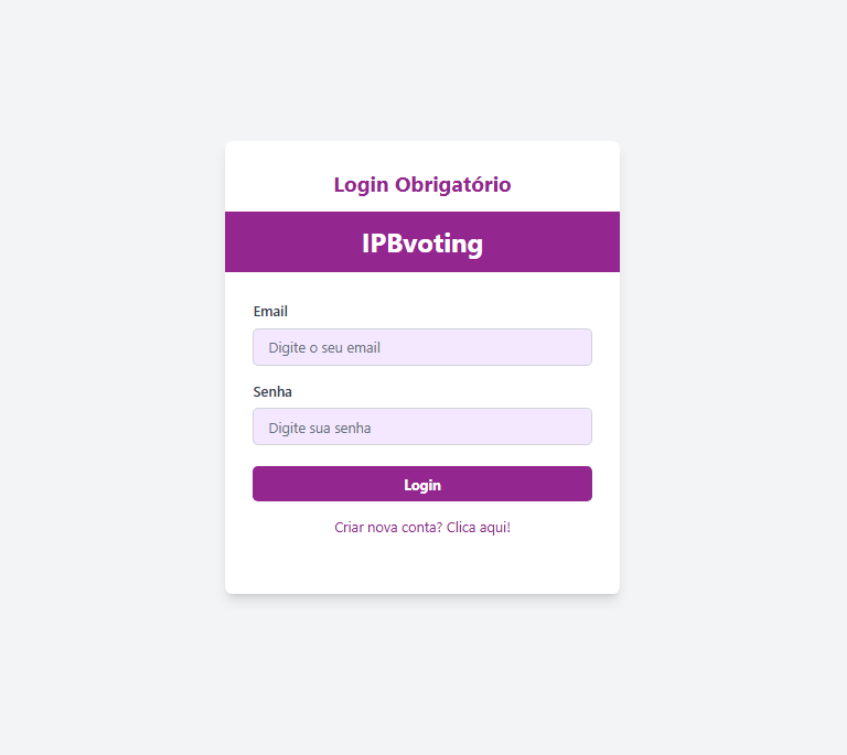

Figure 5.4: Login Screen

After submitting the form, if the credentials are correct, the user is authenticated and redirected to the homepage. Otherwise, the system provides clear feedback through error messages, ensuring users are aware of what went wrong and can act accordingly.

From a technical perspective, the Login component is implemented as a React functional component, using the `useState` hook to store the input values for email and password. Form submission is handled through the `handleSubmit` function, which makes an asynchronous HTTP POST request to the backend authentication endpoint using `axios`. The request includes the credentials in *`application/x-www-form-urlencoded`* format, which is compatible with FastAPI's OAuth2 token endpoint.

POST request to obtain JWT token:

```
const formData = new URLSearchParams();
formData.append('username', email);
formData.append('password', password);

const response = await axios.post(`${API_BASE_URL}/token`, formData, {
  headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
});
```

Listing 18: Obtain an access/refresh JWT pair via OAuth2 password flow

Once the response is received, the component extracts the `access_token`, `refresh_token`, and `user_id` from the payload and stores them in the browser's `localStorage`. These tokens are essential for maintaining the user's session and authorizing further requests throughout the system.

Storing tokens in localStorage:

```
localStorage.setItem('access_token', access_token);
localStorage.setItem('refresh_token', refresh_token);
localStorage.setItem('user_id', user_id);
```

Listing 19: Persist JWT tokens in the browser's localStorage

To manage navigation after login, the system uses the `useNavigate` hook from `react-router-dom`. This allows a seamless redirection to the homepage upon successful authentication. Throughout the login flow, toast notifications are used to inform users about the outcome of the process, whether it is a successful login or an error, using the `react-toastify` library.

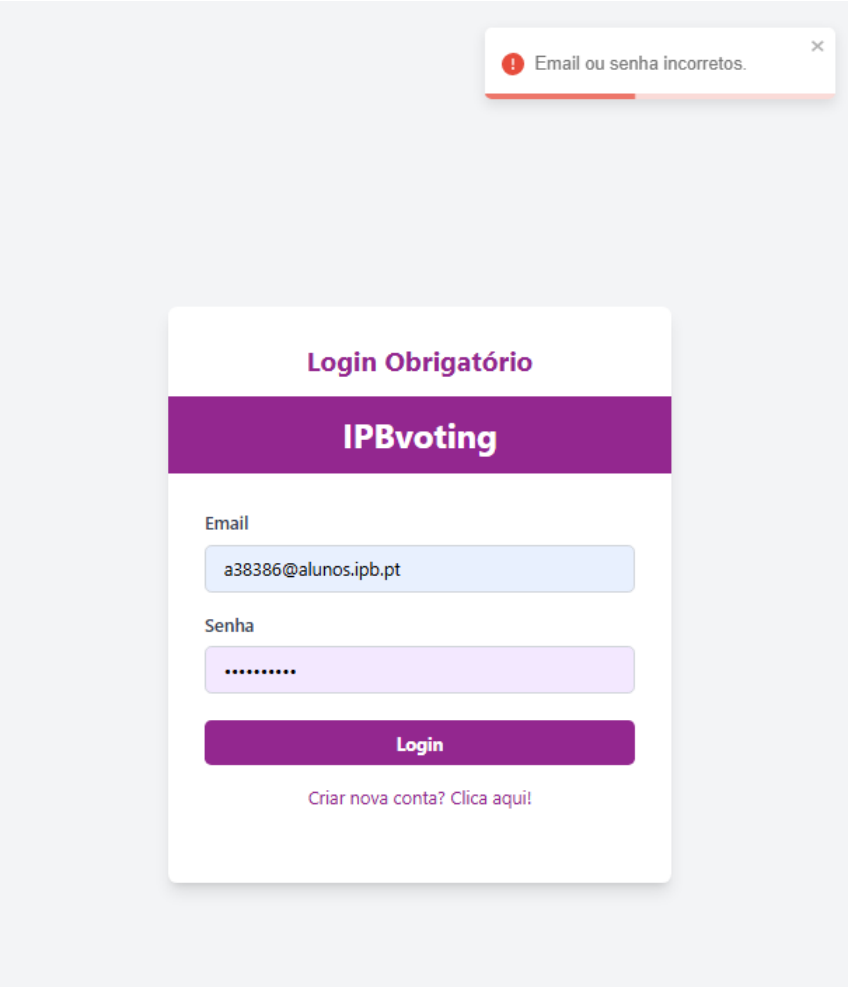

Figure 5.5: Login screen - Email or Password incorrect

5.4.3 Homepage

After a successful login, the user is redirected to the homepage, the main landing screen of the IPBvoting platform. This page acts as a central hub, combining navigation access, voting guidance, feature highlights, and institutional resources, all within a clean and intuitive interface.

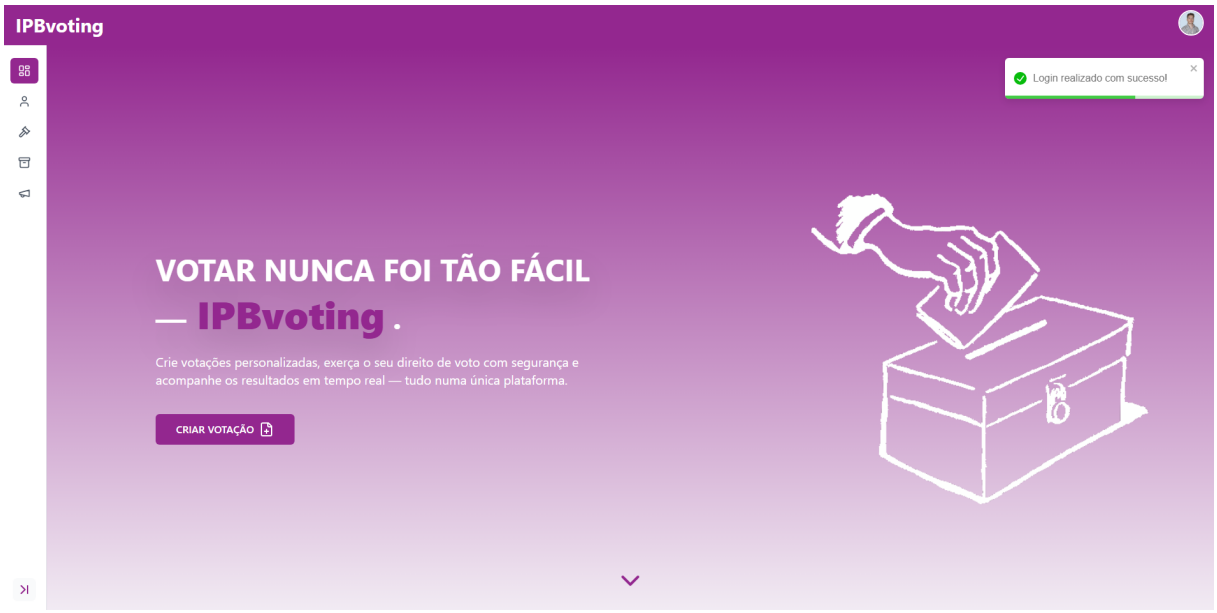

Figure 5.6: Homepage

The layout follows a dashboard-like structure composed of a top navigation bar and a collapsible sidebar for quick access to key areas. The homepage's hero section uses a bold slogan and call-to-action button to invite users to create their first ballot.

A smooth scrolling animation, triggered by a downward arrow icon, enhances the interactive experience and helps guide users through the page flow.

The color palette used aligns with IPB's branding, reinforcing the visual consistency established in earlier pages. The combination of flat colors, generous spacing, and clear typography helps convey the platform's emphasis on accessibility and simplicity.

From a technical perspective, this component is implemented as a React functional component. Navigation structure is handled through reusable layout components such as `<Navbar />` and `<Sidebar />`. Interactivity such as scrolling to sections or toggling steps is controlled using hooks like `useRef` and `useState`.

Using useRef to scroll to sections:

```
const featuresRef = useRef(null);
const scrollToFeatures = () => {
```

```
  if (featuresRef.current) {
    featuresRef.current.scrollIntoView({ behavior: 'smooth' });}
};
```

Listing 20: Smooth-scroll to a section with React `useRef`

A major UX component of the homepage is the tutorial section, which guides users step-by-step through the voting process.

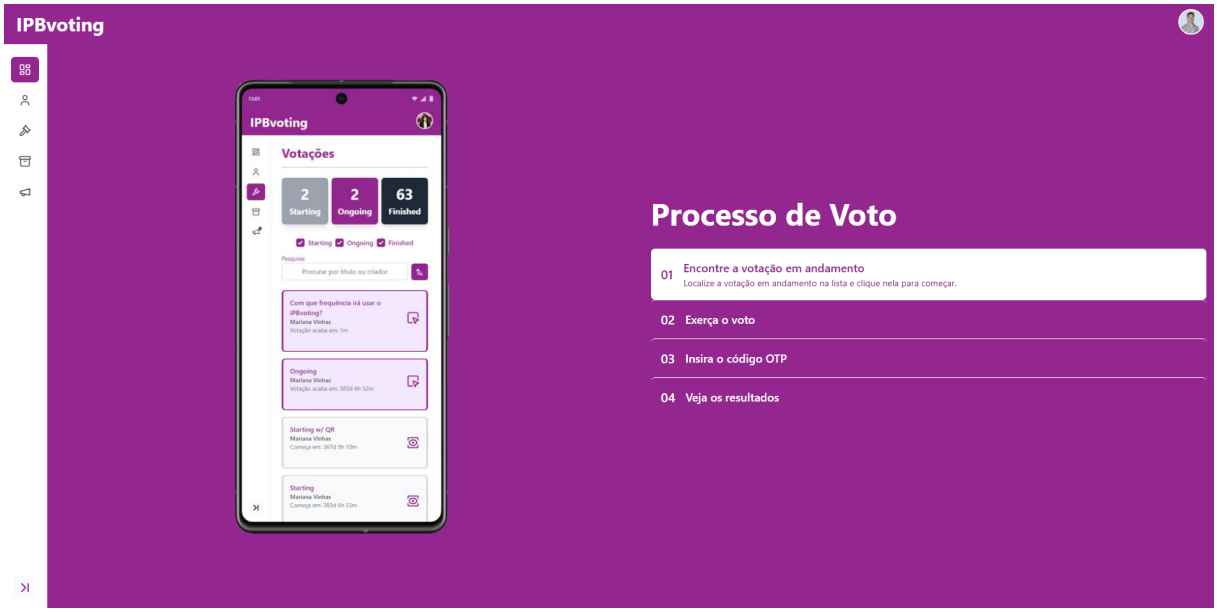

Figure 5.7: Homepage - Tutorial Section

This is implemented with a dynamic list that highlights the selected step and updates the accompanying image and description accordingly.

