Aplicação de um contrato inteligente blockchain para o mercado imobiliário brasileiro

Wallyson Nunes Alves Lima[1]*; Alexander Chaves Lopez[2]

Application of a Blockchain Smart Contract for the Brazilian Real Estate Market

This work investigated the application of blockchain-based smart contracts for the Brazilian real estate market, with the objective of reducing bureaucracy, enhancing legal security, and optimizing property purchase and sale processes. Using the Ethereum platform and the Solidity language, the WebSmartOffice system was developed to simulate the operations of a notary office, from contract signing to execution. The methodology involved building a monolithic application with a Kotlin backend, blockchain integration via Web3j, and a frontend implemented in Angular. To adapt the solution to the national context, the proposed model foresees the use of notary offices as network validators, ensuring the integrity and authenticity of the stored contracts. The results indicate that the adoption of smart contracts can reduce costs, minimize fraud risks, automate transactions, and strengthen the security of property records. It is concluded that the adoption of this technology is feasible and may represent a significant advancement for the Brazilian property registration system.

Abstract ou **Resumen**

This study investigated the application of blockchain-based smart contracts to the Brazilian real estate market, aiming to reduce bureaucracy, increase legal security, and optimize real estate purchase and sale processes. Using the Ethereum platform and the Solidity language, the WebSmartOffice system was developed to simulate notary operations, from contract signing to execution. The methodology involved the construction of a monolithic application with a Kotlin backend, blockchain integration via Web3j, and an Angular frontend. To adapt to the national context, the proposed model envisions notary offices as validators of the network, ensuring the integrity and authenticity of the stored contracts. The results indicate that the application of smart contracts can reduce costs, minimize fraud risks, automate transactions, and strengthen the security of real estate records. It is concluded that adopting this technology is feasible and can represent a significant advancement in the Brazilian property registration system.

Keywords ou **Palabras Clave:** Web 3.0, Cartórios Digitais, DREX, PIX, Ethereum, Dapps.

1.Introduction

One of the safest and most profitable forms of investment is the real estate market, both in the residential and commercial sectors. However, like any type of investment, this market presents advantages and disadvantages, facing challenges such as the need for third parties to verify information, administrative costs, investment security, bureaucracy, and transparency in property ownership. In this context, Industry 4.0 technologies, especially the use of blockchain and smart contracts, emerge as promising alternatives for the sector's modernization. The application of these technologies can simplify property purchase and sale processes, reduce bureaucracy, eliminate intermediaries, and increase transaction security, thereby preventing fraud (Ahmad et al., 2020).

Blockchain can be defined as a shared, secure, and continuously growing ledger system in which all participants maintain a copy of the records, which can only be altered through consensus among the parties involved. Structured as a chain of blocks, each block contains a set of transactions or data and has a unique identifier called a hash, as well as the hash of the previous block, thereby ensuring the sequential integrity of the information (Bashir, 2020). The introduction of this technology into the real estate market aims precisely to mitigate fraud risks by increasing the reliability and transparency of property records.

Among blockchain-based platforms, Ethereum stands out for its decentralized, open-source nature and for supporting the creation of smart contracts and decentralized applications (dApps). In addition to functioning as a single state machine and a virtual machine, Ethereum synchronizes and stores system state changes, using the cryptocurrency Ether (ETH) to measure and limit the operational costs of transactions and code execution. In this study, the Ethereum platform is employed for the implementation and execution of smart contracts that validate real estate transactions (Antonopoulos, 2018).

Smart contracts, in turn, are defined as sets of promises expressed in digital form, with protocols that ensure the automatic fulfillment of obligations agreed upon between parties. In practical terms, they are self-executing programs that run on the blockchain network, automatically triggering their clauses when predefined conditions are met, without the need for intermediaries such as lawyers or brokers. The immutability, transparency, and security of these contracts are guaranteed by the intrinsic properties of blockchain, which prevent modifications after registration (Antonopoulos, 2018).

The implementation of smart contracts is carried out using high-level programming languages, the most widely used being Solidity, specifically designed for the development of smart contracts that run on the Ethereum Virtual Machine (EVM). This language enables the definition of rules, business logic, data storage, and financial interactions securely and immutably on the blockchain.

Using Solidity, developers can create self-executing contracts that enable decentralized transactions without the need for intermediaries. The contract source code is compiled, generating bytecode that is interpreted and executed by the EVM, the runtime environment responsible for processing contracts on the Ethereum network.

After compilation, it is necessary to create a specific transaction to deploy the contract on the blockchain, which results in the generation of a unique address used in future operations, such as transferring funds or invoking specific functions. A central concept in this process is GAS, the unit of measurement that represents the computational effort required to execute operations on the network. Users must pay, in Ether (ETH), the amount corresponding to gas consumption, thus remunerating validators and ensuring the continuity and security of the network (Antonopoulos, 2018).

Thus, this work aims to develop a system based on smart contracts to validate property purchase and sale transactions, with the purpose of optimizing the processes involved, reducing bureaucracy, increasing efficiency, and strengthening legal security within the context of Brazilian notary offices.

Methodology or Materials and Methods

For the system called WebSmartOffice, designed to process property purchase and sale transactions using smart contracts, a monolithic architecture was adopted. The backend layer of the application was implemented in Kotlin, a modern, statically typed programming language compatible with the Java ecosystem and running on the Java Virtual Machine (JVM). The choice of Kotlin was based on its interoperability with Java, as well as its robustness, security, and the JVM’s widespread adoption in large-scale systems — all essential characteristics to ensure the system's scalability and reliability. Additionally, the extensive library ecosystem and multiplatform support were determining factors for this decision.

The backend was structured using Spring Boot, a framework that facilitates the initial project configuration and the integration of required libraries. The project organization follows the principles of Clean Architecture and Clean Code, ensuring the correct definition and naming of classes, objects, methods, and variables, as well as promoting maintainable and evolvable code.

For the persistence layer implementation, Spring Data was used, integrating the application with a relational PostgreSQL database. The choice of an SQL database is justified by its robustness, security, open-source nature, and wide adoption in commercial systems and by the free software community. Considering that the system's data structure would be rigid, a relational database was chosen to minimize frequent changes to the model during development.

User authentication and authorization were implemented using Spring Security, a widely adopted, secure, and easy-to-use technology that ensures access control to system resources.