Step-by-step tutorial with dynamic state:

```
const [currentStep, setCurrentStep] = useState(0);
const steps = [
  { title: 'Encontre a votação...', content: '...', image: '...' },
  { title: 'Exerça o voto', content: '...', image: '...' },
  // ...];
```

Listing 21: React state logic for the homepage tutorial steps

Additionally, the homepage includes a feature section that presents six key benefits of the IPBvoting platform, such as security, transparency, ease of use, and scalability. Each benefit is visually represented with an icon from the Lucide library, and designed using Tailwind CSS utility classes for consistent styling.

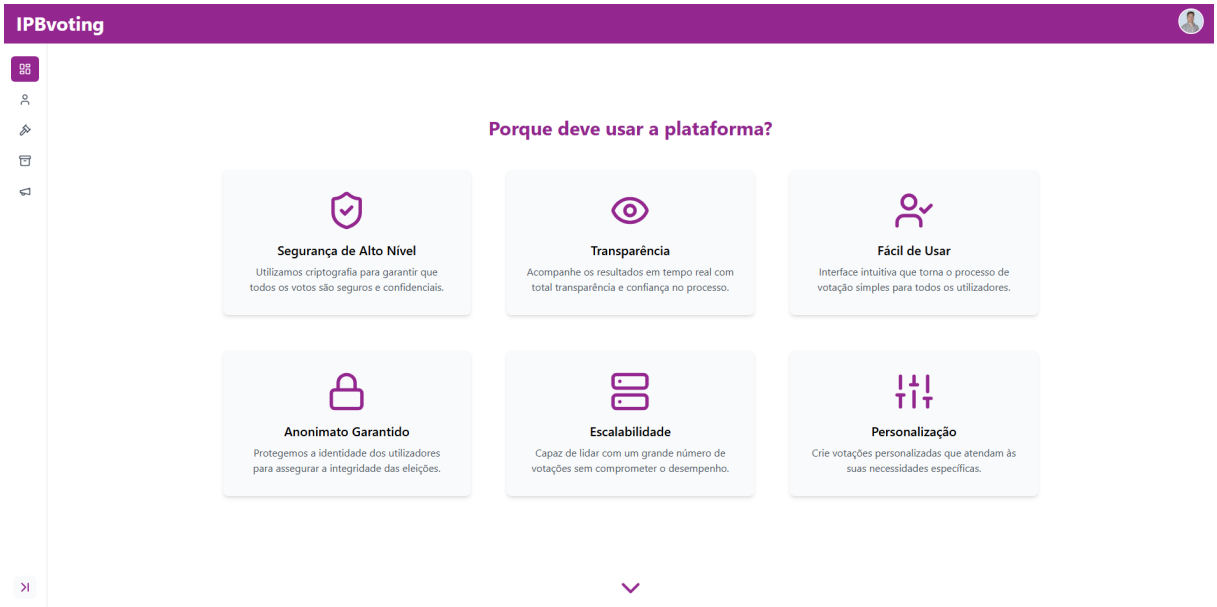

Figure 5.8: Homepage - Feature Section

At the end of the homepage, users find a well-organized section of useful IPB-related

links. These are presented in three columns, encouraging exploration of complementary institutional services and information.

The overall design encourages engagement and fosters confidence in the voting process by educating users and showcasing the platform's reliability and ease of use.

5.4.4 Profile

The Profile page allows users to visualize and manage their personal information, voting statistics, and OTP (One-Time Password) preferences. It plays a crucial role in giving users control over how they authenticate their votes and review their activity within the platform. The design follows the same visual language adopted throughout IPBvoting, maintaining consistent spacing, neutral backgrounds, and institutional colors.

Upon accessing the Profile page, the user is greeted with a hero banner, a personalized avatar, and their name. Below this, the content is split into two main columns: personal details and system statistics on the left, and OTP configuration and metrics on the right. This layout ensures clarity and accessibility across different screen sizes, thanks to a responsive grid based on Tailwind CSS.

The page begins by fetching the authenticated user's data from the backend, along with aggregated statistics such as the total number of created ballots and active ballots. These requests are executed in parallel using the `Promise.all()` API, ensuring fast loading without blocking the interface.

Fetching user profile and statistics:

```
const [userRes, statsRes] = await Promise.all([
  api.get('/users/me'),
  api.get('/users/me/statistics'),
]);
```

Listing 22: Parallel API calls to fetch user profile and stats

All user data is stored using React's `useState` hook, and the initial fetch is triggered

inside a `useEffect`, which ensures the logic runs only once when the component mounts. If any error occurs (such as an expired token), the user is redirected to the login page, and toast notifications are used to inform them of any issues encountered.

The form structure on this page is primarily read-only, except for the OTP preference section. Here, the user can choose between receiving OTP codes via email or phone. If "phone" is selected, the form dynamically displays fields for the country code and mobile number. This is controlled using React's conditional rendering.

Once the user selects their preferred OTP method and clicks save, the new preferences are sent to the backend via an HTTP `PUT` request. Immediately after saving, the user profile and statistics are re-fetched to ensure the UI reflects the updated state.

Additionally, two statistics are shown to the user via small stat cards: the total number of ballots created and the number of ongoing ones. These are implemented as reusable components for better code organization and visual consistency.

To complete the page, a logout button is placed in the bottom right corner. When clicked, it removes the access and refresh tokens from `localStorage` and navigates the user back to the login page using React Router's `useNavigate` hook.

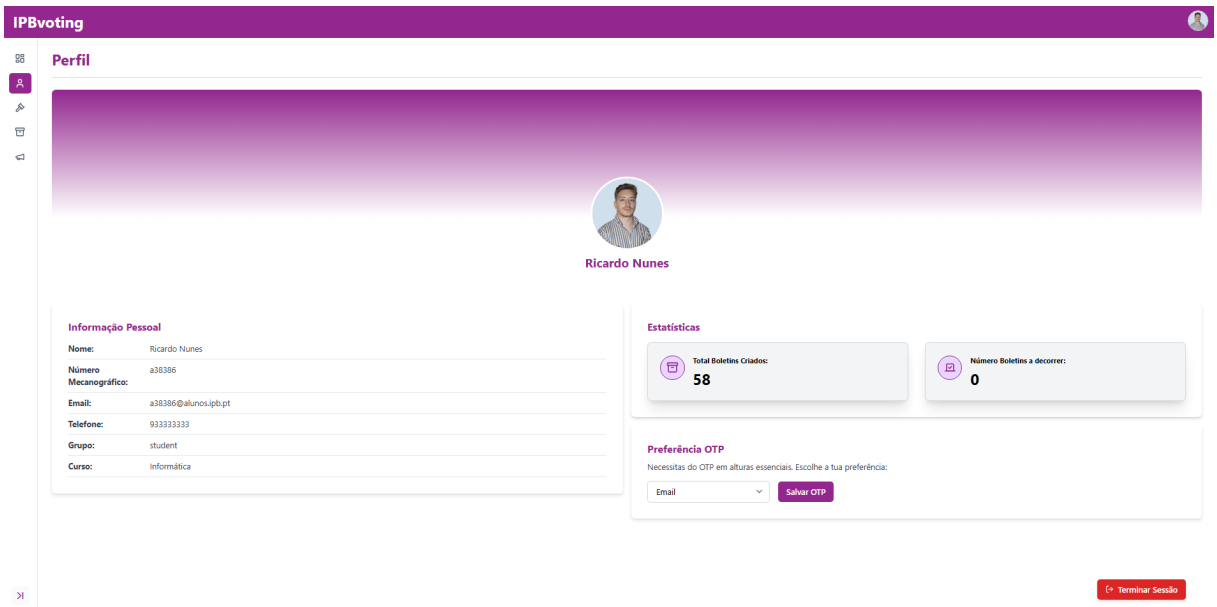

Figure 5.9: Profile

5.4.5 Ballot Feed

The Ballot Feed page serves as the main listing view for all accessible ballots in the system. Its primary purpose is to provide users with an overview of active, scheduled, and completed voting sessions, and to allow quick access to more detailed ballot information. Each ballot is presented using a reusable component called `VotingCard`, which is also shared with the "My Ballots" page for ballot owners.

The interface design prioritizes clarity and filterability. Users can toggle the visibility of ballots by state (Finished, Starting, Ongoing), perform searches, and control sorting preferences via the filtering panel. Additionally, voting statistics are summarized at the top of the page using the reusable `BallotStats` component, helping users quickly grasp the overall activity.

Ballots are fetched from the backend using an authenticated API call and enhanced with a computed field called `state`, which determines whether the ballot is ongoing, starting soon, or already finished. This value is dynamically calculated based on the current date and the ballot's `start_time` and `end_time`.

Calculating ballot state based on time:

```
const getBallotState = (ballot) => {
  const now = new Date();
  const startTime = new Date(ballot.start_time);
  const endTime = new Date(ballot.end_time);
  if (now < startTime) return 'Starting';
  if (now <= endTime) return 'Ongoing';
  return 'Finished';
};
```

Listing 23: Determine whether a ballot is starting, ongoing, or finished

The filtering logic allows dynamic updates when toggling checkboxes or submitting a search query. Search input is debounced using a custom React hook (`useDebounce`) to

avoid unnecessary API requests while the user is still typing. This enhances performance and reduces backend load.

The page includes pagination, showing a limited number of ballots per page and updating as the user navigates. Voting data is sorted based on both the state priority and timestamp, giving higher visibility to ongoing and upcoming votes.

Sorting and filtering ballots:

```
data.sort((a, b) => {
  const priA = STATE_PRIORITY[a.state];
  const priB = STATE_PRIORITY[b.state];
  if (priA !== priB) return priA - priB;
  const dateA = new Date(a.state === 'Starting'
  ? a.start_time : a.end_time);
  const dateB = new Date(b.state === 'Starting'
  ? b.start_time : b.end_time);
  return sortDirection === 'desc' ? dateB - dateA : dateA - dateB; }
```

Listing 24: Sort ballots by priority and timestamp after filtering

Each ballot is rendered using the `VotingCard` component, which adapts its layout and icons based on the ballot's properties and the user's role. This component visually indicates multiple states and options, such as:

- Whether the user has voted.
- Whether the ballot is private.
- Whether voting is anonymous.
- If results have been published or failed integrity verification.
- If the ballot supports joining via QR code.