Regarding smart contract management, Hardhat was used as the development, compilation, and deployment environment, along with the Hardhat Network to simulate a local Ethereum blockchain. The language chosen for implementing the contracts was Solidity, the market standard for development on the Ethereum network. Due to the need for real funds to use Ethereum's public test networks, a private local network was created using Hardhat Network with simulated Ethereum accounts.

Integration between smart contracts and the backend application was achieved using the Web3j library, which enables direct communication with Ethereum blockchains from Java/Kotlin applications. This approach allowed interaction with smart contracts in a corporate environment, preserving the solution's robustness and scalability.

For the frontend layer development, the Angular framework was used, widely adopted in the market and supported by extensive documentation, learning resources, and a robust, secure architecture. Additionally, HTML, CSS, Bootstrap, JavaScript, and TypeScript were employed to build the user interface and user experience.

The development flow followed these stages: initially, the backend was built with all its endpoints, using Swagger for service documentation and testing. Next, backend authentication and authorization were implemented, exposing the endpoints through HTTP calls following the REST API standard. Subsequently, the frontend was developed with Angular, including screen creation, routing, and access control through authentication.

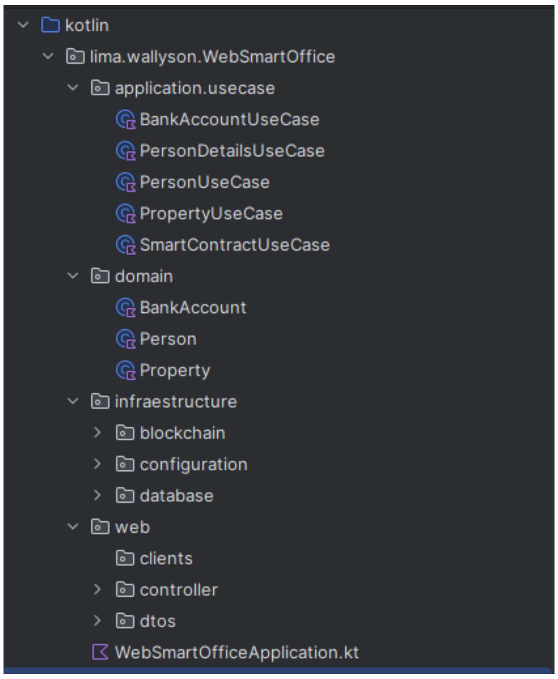
Finally, the smart contracts were implemented in Solidity, compiled using Hardhat, and integrated with the backend by generating Java classes using the Web3j library, consolidating communication between the system's different layers.

Results and Discussion

For the development of the WebSmartOffice system, good software engineering practices were adopted, such as Clean Code (MARTIN, 2019), Clean Architecture (MARTIN, 2018), and Domain-Driven Design (DDD) (EVANS, 2012). The application of clean code principles involved the use of self-explanatory names for variables and methods, the elimination of code duplication, the creation of cohesive functions, each with a single responsibility, as well as the inclusion of clear and relevant comments, consistent formatting, and structured error handling, in line with Clean Code guidelines (Martin, 2019).

The adoption of Clean Architecture ensured independence from specific frameworks and technologies, allowing, for example, the WebSmartOffice system not to be tied to a single database system, enabling its replacement if necessary. Furthermore, the interfaces between the application layers were clearly defined, promoting separation of concerns, code readability, and maintainability.

Another fundamental aspect was the isolation of the domain, in accordance with DDD, which establishes the domain and its business rules as the core of the application, independent of the technology used. Thus, dependencies are always directed toward the center of the architecture, reinforcing the priority of business rules over implementation details.



```
kotlin
  lima.wallyson.WebSmartOffice
    application.usecase
      BankAccountUseCase
      PersonDetailsUseCase
      PersonUseCase
      PropertyUseCase
      SmartContractUseCase
    domain
      BankAccount
      Person
      Property
    infraestructure
      blockchain
      configuration
      database
    web
      clients
      controller
      dtos
    WebSmartOfficeApplication.kt
```

Figure 1 – Directory structure following Clean Architecture – author (2025)

The adoption of Clean Architecture offers, as its main advantage, the clear separation of responsibilities among the system’s layers. The business logic does not depend on any specific framework, as it is isolated in its own dedicated layer. This approach allows the application to be structured into well-defined directories, which significantly contributes to code organization, increased maintainability, scalability, testability, and readability.