These visual cues are implemented using conditional rendering and dynamic tooltips powered by the `Tippy.js` library. For example, different icons are shown depending on whether the user is a participant, the ballot has ended, or the vote integrity has failed.

Example of QR modal and voting state logic:

```
{isMyBallot && allowJoinViaQR && (currentState === 'Ongoing'
|| currentState === 'Starting') && (
  <button onClick={openQRModal}>
    <LucideQrCode />
  </button>)}
```

Listing 25: Conditional rendering of QR-code access inside VotingCard

The `VotingCard` is also responsible for routing users to different destinations depending on the context: ballot details, the vote screen, or the results page. These paths are conditionally computed based on the ballot state and whether the results are published.

Finally, a "Create Ballot" button is placed at the bottom-right of the page to allow quick access for users with ballot creation permissions. The page ensures smooth navigation, strong responsiveness, and modularity, built with Tailwind CSS and React component composition.

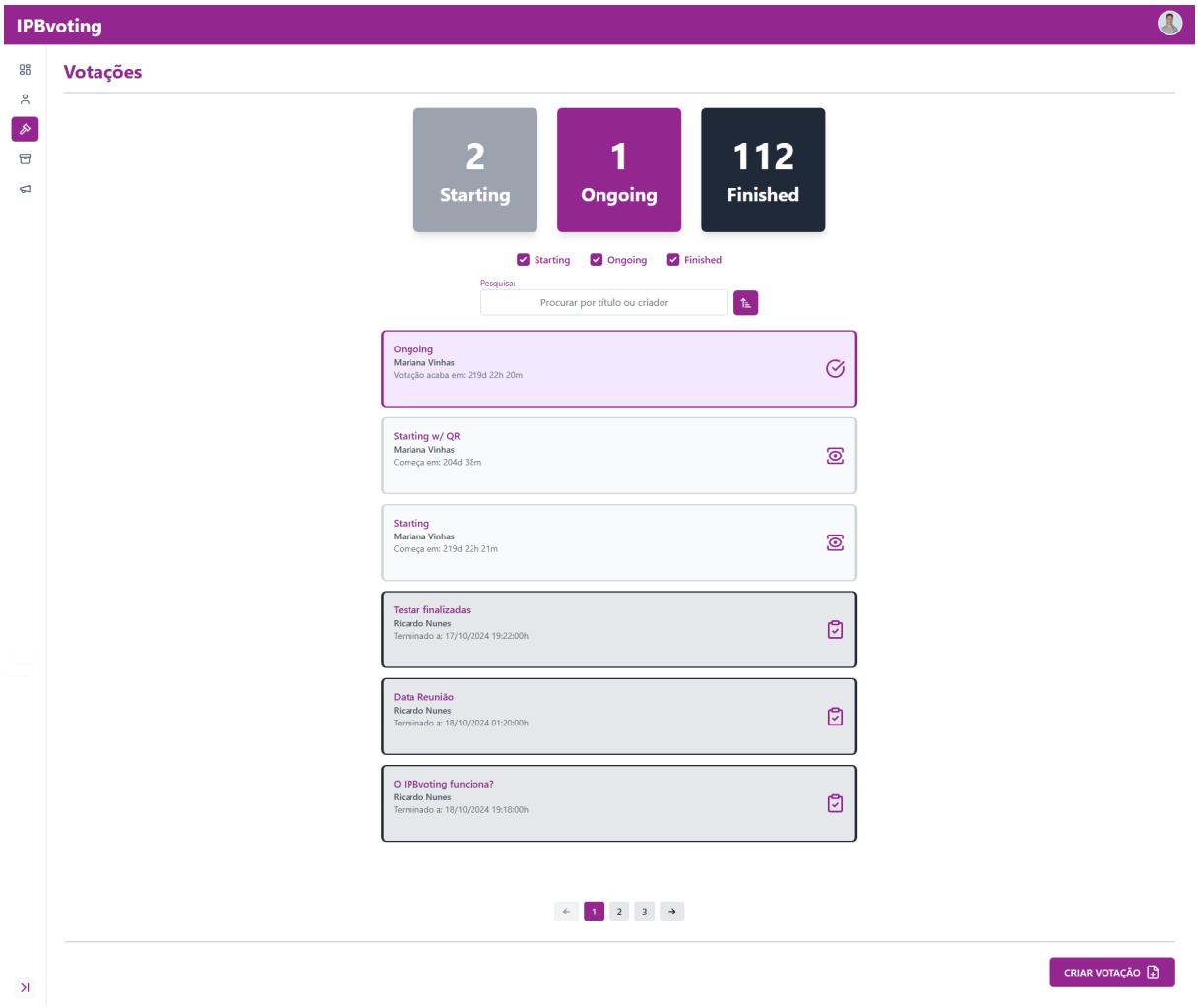

Figure 5.10: Ballot Feed

5.4.6 My Ballots

The My Ballots page was developed to give authenticated users a clear and organized overview of the ballots they have personally created. Functionally, it is very similar to the Ballot Feed page, reusing several key components such as `VotingCard`, `BallotStats`, `BallotFilters`, and `PaginationControls`. However, My Ballots is tailored to provide ballot owners with additional controls and feedback over their own voting sessions.

When this page is loaded, the application makes two authenticated requests: one to fetch the current user's metadata and another to retrieve all ballots created by that user.

After retrieval, each ballot is enriched with a computed `state` property that indicates whether the vote is "Starting," "Ongoing," or "Finished." This logic is consistent with the global feed:

Determine ballot state:

```
const getBallotState = (ballot) => {
  const now       = new Date();
  const startTime = new Date(ballot.start_time);
  const endTime   = new Date(ballot.end_time);
  if (now < startTime) return 'Starting';
  if (now <= endTime)  return 'Ongoing';
  return 'Finished';
};
```

Listing 26: Same time-window logic reused in the My Ballots page

Filtering, searching, and sorting logic follows the same flow as the Ballot Feed. Debounced input handling, sorting by state priority, and dynamic updates are all maintained. The only notable difference is that the data source is now scoped to the current user.

The component `VotingCard` is used for each listed ballot. When rendered in the context of My Ballots (through the `isMyBallot` flag), it displays additional management options:

- Trash icon to delete the ballot.
- Settings button to access the results or publish them.
- Status icons indicating QR participation, anonymity, and publication rules.
- Red alert if integrity verification failed.

Ballot deletion is handled through a confirmation modal and an API call. Upon success, the deleted ballot is removed from the local state and a success toast is shown.

This is demonstrated in the following snippet:

Deleting a ballot and updating state:

```
const handleDeleteSuccess = (deletedId, errMsg) => {
  if (deletedId) {
    setVotingData((prev) =>
      prev.filter((b) => b.ballot_id !== deletedId));
    toast.success('Ballot deleted successfully!');
  } else if (errMsg) {
    toast.error(errMsg);
  }
};
```

Listing 27: Callback to update UI after successful ballot deletion

The presence of advanced controls and visual icons in `VotingCard` makes it easier for users to manage their ballots without needing to navigate into each one. Ballots with pending result publication, failed vote integrity, or upcoming start times are all visually marked to ensure proactive management.

Pagination is available to limit the number of displayed cards, and filters help the user isolate specific ballot types. The user experience remains consistent across devices thanks to Tailwind CSS and responsive layout design.

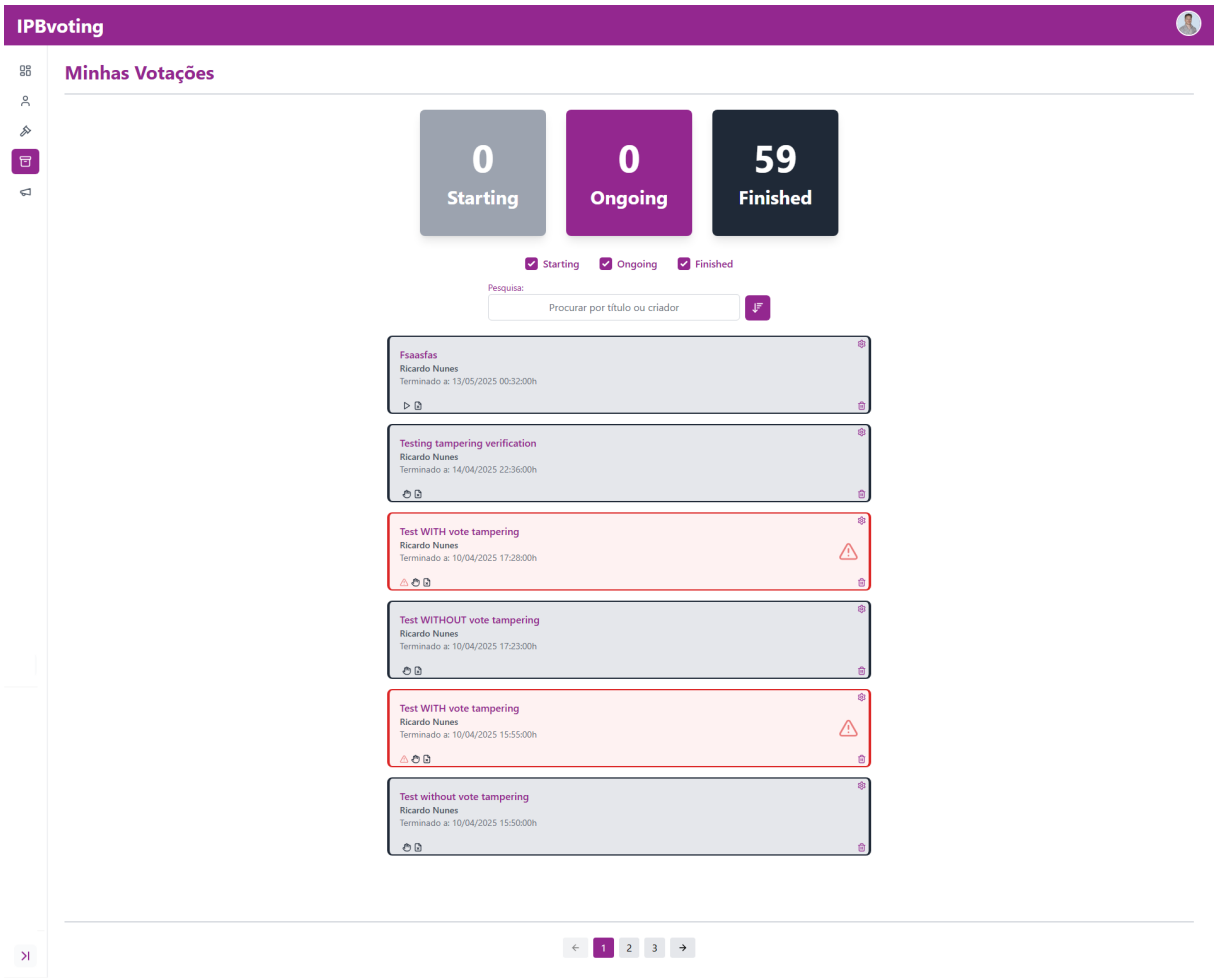

Figure 5.11: My Ballots - Showing user-created ballots and their statuses

5.4.7 VotingCard Component

The `VotingCard` is a highly dynamic, context-aware card that represents a single ballot both in the global *Ballot Feed* and in the user-specific *My Ballots* view. Although the same React component is reused, its appearance and available actions adapt according to two main factors:

- **Ballot state** (*Starting*, *Ongoing*, *Finished*) and integrity status.
- **Viewer role** (creator viewing their own ballot ⇒ `isMyBallot=true` vs. participant/observer).

Exterior colour and background

The card's outline and background tint communicate the ballot's status at a glance. A detailed mapping of status colours is provided in Appendix B.7 (Table B.7).

The border colour is computed at runtime, a condensed version of the logic is shown below:

Dynamic border colour based on state and integrity

```
const borderColor = integrityFailed ? 'red'
  : currentState === 'Ongoing'  ? 'ipbColor'
  : currentState === 'Finished' ? 'gray-800'
  : 'gray-300';  // Starting
```

Listing 28: Compute CSS border colour for a VotingCard at runtime

Informational icons – bottom-left

These icons provide a compact summary of each ballot's configuration, including its anonymity, visibility, integrity status, and result publication rules. They are shown only to the ballot creator and are accompanied by tooltips using `Tippy.js` for clarity. A full list of icons and their meanings is available in Appendix B.3.

Action buttons – centre-right

This section shows contextual buttons based on the viewer and the ballot state. Voters may see a "Vote Now" button, a link to results, or simply a "View Only" label, depending on whether voting is active and their participation status. The complete mapping is detailed in Appendix B.4.

Administrative actions – My Ballots

In the **My Ballots** view, ballot creators have additional actions available, such as manually publishing results or displaying the QR code for participant onboarding. These

options are icon-driven and summarized in Appendix B.5.

Persistent owner controls

All ballots created by the logged-in user display persistent management controls — a settings icon in the top-right corner and a delete button in the bottom-right. These ensure quick access to editing and deletion actions. See Appendix B.6 for their visual representation.

Visual Comparison

Figure 5.12 presents a standard *Ongoing* card from the *Ballot Feed* page. Figure 5.13 illustrates an example of a finished ballot displayed in the *My Ballots* page, while Figure 5.14 depicts a ballot flagged for integrity failure, highlighted in red and accompanied by a pulsing alert icon.

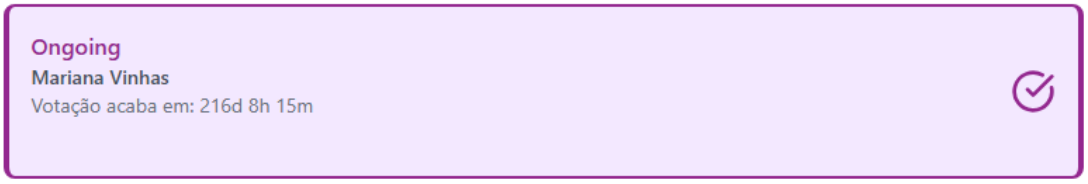

Figure 5.12: Ballot Feed - Voting Card (Ongoing)

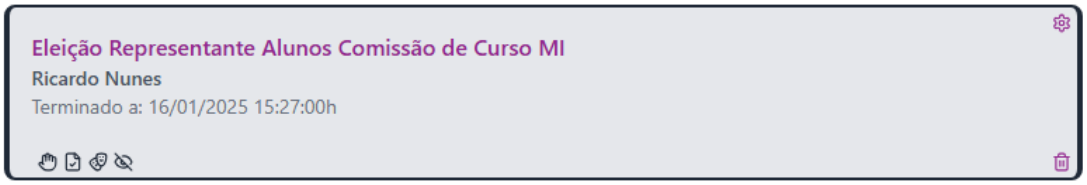

Figure 5.13: My Ballots - Voting Card (Finished)

Figure 5.14: My Ballots - VotingCard (Integrity Failure)

This flexible design, driven by the `isMyBallot` flag and a concise set of props, the same component can operate in two very different contexts while maintaining a consistent user experience and minimizing code duplication.

5.4.8 Ballot Type Selection

After clicking the *Create Ballot* button on the *Homepage* or *Ballot Feed* page, users are redirected to a ballot type selection screen. This page serves as the entry point for creating a new vote and allows the user to choose between two different voting formats: ***Poll*** or ***Election***.

Interface and UX

The design of the selection screen emphasizes clarity, accessibility, and responsive behavior. On large screens, both options are displayed side-by-side with a vertical divider and the label *OR* between them. On smaller devices, the options stack vertically with a horizontal divider instead. This ensures a seamless user experience across devices.

Each option is presented as an interactive card with a colored gradient background using IPB color (#93278F) combined with a secondary tone, an illustrative image to help visually differentiate the options, a short description explaining the use case for each voting type and a hover effect to indicate interactivity.

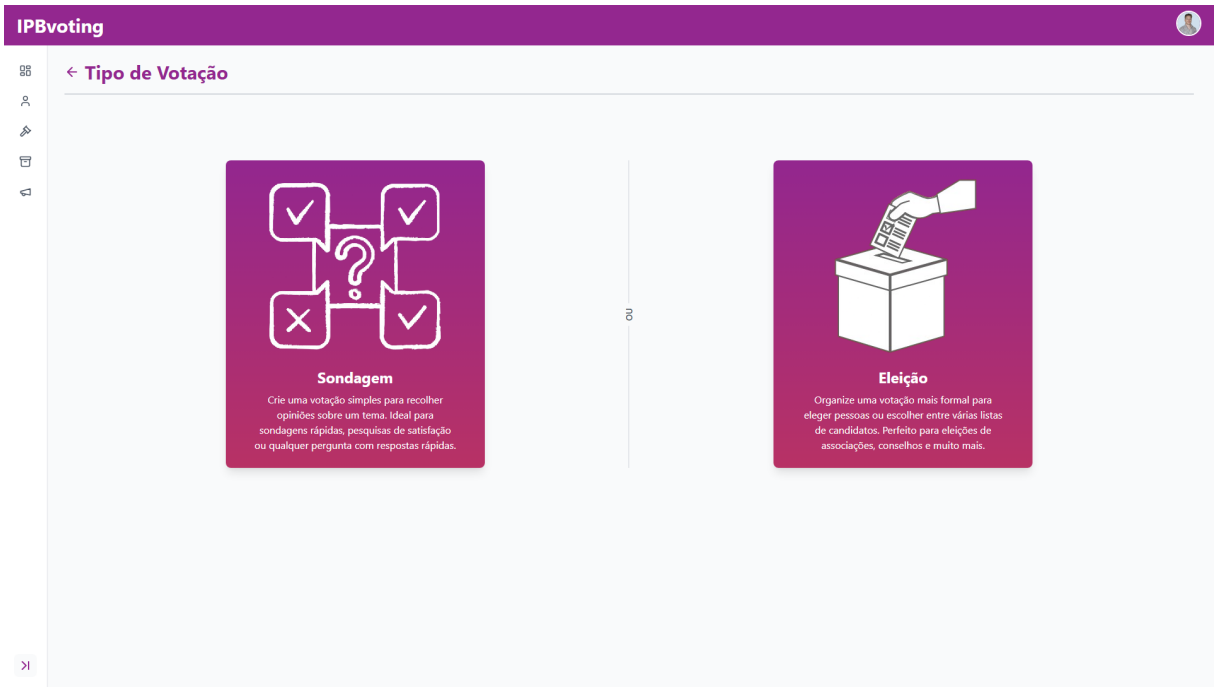

Figure 5.15: Ballot Type Selection Screen

Routing logic

Each card is clickable and triggers a route change using React Router's `useNavigate` hook. Depending on the selection, the user is redirected to a specific form flow for either a poll or an election ballot:

Route navigation on option click:

```
onClick={() => navigate('/create_ballot/poll')}
// or
onClick={() => navigate('/create_ballot/election')}
```

Listing 29: Routing logic triggered when selecting a ballot type.

Use case distinction

The purpose of distinguishing between these two ballot types is to tailor the creation process and data structure accordingly:

- **Poll** is used for lightweight, opinion-based votes such as satisfaction surveys, quick polls, or simple questions with predefined answers.
- **Election** is suitable for formal decision-making processes involving the selection of individuals or lists (e.g., electing student representatives or association boards).

By making this distinction early in the user journey, the system can guide the user through an appropriate configuration flow while preserving a consistent and intuitive UX.

5.4.9 Ballot Creation Flow

The `BallotCreationFlow` component manages the full multi-step process of setting up a new ballot. It supports two types of voting: *polls*, for simple opinion gathering, and *elections*, for selecting candidates from predefined lists. This flow consists of two subpages: `BallotCreation` (ballot details and options) and `BallotMembers` (participant selection), connected using React Router's nested routing system.

Dynamic logic based on ballot type

The core logic of the flow is driven by the `ballotType` prop passed from the previous selection page (Figure 5.15). Upon loading, this type is stored inside the shared `ballotData` state and is used to configure the form fields accordingly. The logic is initialized via a `useEffect` hook:

Synchronizing ballot type and default structure

```
useEffect(() => {
if (ballotType === 'election') {
setBallotData(prev => ({
...prev,
ballot_type: 'election',
options: prev.options.map(o => ({
...o, candidates: o.candidates || [{ name: '' }]
```

```
}))
}));
} else {
setBallotData(prev => ({
...prev,
ballot_type: 'poll',
options: prev.options.map(o => ({ ...o, candidates: [] }))
}));
}
}, [ballotType]);
```

Listing 30: Initialize ballot options and structure based on selected type

This ensures that polls require only option labels, while elections display additional candidate selection fields under each list. Both flows share the same component logic but adaptively render based on the value of `ballot_type`.

User input: Poll vs Election

In a **Poll**, the user must:

- Define a title and description for the ballot.
- Enter at least two option labels (e.g., “Yes” / “No”).
- Choose how many options a voter can select.
- Configure anonymity, blank vote allowance, and publication rules (automatic/manual publication, start/end timestamps, and an optional file attachment).

In an **Election**, the user sees the same inputs, but each option represents a list and includes:

- One or more candidate selectors per list.
- Candidate validation to ensure proper user selection.

Candidate fields are handled with a dedicated `CandidateSelector` component. An election ballot cannot proceed until every list has at least one valid candidate selected.

Poll ballot creation screen

Figure 5.16 shows the interface for creating a standard poll. Note how the options are simple text inputs, with no extra fields per option.