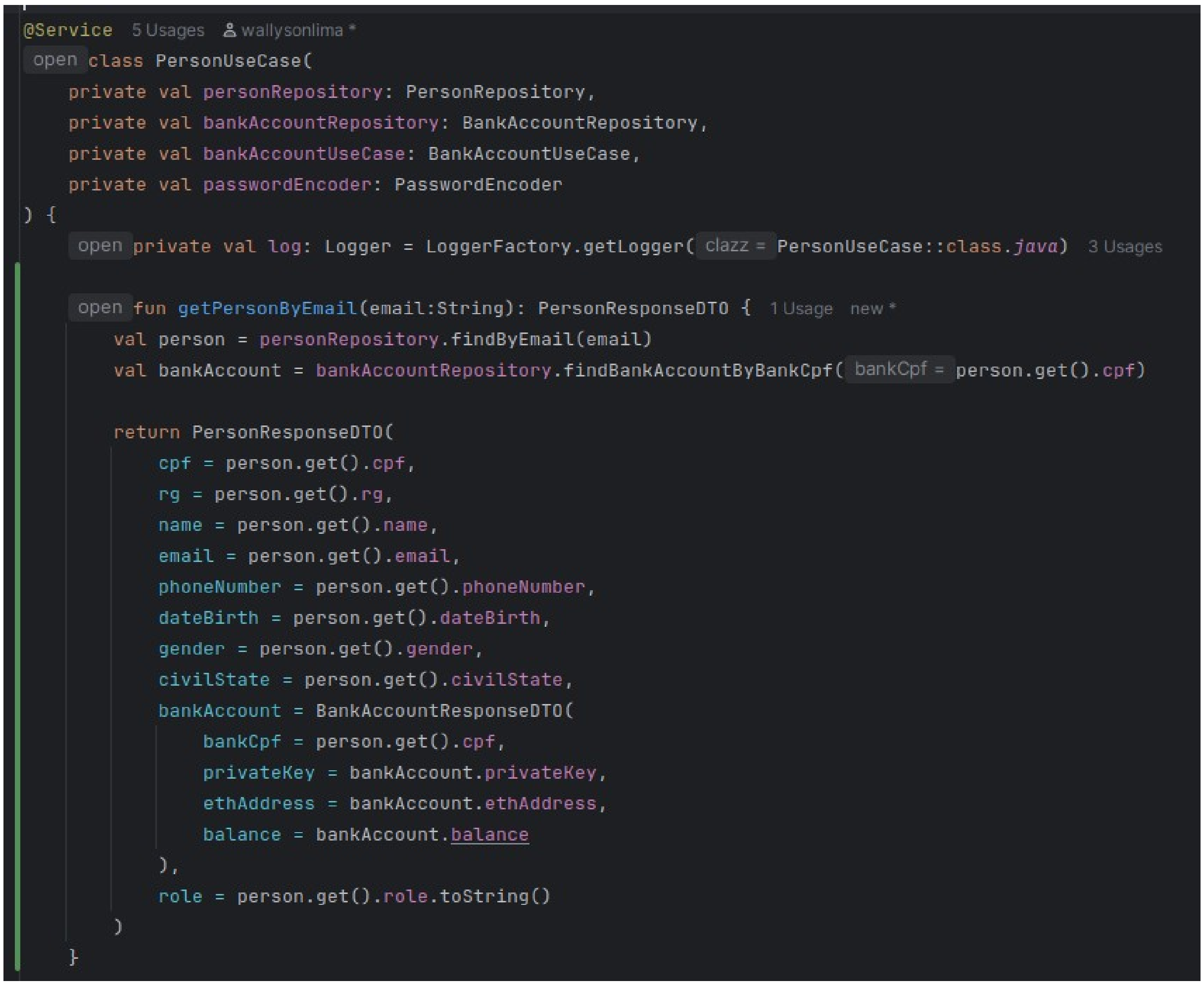
By applying Domain-Driven Design (DDD) within the API, the system becomes business-oriented, ensuring that it accurately reflects the rules, behaviors, and constraints of the real-world problem domain.

When analyzing the code presented in Figure 2, it is evident that good software development principles are applied, aligned with the SOLID guidelines. The `PersonUseCase` component is properly segregated in the application layer, acting as an orchestrator between services, repositories, and the transformation of DTOs (Data Transfer Objects). Furthermore, the code design follows a domain-oriented approach, ensuring that the business logic remains clear and isolated from infrastructure details.

Regarding Clean Code practices, the choice of expressive and self-explanatory names stands out: the `getPersonByEmail` method clearly communicates its purpose, contributing to the readability and maintainability of the code. The implementation also demonstrates the application of fundamental SOLID principles.

The Single Responsibility Principle (SRP) is observed, as the method exclusively performs the operation of retrieving a person by email and structuring the response in a DTO, while delegating data persistence to the repository and the logic related to the bank account to a specialized use case.

Additionally, the Dependency Inversion Principle (DIP) is evident, since the `PersonUseCase` depends on abstractions such as `PersonRepository`, `BankAccountRepository`, and `PasswordEncoder` rather than concrete implementations. This approach promotes greater flexibility, testability, and facilitates the replacement of components, reinforcing the architectural quality of the project.



```
@Service  5 Usages  wallysonlima *
open class PersonUseCase(
    private val personRepository: PersonRepository,
    private val bankAccountRepository: BankAccountRepository,
    private val bankAccountUseCase: BankAccountUseCase,
    private val passwordEncoder: PasswordEncoder
) {
    open private val log: Logger = LoggerFactory.getLogger( clazz = PersonUseCase::class.java)  3 Usages

    open fun getPersonByEmail(email:String): PersonResponseDTO {  1 Usage  new *
        val person = personRepository.findByEmail(email)
        val bankAccount = bankAccountRepository.findBankAccountByBankCpf( bankCpf = person.get().cpf)

        return PersonResponseDTO(
            cpf = person.get().cpf,
            rg = person.get().rg,
            name = person.get().name,
            email = person.get().email,
            phoneNumber = person.get().phoneNumber,
            dateBirth = person.get().dateBirth,
            gender = person.get().gender,
            civilState = person.get().civilState,
            bankAccount = BankAccountResponseDTO(
                bankCpf = person.get().cpf,
                privateKey = bankAccount.privateKey,
                ethAddress = bankAccount.ethAddress,
                balance = bankAccount.balance
            ),
            role = person.get().role.toString()
        )
    }
```

Figure 2 – Good Software Engineering Practices Applied to the Code – author (2025)

If the API expands its functionalities or assumes new responsibilities, the adopted architecture supports its evolution and maintenance, in line with the SOLID principles, which aim to make the software flexible, extensible, and easy to modify.

In the implementation analyzed, the `application` directory contains the use cases, which are responsible for orchestrating the business logic, following the Single Responsibility Principle (SRP) by isolating each workflow into specific components.

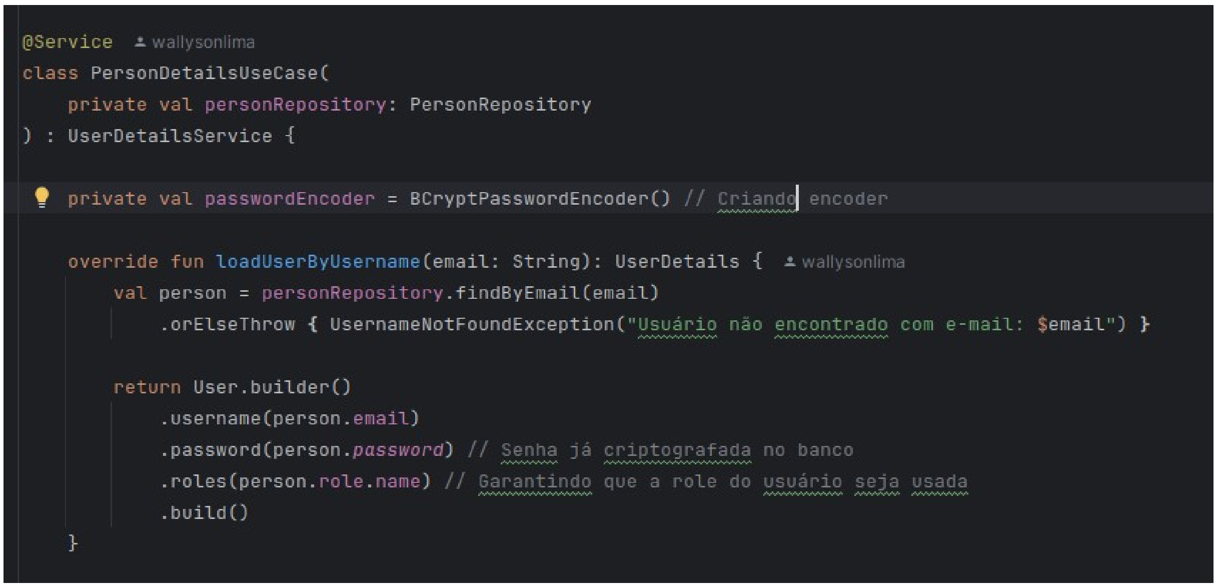
The `domain` directory concentrates the core domain elements, such as entities and value objects, aligning with the Dependency Inversion Principle (DIP) by defining abstractions that are independent of infrastructure details.