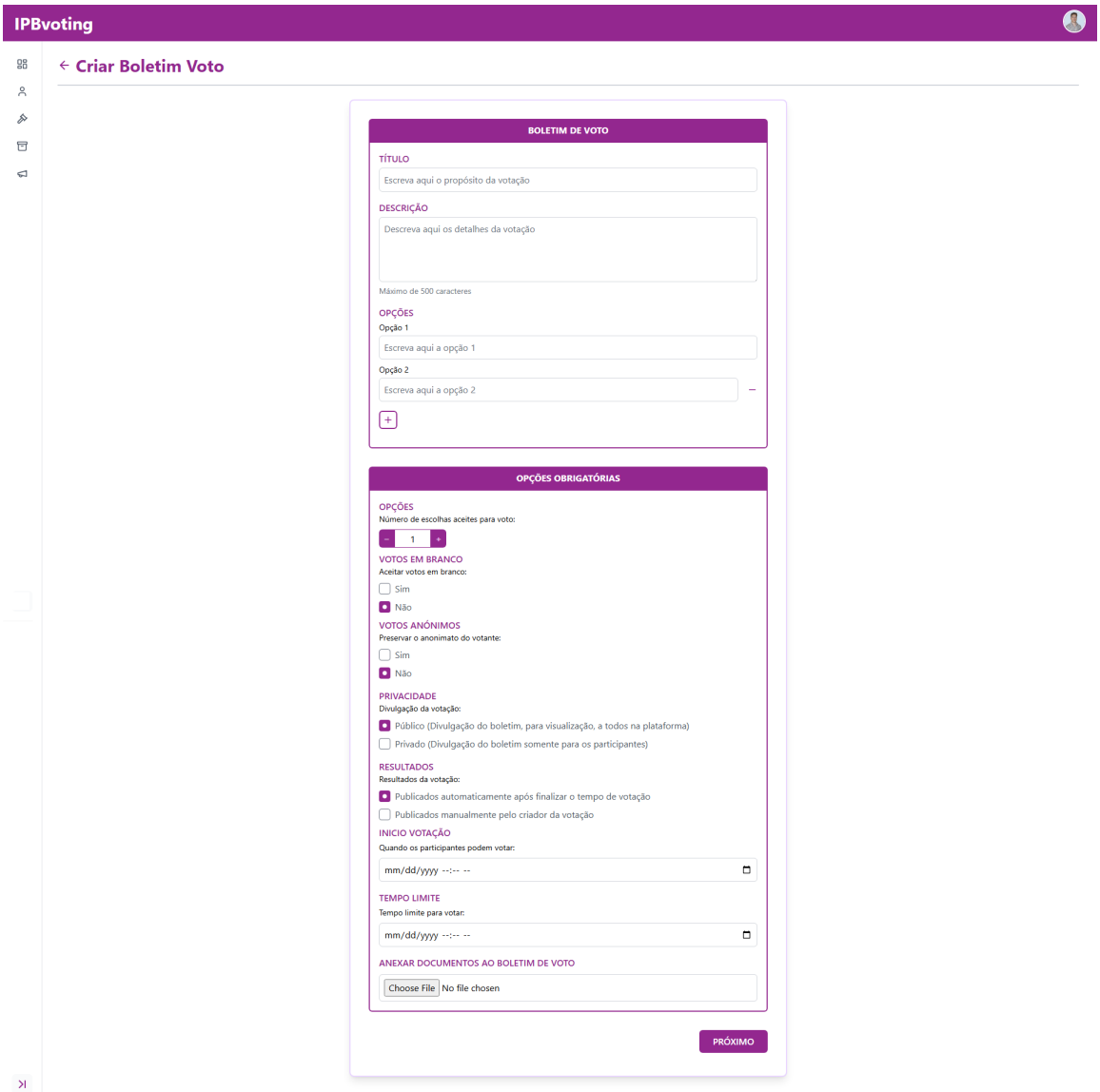

Figure 5.16: Ballot Creation - Poll

Election ballot creation screen

In contrast, Figure 5.17 illustrates the ballot creation flow for an election. Each list (option) includes a set of candidates, where each must correspond to a valid platform user.

These selections are made through a dynamic dropdown powered by the `CandidateSelector` component, with logic to avoid duplicate assignments.

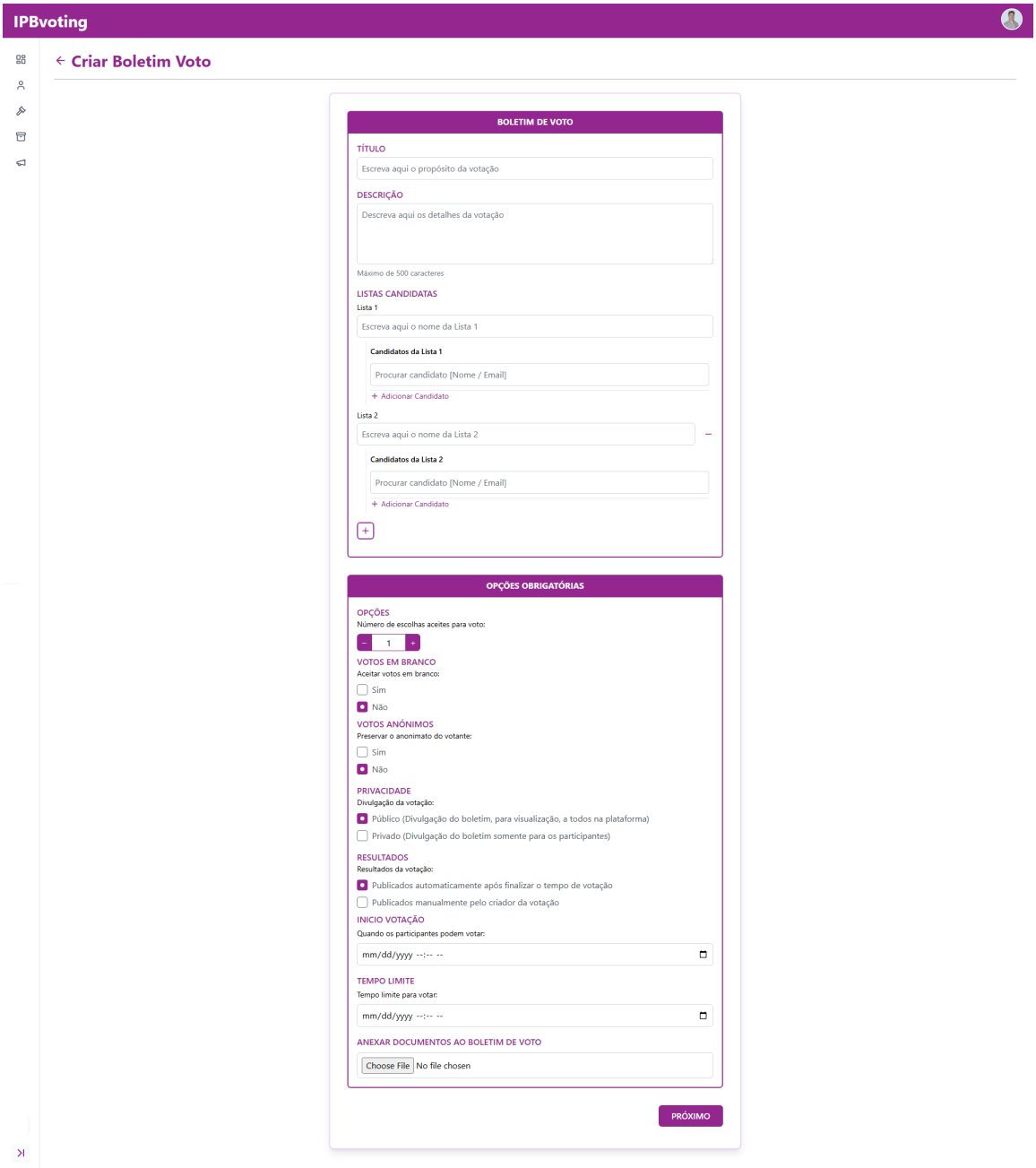

Figure 5.17: Ballot Creation - Election

A detailed overview of the fields, their expected behavior, and validation rules is provided in Appendix B.2.

Flow transition and validation

Once all inputs are valid and the "Next" button is clicked, the component:

1. Validates that the start date is in the future and before the end date.
2. Ensures that each list (in election mode) has at least one selected candidate with a valid `user_id`.
3. Transforms local time inputs into properly formatted `ISO 8601` strings for backend submission.

Participant selection screen

The second and final step of the creation flow is handled by the `BallotMembers` component. It allows the user to define who is eligible to vote in the ballot, either by selecting participants individually or by enabling open access via QR code. This page ensures proper validation and prepares all participant-related data to be submitted with the final ballot.

Modes of participation: At the top of the screen (Figure 5.18), the user can optionally check a box to allow voters to join the ballot using a QR code. This is useful when organizers prefer sharing a join link rather than selecting participants manually.

Participant search and commands: The input field supports two modes:

- **Free search:** users can type names or emails, with autocomplete suggestions fetched dynamically from the backend. Matching users can be clicked to be added.
- **Special commands:** typing `@` reveals bulk selection options, such as `@everyone`, `@student`, or `@course:Informática`, which add entire user groups at once.

These options are provided through a predefined list of commands and resolved through API calls to fetch matching user accounts.

Participants table and management: Once users are added, they appear in a paginated table showing their avatar, full name, Email address and Role/group (e.g., student or professor).

Each participant can be removed individually, and a button in the header allows removing all participants at once. Pagination controls appear if the number of participants exceeds 7, ensuring a consistent layout regardless of list length.

Final confirmation: When the user clicks the "Confirm" button, the system performs validation:

- At least one participant must be selected, unless the "Join via QR" option is enabled.
- Participant list is serialized into IDs for backend submission.
- All previously entered data (from `BallotCreation`) is merged with the current participants list and sent to the backend via a `multipart/form-data` POST request.

Upon success, the user is redirected back to the Ballot Feed and receives a confirmation toast.

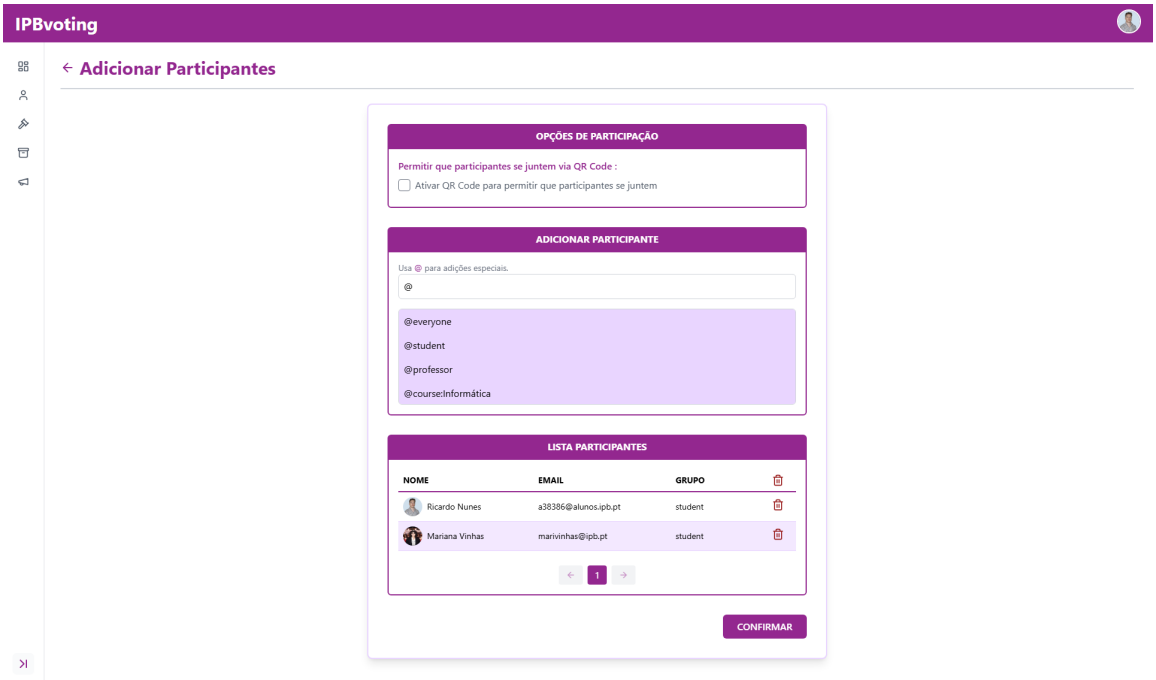

Figure 5.18: Ballot Participant Selection Screen

5.4.10 Voting Process Flow

Once a ballot is successfully created and confirmed, it becomes visible on the *Ballot Feed* of each selected participant. Initially, if the scheduled `start_time` has not yet arrived, the ballot is in the **Starting** state. During this phase, the `VotingCard` displays a countdown until voting opens (Figure 5.19).

Figure 5.19: VotingCard - `Starting` state, showing countdown until voting opens

Clicking the card leads to the `BallotDetails` page (Figure 5.20), where users can review the options, candidates (if applicable), and attached files. However, voting actions are disabled at this stage.

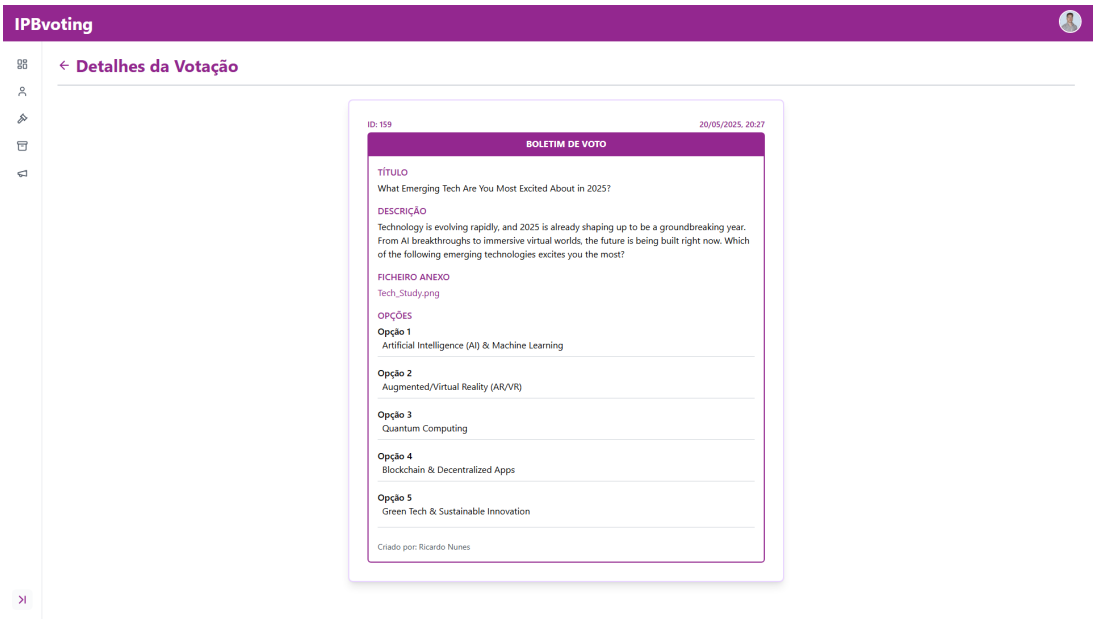

Figure 5.20: Ballot Details Screen - While voting is not yet open

Ballot becomes active: `Ongoing`

Once the `start_time` is reached, the ballot automatically transitions to the **Ongoing** state. The `VotingCard` updates its border color and action button accordingly (Figure 5.21).

Figure 5.21: VotingCard - `Ongoing` state, displaying the *Vote Now* button.

By clicking the `Vote Now` button, the user is directed to the dedicated voting page (Figure 5.22). This screen allows selection of options, while respecting the ballot's configuration (e.g., number of allowed choices, whether blank votes are accepted, etc.).

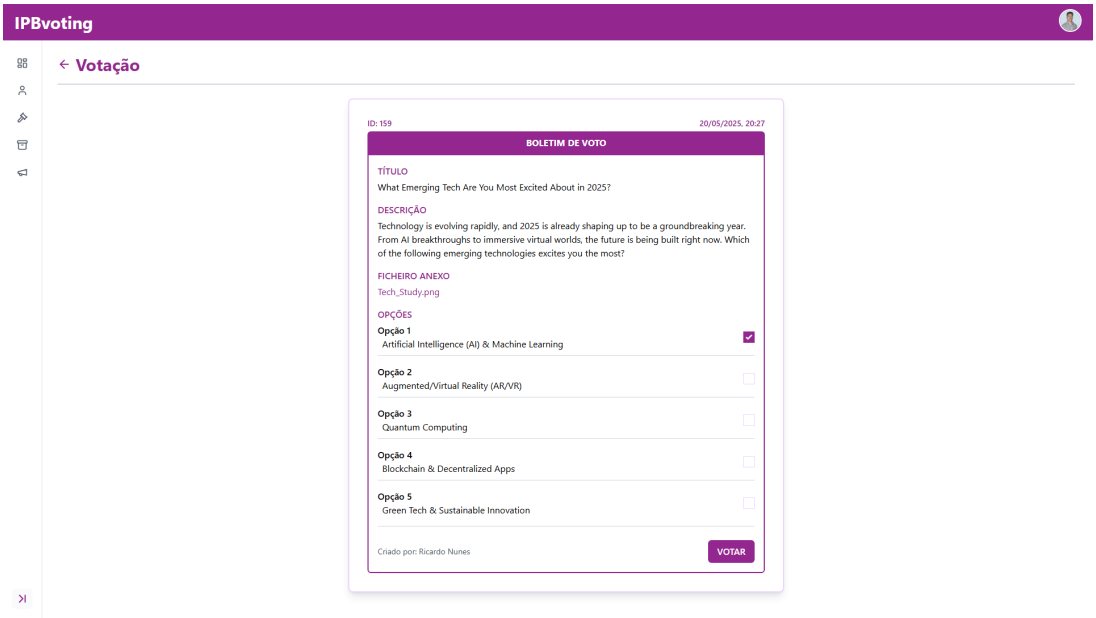

Figure 5.22: Ballot Voting screen

If the ballot is an election, each option expands into a candidate list. For polls, only the option text is shown. The system automatically disables checkboxes that exceed the configured maximum.

Vote submission and OTP security

After choosing the desired options, the user clicks the `Vote` button. This action triggers the first step of the secure two-phase submission: a backend API call initiates the vote and sends a one-time code to the voter's email.

Initial vote submission to trigger OTP

```
await api.post(/ballots/${ballotId}/vote, {
option_ids: selectedOptions,
is_blank: selectedOptions.length === 0,
});
```

Listing 31: Trigger OTP after local vote selection

The user is then prompted with a modal to input the six-digit OTP (Figure 5.23).

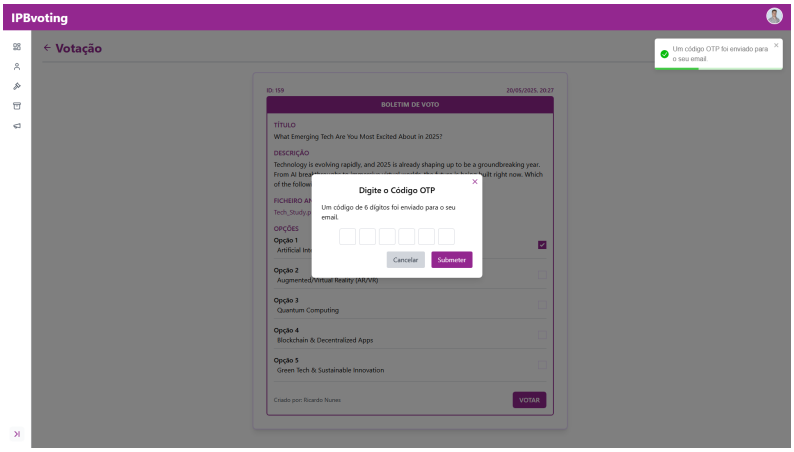

Figure 5.23: OTP Verification Modal

Only after correctly entering the OTP code, the final vote is submitted and securely recorded.

Final vote submission with OTP code

```
await api.post(/ballots/${ballotId}/vote, {
option_ids: selectedOptions,
is_blank: selectedOptions.length === 0,
otp_code: enteredOtp,
});
```

Listing 32: Final vote submission with OTP validation

This dual-step process reinforces security and ensures that only legitimate participants can cast a vote, even if their session or device is compromised.

5.4.11 Results Page

Once a ballot has ended and its results have been published (either automatically or manually by the creator), participants are allowed to view the final outcome through the `ResultsPage` component.

This page provides a clear and visual summary of how each option performed during the voting process. The layout adjusts dynamically depending on whether the ballot is a simple *poll* or an *election* involving candidate lists.

Results Overview

The page fetches data from the endpoint `/ballots/:id/results` and renders the following information:

- Title, description, and creation options of the ballot.
- All voting options, each showing:
  - Total votes received.
  - Percentage share (a pie chart visualization of the voting results, where each segment represents the proportion of votes received by each option.).
  - Candidate list (for elections).
- A clear warning if vote integrity verification has failed.

If the ballot's results are not yet published or the vote has not ended, the system displays a message indicating that results are not available.

Poll Results Display

Figure 5.24 illustrates the results interface for a simple poll. Each option is a text label, and the votes are displayed with a pie chart:

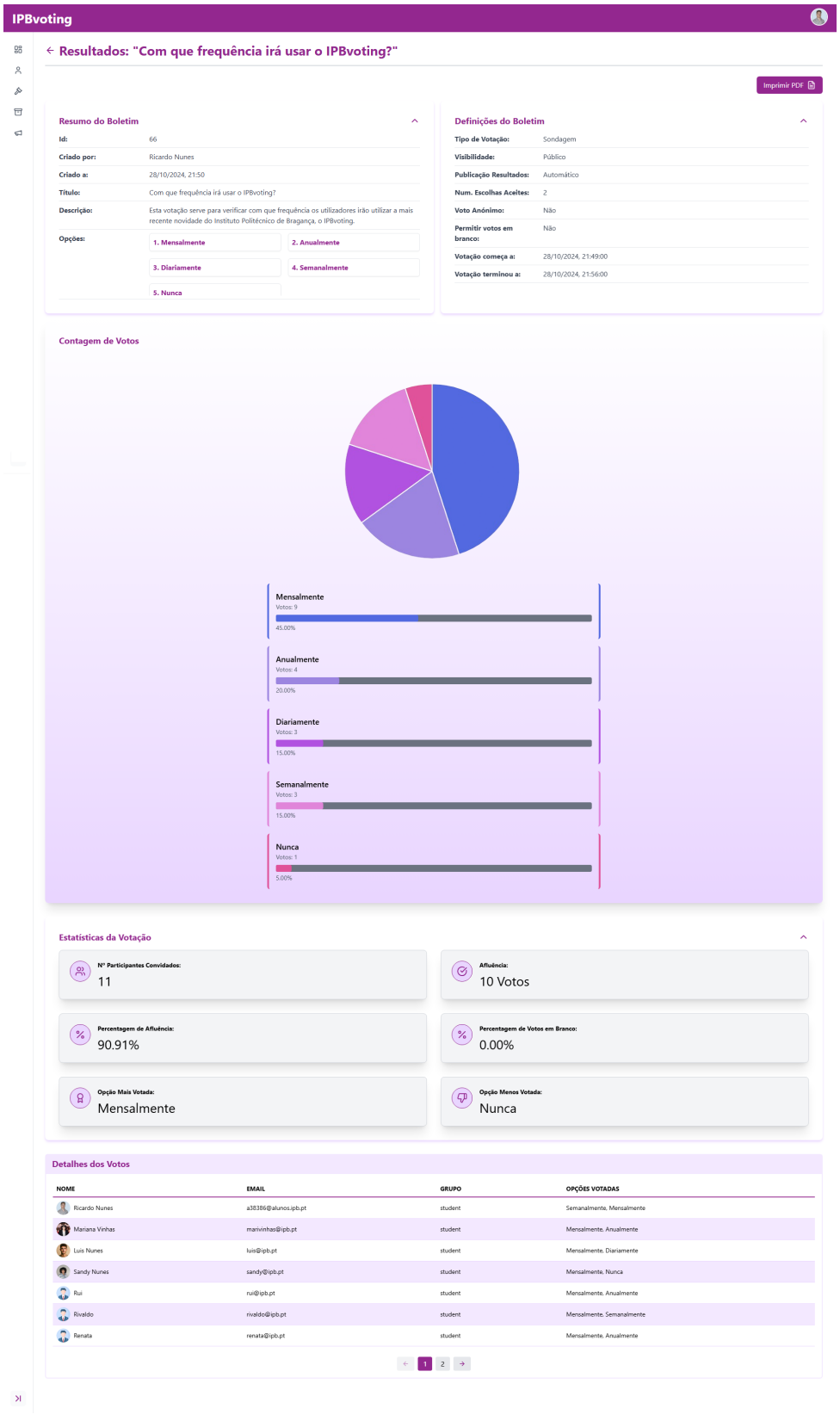

Figure 5.24: Results interface - Poll

Election Results Display

In election ballots, each voting option represents a candidate list, typically identified by a letter or short name (e.g., List A, List B). While the pie chart displays the distribution of votes by list, it does not include the individual candidate names. Instead, the candidates belonging to each list are shown separately within the voting interface, grouped under their respective list title. This approach emphasizes the list as the unit of voting while still providing visibility into its members.