In turn, the `infrastructure` directory brings together the infrastructure components, including blockchain integration configurations, security files implemented with Spring Security, and database connections. This clear separation between domain, application, and infrastructure ensures that changes in external technologies or persistence mechanisms do not directly affect the business rules, reinforcing the Open/Closed Principle (OCP), which states that modules should be open for extension but closed for modification.

This modular structure, oriented by solid software engineering principles, sustains the scalability, maintainability, and testability of the application as new requirements arise.

Finally, the `web` directory contains the controllers, responsible for exposing the endpoints to the frontend, in addition to the DTOs (Data Transfer Objects) used for data transfer between layers. This modular organization follows the principles of Clean Architecture and provides a solid foundation for developing robust and scalable systems.

To handle authentication and authorization in the WebSmartOffice application, Spring Security was used. In Figure 3, it is possible to see where the authentication logic is created.

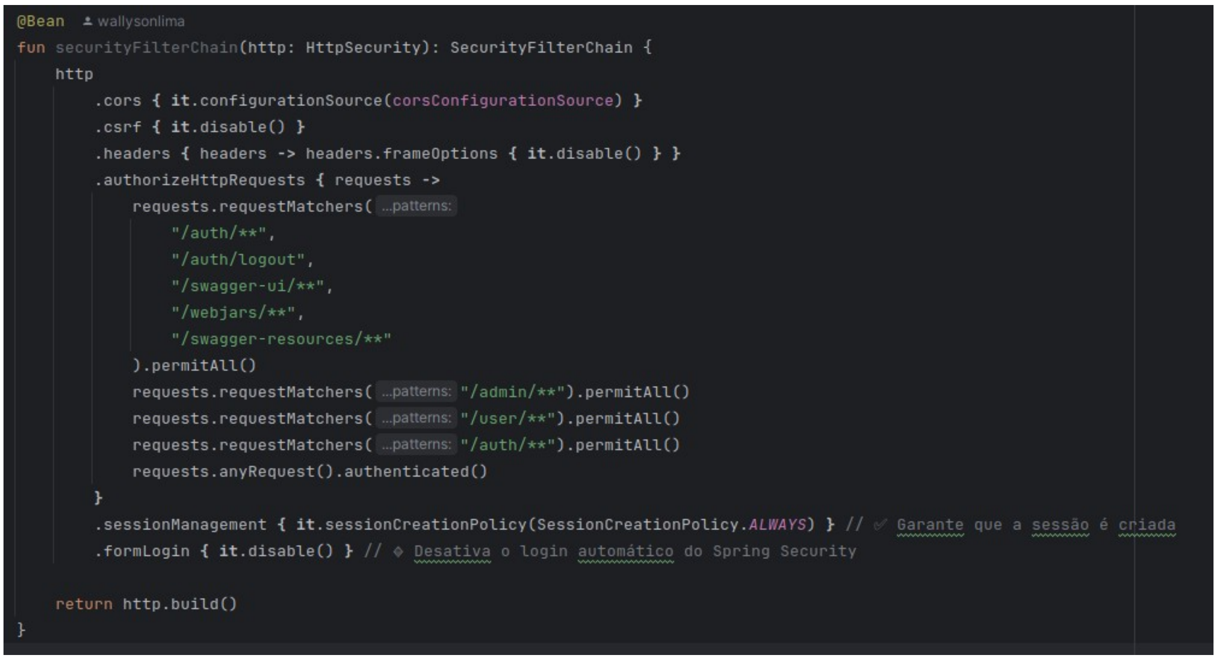


```kotlin
@Service  wallysonlima
class PersonDetailsUseCase(
    private val personRepository: PersonRepository
) : UserDetailsService {

    private val passwordEncoder = BCryptPasswordEncoder() // Criando encoder

    override fun loadUserByUsername(email: String): UserDetails {  wallysonlima
        val person = personRepository.findByEmail(email)
            .orElseThrow { UsernameNotFoundException("Usuário não encontrado com e-mail: $email") }

        return User.builder()
            .username(person.email)
            .password(person.password) // Senha já criptografada no banco
            .roles(person.role.name) // Garantindo que a role do usuário seja usada
            .build()
    }
```

Figure 3 – Method that Performs Authentication in Spring Security – author (2025)

In the `loadUserByUsername` method, the system searches the database for a valid user with the specified email address. Next, the system verifies the stored password (which is already encrypted) and defines the user's role (access profile). In the context of the WebSmartOffice system, there are only two available roles: Admin and User. After validation, the method returns an instance representing the authenticated user.

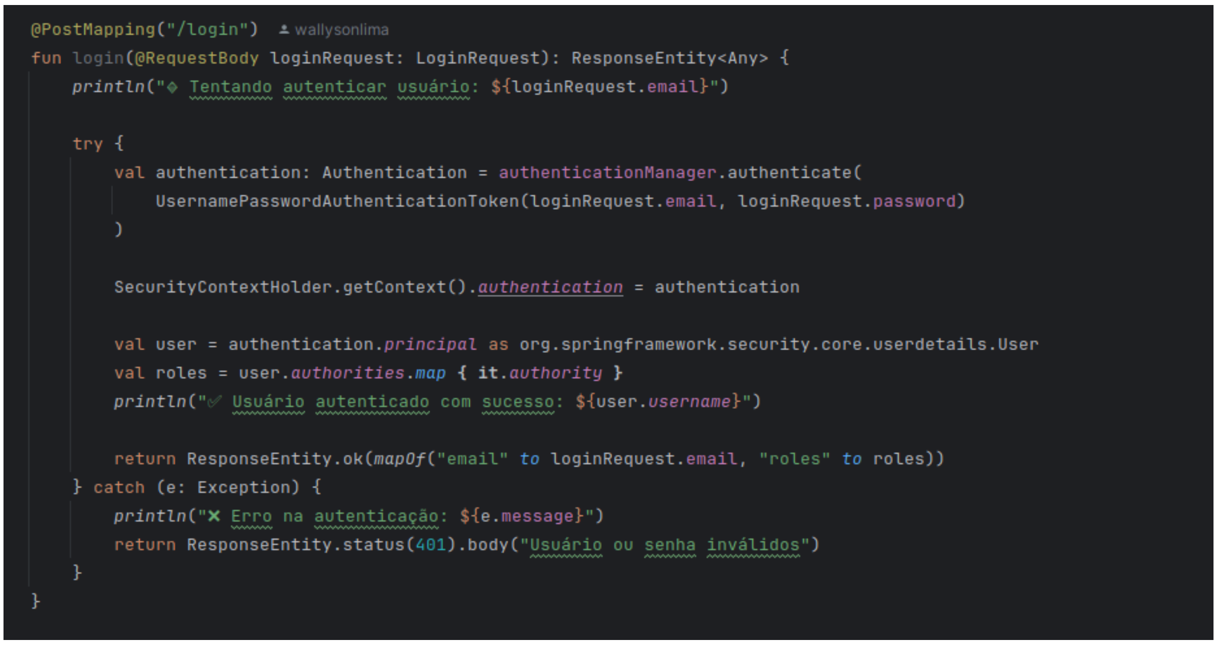
In Figure 4, the authorization configuration of Spring Security is presented, which is responsible for defining the access permissions for the application's routes. At this stage, it is specified whether a particular route requires prior authentication and whether a user session should be created so that the frontend can use it during navigation.



```
@Bean  wallysonlima
fun securityFilterChain(http: HttpSecurity): SecurityFilterChain {
    http
        .cors { it.configurationSource(corsConfigurationSource) }
        .csrf { it.disable() }
        .headers { headers -> headers.frameOptions { it.disable() } }
        .authorizeHttpRequests { requests ->
            requests.requestMatchers( ...patterns:
                "/auth/**",
                "/auth/logout",
                "/swagger-ui/**",
                "/webjars/**",
                "/swagger-resources/**"
            ).permitAll()
            requests.requestMatchers( ...patterns: "/admin/**").permitAll()
            requests.requestMatchers( ...patterns: "/user/**").permitAll()
            requests.requestMatchers( ...patterns: "/auth/**").permitAll()
            requests.anyRequest().authenticated()
        }
        .sessionManagement { it.sessionCreationPolicy(SessionCreationPolicy.ALWAYS) } // ✅ Garante que a sessão é criada
        .formLogin { it.disable() } // ◆ Desativa o login automático do Spring Security

    return http.build()
}
```

Figure 4 – Route Authorization Using Spring Security – author (2025)

As shown in Figure 5, certain requests have been explicitly authorized, such as the login, logout, and API documentation routes via Swagger, which do not require authentication. For simplification purposes in the development environment, routes with the /admin and /user prefixes have also been temporarily allowed. All other requests require the user to be authenticated.



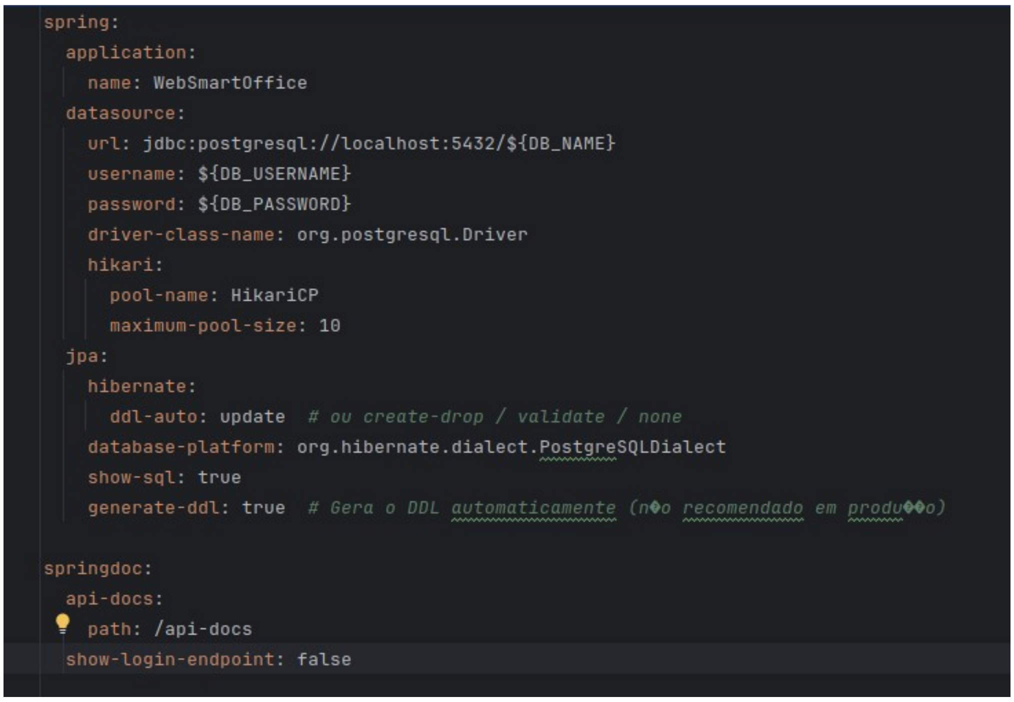
```
@PostMapping("/login")  wallysonlima
fun login(@RequestBody loginRequest: LoginRequest): ResponseEntity<Any> {
    println("◆ Tentando autenticar usuário: ${loginRequest.email}")

    try {
        val authentication: Authentication = authenticationManager.authenticate(
            UsernamePasswordAuthenticationToken(loginRequest.email, loginRequest.password)
        )

        SecurityContextHolder.getContext().authentication = authentication

        val user = authentication.principal as org.springframework.security.core.userdetails.User
        val roles = user.authorities.map { it.authority }
        println("✅ Usuário autenticado com sucesso: ${user.username}")

        return ResponseEntity.ok(mapOf("email" to loginRequest.email, "roles" to roles))
    } catch (e: Exception) {
        println("❌ Erro na autenticação: ${e.message}")
        return ResponseEntity.status(401).body("Usuário ou senha inválidos")
    }
}
```

Figure 5 – User Authentication Logic – author (2025)

In Figure 5, the method responsible for exposing the authentication logic to the frontend is presented, through the “/auth/login” route. This endpoint validates the user’s credentials, generates the corresponding session or token, and initiates the security context required for authenticated navigation.