Vote Integrity Validation

Before rendering the results, the frontend checks the `integrity_failed` flag returned by the backend. This flag is computed using a `hash-chain` mechanism to detect any tampering in the vote sequence. If validation fails, the page blocks result display and shows a red warning message instead.

Frontend Logic for Result Rendering

The example below shows the key logic used to dynamically render options and their vote percentages using JSX. The percentage is calculated on the backend and styled using Tailwind classes:

Rendering result bars based on vote percentage

```
{ballotData.options.map((option, index) => (
  <div key={option.option_id} className="mb-4">
    <p className="font-semibold text-ipbColor">
      Option {index + 1}: {option.option_text}
    </p>
    <div className="w-full bg-gray-200 rounded-full h-4 mt-1">
      <div
        className="bg-ipbColor h-4 rounded-full"
```

```
        />
      </div>
      <p className="text-sm text-gray-600 mt-1">
        {option.vote_count} votes ({option.percentage.toFixed(1)}%)
      </p>
    </div>
))}
```

Listing 33: React snippet to render result progress bars

This visualization component is shared between polls and elections, enabling consistent UI while supporting both ballot types. For elections, each option block includes a nested display of the associated candidates and their avatars.

5.4.12 Notifications Page

The `Notifications` page acts as a personal event feed for each user, providing clear visibility into key activities related to their ballots, such as being added as a participant, casting a vote successfully, or being alerted when a ballot starts or ends. These notifications are grouped by date and displayed in a timeline-like structure.

Live filters and pagination

The top portion of the page provides users with tools to filter their notification history by date range, perform keyword searches (ballot title or ballot creator name), and sort by newest or oldest events. These filters are reactive and update results in real time using the `useEffect` hook:

Reactively fetch notifications when filters change

```
useEffect(() => {
  const fetchData = async () => {
    const params = {
      ...(fromDate && { from_date: adjustForTZ(fromDate) }),
      ...(toDate   && { to_date  : adjustForTZ(toDate) }),
      sort_direction: sortDirection,
      ...(searchQuery && { search: searchQuery }),
    };
    const { data } = await api.get('/notifications', { params });
    setNotifications(data);
    setCurrentPage(1);
    await api.post('/notifications/mark_as_read');
  };
  fetchData();
}, [fromDate, toDate, sortDirection, searchQuery, dateError]);
```

Listing 34: Automatically refetch filtered notifications and mark them as read

The notifications are paginated 20 per page, with intuitive navigation and a grouped-by-date display. Dates appear on the left in a vertical timeline format, with each notification indented to form a chronological narrative of events.

Notification structure and styling

Each notification entry includes a timestamp, dynamic message content, and conditional styling depending on whether the notification was marked as read. Unread notifications pulse with a red icon to draw attention (Figure 5.25).

Types of supported notifications include:

- **Participant Added:** The user was invited to a new ballot.

- **Vote Successful:** A confirmation that their vote was submitted.
- **Ballot Started:** A vote they are part of has just begun.
- **Ballot Ended:** The vote has concluded and results may now be available.

The following frontend logic determines what message to display based on the notification type:

Determine message content by notification type

```
switch (n.notification_type) {
  case 'participant_added':
    content = `${creator} adicionou-o como participante em ${title}`;
    break;
  case 'vote_successful':
    content = `O seu voto em ${title} foi registado com sucesso`;
    break;
  case 'ballot_started':
    content = `${title} começou. Vá exercer o seu direito de voto.`;
    break;
  case 'ballot_ended':
    content = `${title} terminou. Verifique os Resultados.`;
    break;
  default:
    content = 'Notificação';
}
```

Listing 35: Frontend logic to personalize notification messages

Visual example

Figure 5.25 presents the interface in use, with notifications grouped by day, showing both read and unread states.

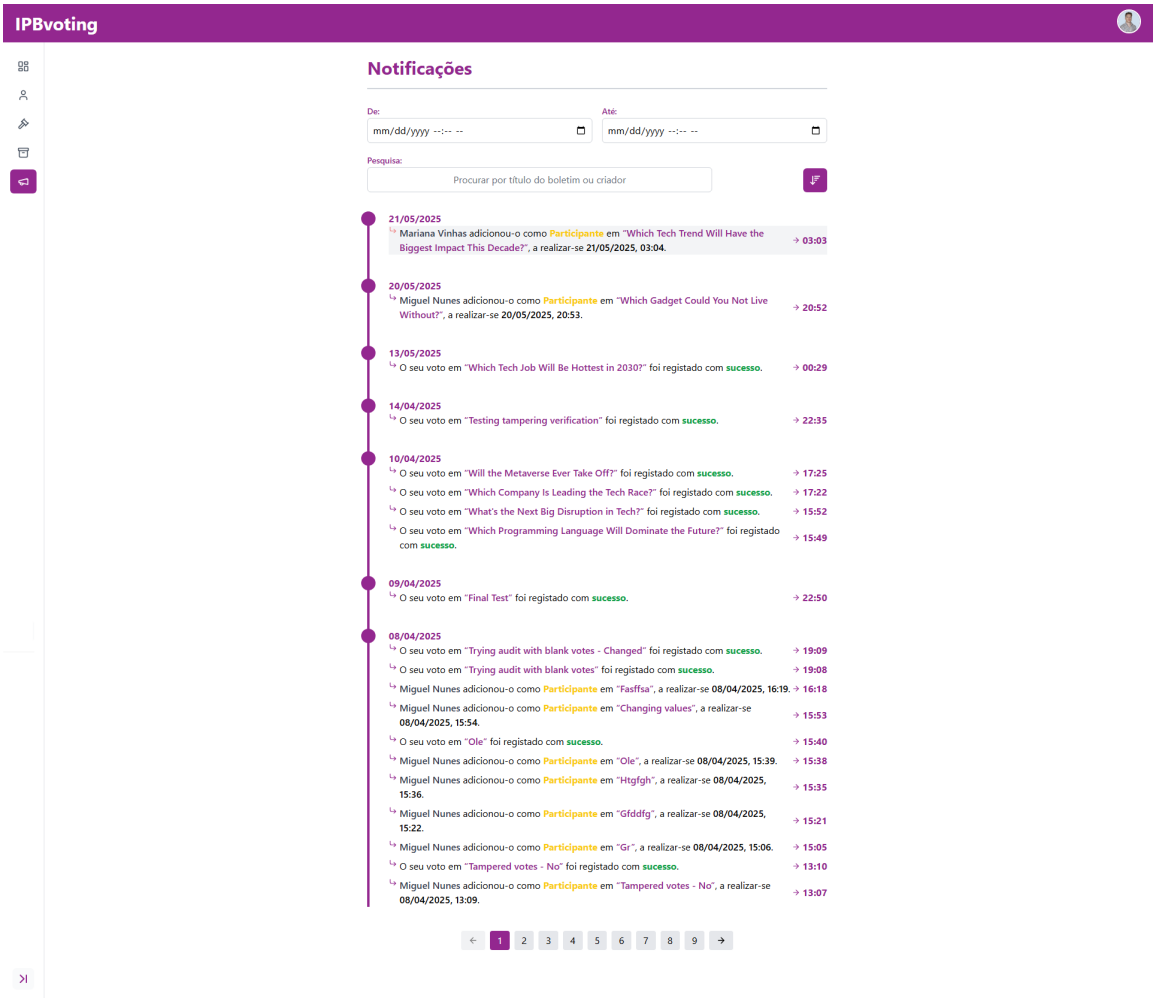

Figure 5.25: Notification feed

This timeline approach balances usability and clarity, ensuring users can quickly grasp recent ballot-related activity while also searching past events if needed.

5.4.13 Not Found Page

The `NotFound` component is rendered whenever a user navigates to a route that does not exist. It provides a user-friendly fallback for broken links, outdated bookmarks, or incorrect URLs.

The page layout maintains the platform's design consistency by including the persistent `Navbar` and `Sidebar`, ensuring users remain oriented even in error states. The image used is rendered responsively.

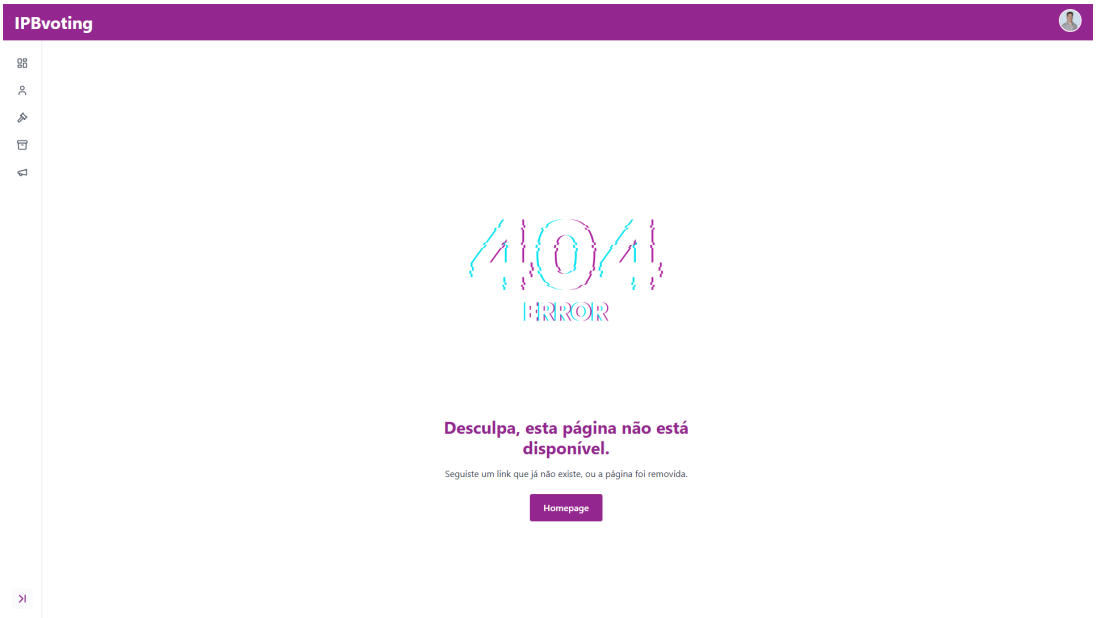

Figure 5.26: Custom 404 – Not Found page

This page plays a small but important role in ensuring smooth user experience even during navigation errors. It helps maintain trust and offers a quick recovery path instead of showing a default browser error screen.

Rendering fallback when no route matches

```
<Route path="*" element={<NotFound />} />
```

Listing 36: 404 error for all unknown paths

Chapter 6

Conclusions

This dissertation presented the design and implementation of an electronic voting (e-voting) platform tailored to academic institutions. The system was developed in response to the limitations of traditional voting methods, such as inefficiency, lack of transparency, and barriers to participation. Through the adoption of modern web technologies, the platform aimed to modernize academic voting by ensuring accessibility, integrity, and real-time engagement among its users.

The core objectives defined at the outset of this work have been largely accomplished. The implemented system supports secure user authentication, flexible creation of polls and elections, and vote submission with a verification layer based on OTP codes. Additional features such as automatic and manual result publication, a hash-chain mechanism to preserve vote integrity, and a dynamic notification system were successfully incorporated and mapped to well-defined functional requirements.

From a usability standpoint, the UI mockups created in Figma provided a structured visual blueprint that informed the frontend development in React. The application was designed to accommodate both technical and non-technical users, with special attention to inclusivity, accessibility, and responsive design. This aligns closely with the project's stated goal of fostering engagement and democratic participation within academic communities.

The backend, built in FastAPI and supported by a PostgreSQL database, included

robust mechanisms for data consistency and traceability. Hash-based integrity verification, immutable vote records, and layered access control were implemented to satisfy key non-functional requirements related to security and transparency. Additionally, modularity and API design enable the system to be extended or integrated with other services in the future.

Despite these achievements, one notable limitation concerns the absence of systematic testing under realistic conditions. While the system was architected with scalability in mind, formal stress testing, benchmarking, and user-centered usability evaluations were not conducted. These omissions prevent full factual validation of certain non-functional requirements.

Given these limitations, there are several promising directions for future work. Tests, load simulations, and usability evaluations would strengthen the platform's reliability and ensure continued compliance with functional and non-functional requirements. The introduction of role-based dashboards and analytics could further support decision-making by enhancing data visibility for administrators and participants.

Finally, integration with institutional systems such as LDAP directories could enhance authentication and administration processes.

In summary, this project demonstrates that a secure, user-friendly, and extensible e-voting system for academia is not only feasible but also practical. While future enhancements are essential to scale and validate the system under real-world conditions, the foundational architecture and feature set developed in this work provide a solid basis for trustworthy digital elections in educational environments.

Bibliography

[1] *EVoting*, https://evoting.com/en/plataformas/euniversidades/, Accessed: 2024.

[2] *ElectionBuddy*, https://www.electionbuddy.com/, Accessed: 2024.

[3] *Voatz*, https://www.voatz.com/, Accessed: 2024.

[4] *Simply Voting*, https://www.simplyvoting.com/, Accessed: 2024.

[5] *X2Vote*, https://www.x2vote.com/pt, Accessed: 2024.

[6] J. M. Fernandes and R. J. Machado, *Requirements in Engineering Projects*, 1st. Cham, Switzerland: Springer International Publishing, 2016, ISBN: 9783319292020.

[7] K. Wiegers and J. Beatty, *Software Requirements*, 3rd. Redmond, WA, USA: Microsoft Press, 2013, ISBN: 9780735679665.

[8] *Git – Distributed version control system*, https://git-scm.com/, Accessed: 2024.

[9] *React – A JavaScript library for building user interfaces*, https://reactjs.org/, Accessed: 2024.

[10] *FastAPI – The web framework for building APIs with Python*, https://fastapi.tiangolo.com/, Accessed: 2024.

[11] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," Internet Engineering Task Force, RFC 7519, 2015, Defines the compact security token used in this project. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7519.

[12] N. Provos and D. Mazieres, “A future-adaptable password scheme,” in *Proceedings of the 1999 USENIX Annual Technical Conference*, 1999, pp. 81–92.

[13] *PostgreSQL 17 Documentation*, Part II, Ch. 5 “Data Definition” and Part V, Ch. 43 “Triggers” give the syntax and examples referenced in this dissertation, The PostgreSQL Global Development Group, 2024. [Online]. Available: `https://www.postgresql.org/docs/17/`.

Appendix A

Requirements Tables

A.1 Functional Requirements

ID	Description	Priority
FR1.1	Provide secure login using JWT authentication.	High
FR1.2	Store passwords securely using bcrypt hashing.	High
FR2.1	Allow creation of polls and elections.	High
FR2.2	Enable configuration of start and end dates.	High
FR2.3	Allow setting of visibility, anonymity, and publication mode.	High
FR3.1	Support searching and adding participants manually.	Medium
FR3.2	Support adding participants using @commands.	Medium
FR3.3	Support adding participants via QR code.	Medium
FR4	Generate a SHA-256 hash chain for every vote cast.	High
FR5.1	Send a 6-digit OTP to the voter before vote submission.	High
FR5.2	Accept and verify OTP before persisting the vote.	High
FR6.1	Support automatic or manual publication of results.	Medium
FR6.2	Make published results visible to eligible voters.	Medium
FR6.3	Allow downloading of results as a PDF file.	Medium

FR7.1	Calculate and display percentage of votes per option.	Medium
FR7.2	Render results using pie chart.	Medium
FR8.1	Notify users of new voting invitations.	Medium
FR8.2	Notify users when a ballot starts.	Medium
FR8.3	Notify users when a ballot ends.	Medium
FR8.4	Notify users upon successful vote submission.	Medium
FR9	Provide an endpoint to verify chain integrity (OK / FAIL).	High

Table A.1: Functional requirements

A.2 Non-Functional Requirements

ID	Description	Category
NFR1	Support up to 10,000 concurrent voters without degrading responsiveness.	Performance
NFR2	Ensure median API response times stay below 200 ms under expected load.	Performance
NFR3	Enable horizontal scalability for backend services (FastAPI).	Performance
NFR4	Ensure database write operations during voting stay below 100 ms.	Performance
NFR5	Enforce HTTPS with TLS 1.3 for all client-server communication.	Security
NFR6	Encrypt votes and personal data at rest using AES-256.	Security
NFR7	Prevent tampering by using vote hash-chaining with SHA-256.	Security
NFR8	Maintain 99.5% uptime during election periods.	Reliability
NFR9	Recover from failure in under 60 seconds via PostgreSQL WAL + backups.	Reliability
NFR10	Ensure the user interface supports clear color contrast, full keyboard navigation, and compatibility with screen readers.	Usability
NFR11	UI must be fully functional on mobile, tablet, and desktop screen sizes.	Usability
NFR12	Support accessibility tools (e.g., screen readers, zoom) without breaking layout.	Usability
NFR13	Ensure user data processing complies with GDPR (e.g., consent, right to deletion).	Legal

Table A.2: Expanded Non-functional Requirements

A.3 User Stories

ID	Actor	User Story	Acceptance Criteria	Priority
USU1	User	As a user, I want to log in securely so that I can access the platform.	Credentials are validated; JWT tokens are returned on success.	High
USU2	User	As a user, I want to receive real-time notifications about key events.	New invitations, start/end of ballots, and successful votes trigger notifications.	Medium
USU3	User	As a user, I want to update my profile information and OTP preferences.	Changes to phone number or OTP settings are saved and reflected in future actions.	Low
USU4	User	As a user, I want to see a list of my past and ongoing ballots.	Dashboard displays current and completed ballots with their status.	Medium
USC1	Creator	As a creator, I want to configure a poll or election so I can collect votes.	Form blocks past dates and empty titles; API returns `ballot_id` on success.	High
USC2	Creator	As a creator, I want to define ballot settings such as anonymity and visibility.	Options for anonymity, visibility, and result publication are enforced and stored.	High
USC3	Creator	As a creator, I want to add participants manually, by @command or QR.	Users are added only once; QR and command methods are validated.	Medium

USC4	Creator	As a creator, I want to publish results manually or set them to publish automatically.	Auto: results visible at `end_time`; Manual: button triggers publication.	Medium
USC5	Creator	As a creator, I want to download the final results as a PDF.	A button becomes available after publication; file is correctly generated.	Medium
USC6	Creator	As a creator, I want to cancel a ballot before it starts.	Ballot is flagged as canceled; participants are notified.	Low
USV1	Voter	As a voter, I want to cast a vote securely using an OTP code.	OTP sent and validated; vote stored only after correct OTP.	High
USV2	Voter	As a voter, I want to submit a blank vote if permitted by the ballot.	Blank option is visible and accepted only if the ballot allows it.	Medium
USV3	Voter	As a voter, I want to view the final results once they are published.	Results page shows vote percentages and totals after publication.	Medium

Table A.3: User stories by actor

Appendix B

Additional Tables and Figures

B.1 Registration Field Behaviors and Validation Rules

Field	**UX Behaviour**	**Validation Rule**
Full name	Autofocus on page load	Required
Email	Mobile keyboard shows "@" shortcut	Required; must match valid email pattern
Password / Confirm password	Masked input; tooltip lists the minimum requirements	Required; must match each other
User group	Single-select drop-down (Student, Lecturer, Administration)	Required
IPB number	Digits only, max length 8	Required
Course	Free text field	Optional
Profile picture	File picker limited to images	Optional

Table B.1: User registration fields

B.2 Ballot Creation Fields and Validation Rules

Field	**UX Behaviour**	**Validation Rule**
Title	Autofocus on first input; auto-capitalizes first letter	Required; non-empty string
Description	Textarea with character counter	Required; max 500 characters
Options/Listas	Dynamically add/remove fields	At least one required; values must not be empty
Candidates (if election)	Selector per list; excludes already selected users	At least one candidate per list with valid `user_id`
Accepted Choices	Numeric selector (1 to number of options)	Required; min 1, max number of options
Allow Blank Votes	Radio buttons (Yes/No)	Required
Anonymous Voting	Radio buttons (Yes/No)	Required
Visibility	Radio buttons (Public/Private)	Required
Result Publication	Radio buttons (Auto/Manual)	Required
Start Time	Datetime input with min constraint	Required; cannot be in the past
End Time	Datetime input with min constraint	Required; must be after start time
File Attachment	File input accepting documents	Optional; single file

Table B.2: Ballot creation fields

B.3 Ballot Configuration Icons

Icon	Meaning	Explanation
▶ / ✋	Publication mode	▶: automatic publication, ✋: manual trigger required
☑ / ✖	Blank votes	☑: blank votes allowed, ✖: blank votes forbidden
🎭	Anonymity	Voter identities are hidden
🙈	Privacy	Only invited users can access the ballot
⚠	Integrity alert	Vote chain verification failed — possible tampering

Table B.3: Icon summary for configuration options

B.4 Ballot Feed Action Icons

Icon	Context	Condition / Action
🖱	Ballot Feed	*Vote Now* — a participant in an ongoing vote that has not yet voted
📋	Ballot Feed	*Results* — shown when results are published
👁	Ballot Feed	*View Only* — default fallback if no other action applies

Table B.4: Action icons in Ballot Feed

B.5 My Ballots Administrative Icons

Icon	Context	Condition / Action
📋	My Ballots	*Publish Results* — voting finished and results are not yet published
▦	My Ballots	*QR Code Modal* — available if joining via QR code is enabled

Table B.5: Administrative icons available in My Ballots view

B.6 Owner Management Controls

Icon	Position	Action
⚙	Top-right	Opens the ballot's settings or results page
🗑	Bottom-right	Opens a confirmation modal and triggers a DELETE API call

Table B.6: Persistent icons for ballot management

B.7 Ballot Status Color Mapping

Colour	Meaning
	Starting – scheduled but not yet open.
	Ongoing – voting is open.
	Finished – voting closed; results may follow.
	Integrity Failed – tampering detected.

Table B.7: Color coding for ballot statuses