The use of Spring Security in this context brings several advantages for backend applications. It provides a robust, flexible, and extensible authentication and authorization mechanism, enabling fine-grained access control to API routes based on roles, permissions, and custom security rules. Additionally, Spring Security has native integration with modern security standards such as JWT, OAuth2, HTTP sessions, and password encryption using secure algorithms like Bcrypt.

To create the database connection, JPA and Hibernate were used, configuring the connection in an application.yml file. The database username and password are stored in a system variable to avoid exposing vulnerabilities in the API. In the URL field, the database connection string is defined, and the driver-class-name indicates the specific driver used to connect to the database. This is an example of the Dependency Injection concept in Spring Data, where developers do not need to manage the database connection manually — it is injected automatically by Spring. In this way, it is possible to quickly switch the database by simply updating the connection string and the driver, as shown in Figure 6.

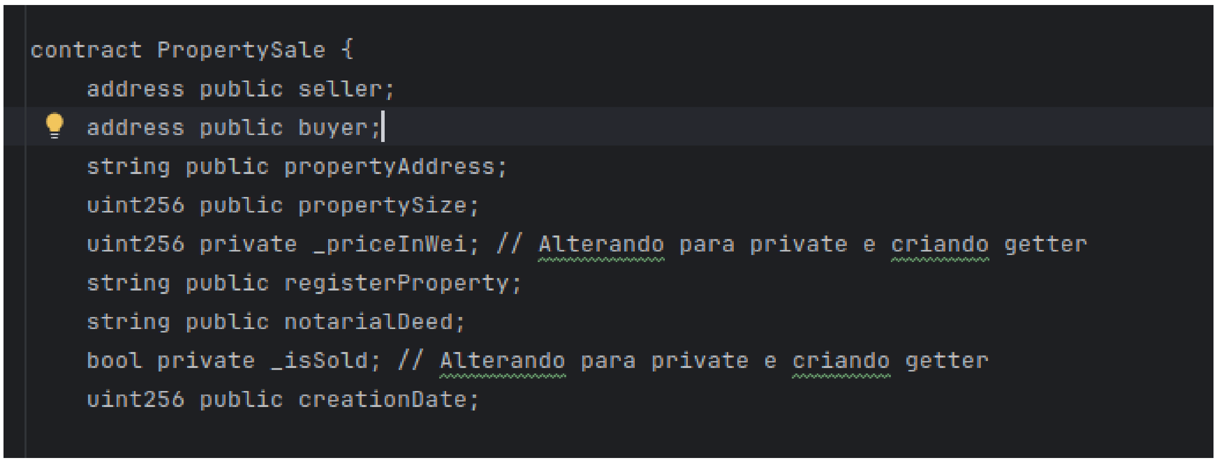


```yaml
spring:
  application:
    name: WebSmartOffice
  datasource:
    url: jdbc:postgresql://localhost:5432/${DB_NAME}
    username: ${DB_USERNAME}
    password: ${DB_PASSWORD}
    driver-class-name: org.postgresql.Driver
    hikari:
      pool-name: HikariCP
      maximum-pool-size: 10
  jpa:
    hibernate:
      ddl-auto: update  # ou create-drop / validate / none
    database-platform: org.hibernate.dialect.PostgreSQLDialect
    show-sql: true
    generate-ddl: true  # Gera o DDL automaticamente (n�o recomendado em produ��o)

springdoc:
  api-docs:
    path: /api-docs
  show-login-endpoint: false
```

Figure 6 – application.yml File – author (2025)

The smart contract developed in this work was implemented using Solidity, the main programming language used for creating smart contracts on the Ethereum network. The contract represents a generic model for property purchase and sale, which can be reused in various transactions of this type by simply changing the specific data related to the buyer, seller, and the property involved.

As illustrated in Figure 7, the contract requires the definition of essential information, such as the Ethereum addresses of the buyer and seller, the property value (in Ether), the property size, the property address, the property registration, and the notarial deed.



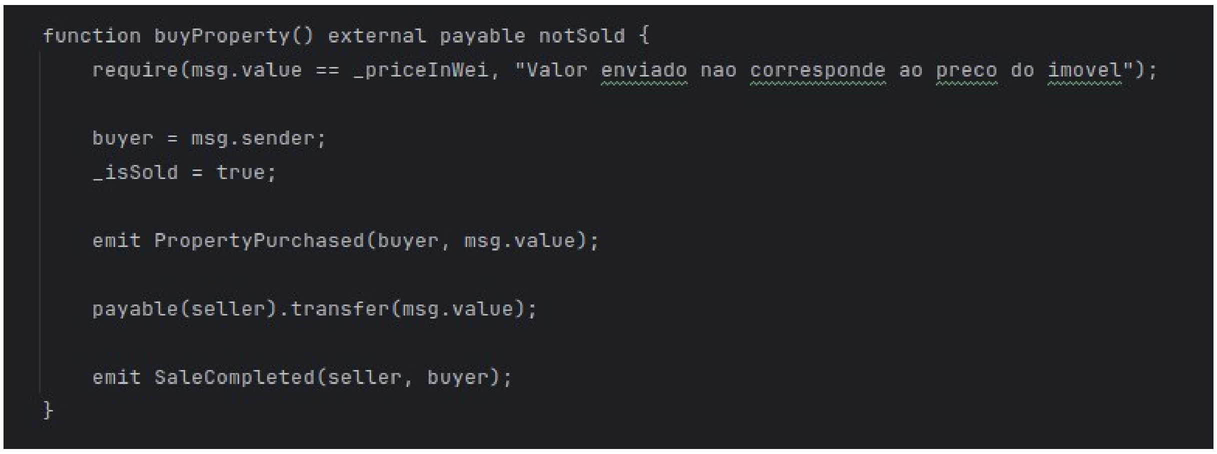
```
contract PropertySale {
    address public seller;
    address public buyer;
    string public propertyAddress;
    uint256 public propertySize;
    uint256 private _priceInWei; // Alterando para private e criando getter
    string public registerProperty;
    string public notarialDeed;
    bool private _isSold; // Alterando para private e criando getter
    uint256 public creationDate;
```

Figure 7 – Information Used in the Smart Contract – author (2025)

The process of using the smart contract involves three main stages:

1. Compilation: The Solidity code is compiled using tools such as Solc (Solidity Compiler), which generate the ABI and bytecode files required for deployment.
2. Deployment on the network: The contract is then sent and registered on an Ethereum blockchain — in this case, the HardHat network is used.
3. Interaction with the contract: Once deployed, any party can interact with the contract, provided they comply with the rules and conditions defined in the code.

The core logic of the transaction is encapsulated in the `buyProperty()` method, which executes the property sale. The transaction amount is represented in Wei, the smallest unit of the Ether cryptocurrency. The method validates whether the amount sent matches the stipulated property value, registers the buyer through `PropertyPurchased`, transfers the amount to the seller using the `payable` keyword, and finalizes the operation by emitting the `SaleCompleted` event, which confirms the completion of the sale between the parties, as shown in Figure 8.



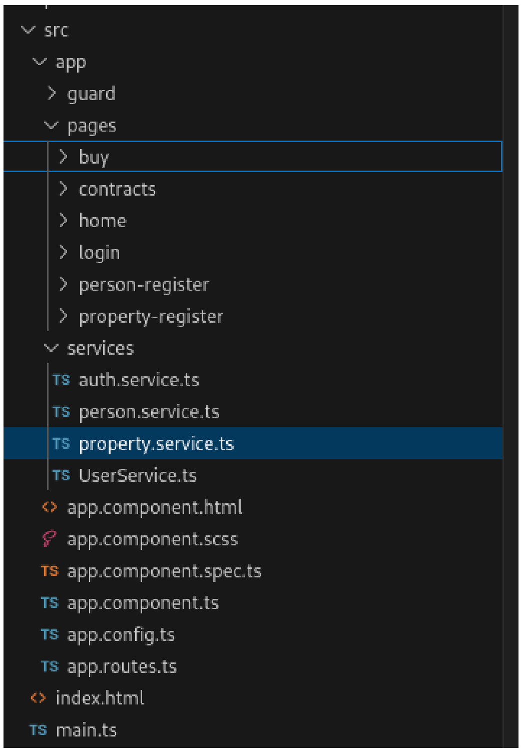
```
function buyProperty() external payable notSold {
    require(msg.value == _priceInWei, "Valor enviado nao corresponde ao preco do imovel");

    buyer = msg.sender;
    _isSold = true;

    emit PropertyPurchased(buyer, msg.value);

    payable(seller).transfer(msg.value);

    emit SaleCompleted(seller, buyer);
}
```

Figure 8 – `buyProperty` Method – author (2025)

The choice to use the Solidity language and the Ethereum network is justified by several factors. Ethereum is the leading platform for executing smart contracts, offering security, decentralization, immutability, and wide adoption in the market. Solidity, in turn, is a contract-oriented language with syntax inspired by traditional languages such as JavaScript and C++, and it benefits from strong community support, extensive documentation, and direct compatibility with the Ethereum ecosystem, which facilitates the development, testing, and auditing of reliable smart contracts.

To enable the connection with the frontend, REST API calls over the HTTP protocol were used. The system's interface was developed with Angular 17, using HTML, CSS, JavaScript, and TypeScript, which are widely established technologies for modern web application development. Angular was chosen for being a widely used, robust, and scalable framework that offers native support for reusable components, dependency injection, routing, reactive forms, and easy integration with APIs.

The frontend structure was organized following the convention proposed by Angular itself, adopting the use of reusable components to promote modularity, ease of maintenance, and code reusability. As illustrated in Figure 9, the project was divided into specific directories with well-defined responsibilities.



```
src
  app
    guard
    pages
      buy
      contracts
      home
      login
      person-register
      property-register
    services
      auth.service.ts
      person.service.ts
      property.service.ts
      UserService.ts
    app.component.html
    app.component.scss
    app.component.spec.ts
    app.component.ts
    app.config.ts
    app.routes.ts
  index.html
  main.ts
```

Figure 9 – Directory Structure Organization Following Angular’s Convention – author (2025)

The pages directory contains the components responsible for each functionality of the application, where each page is composed of a .html file (visual structure), a .css file (styling), and a .ts file (responsible for the logic and behavior of the interface, implemented in TypeScript). Additionally, there is a services directory, which centralizes the frontend’s business logic and handles communication with the backend API through HTTP requests. This separation allows the data consumption logic to remain decoupled from the presentation layer, promoting cohesion and reusability.

At the root of the project is the app directory, which represents the core of the application. It contains the main component (AppComponent), the routing system that manages navigation between pages, and the global configuration files of the application.

This modular structure, based on the component-driven architecture of Angular (ANGULAR, 2025) and aligned with the principles of Clean Architecture (MARTIN, 2017), contributes significantly to the scalability, maintainability, and clarity of the frontend project. Segmenting the application into reusable and independent components favors the separation of concerns, making the code more organized, predictable, and easier to evolve in scenarios of functional expansion or ongoing maintenance.

Smart Contract technology allows agreements to be executed directly between contracting parties, eliminating the need for intermediaries such as banks and notary offices. As demonstrated in the work of Ahmad, Alqarni, Almazroi, and Alam (2020), operations such as property purchases, sales, or rentals can be conducted directly between the parties involved. However, considering the Brazilian reality, the complete absence of intermediaries would be an utopian proposal, given the legal obligation for notary offices to operate. Therefore, the proposed architecture was designed taking into account the Brazilian notary system.

In the suggested model, the execution of Smart Contracts would be carried out by the notary offices. The Ethereum network would function as a distributed network, where, for each new contract executed, a node would be added to the network. In this way, each notary office would have its own copy of this distributed network, as illustrated in Figure 10.

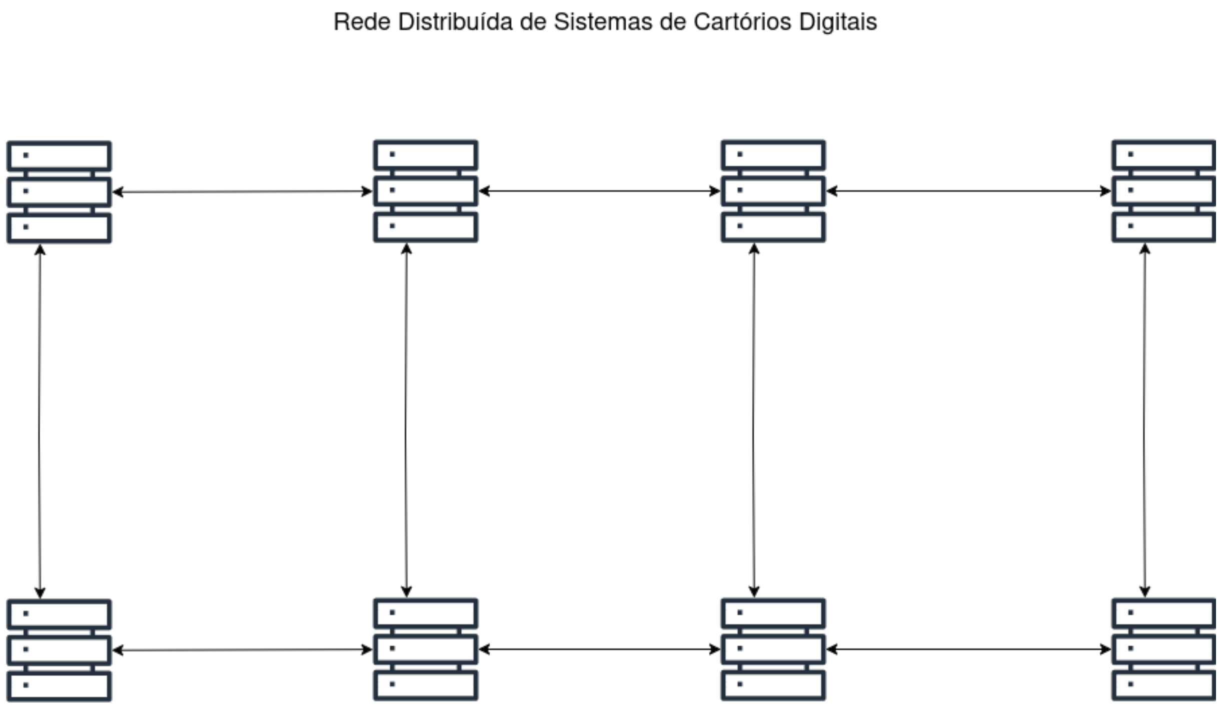


Figura 10 – Distributed system notorial deed – Author (2025)

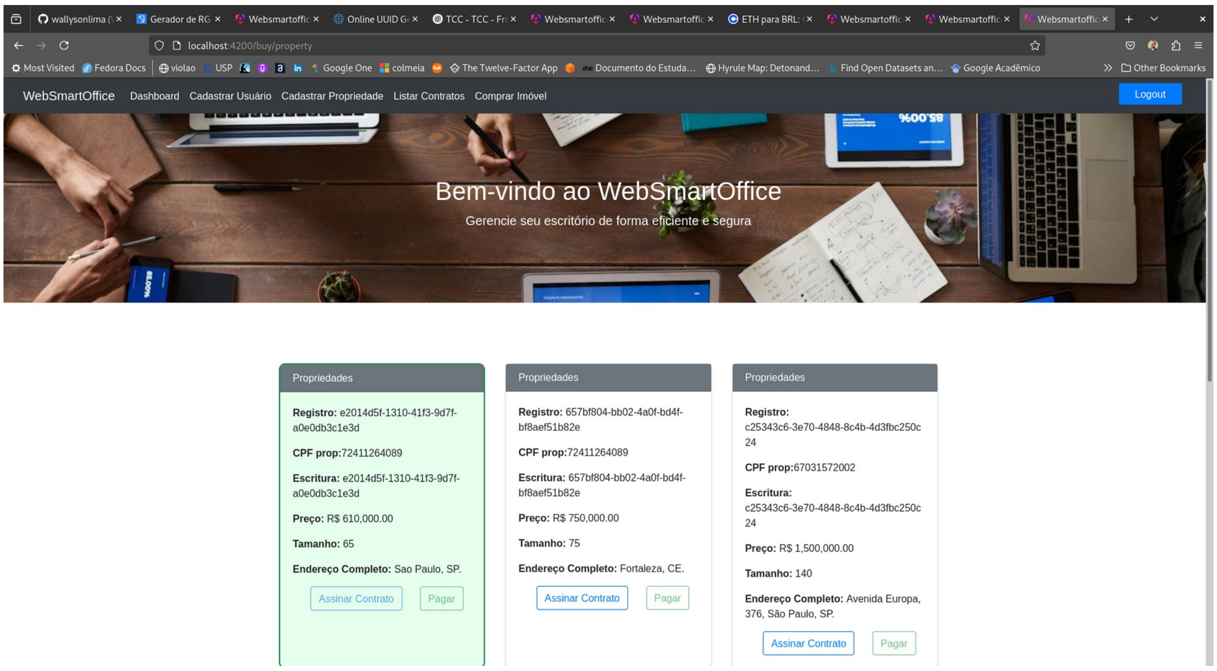
For each type of operation carried out at a notary office, it would be necessary to develop a specific contract model. For example, property purchase and sale contracts, rental agreements, or vehicle purchase contracts would use standardized templates, changing only the specific information of the parties involved.

In the WebSmartOffice system — an application designed to simulate the operation of a notary office — the user authenticates, views the available properties, selects one of them, and signs the contract. Next, the contract is deployed with the provided information and inserted into the Ethereum network. When the user selects the "pay" option, the contract is effectively executed: the buyer’s balance is verified, the transfer of funds between Ethereum accounts takes place, and then the transfer of ownership of the asset is completed. The contract is then finalized, and a block containing the transaction record is added to the network.

Due to time and scope limitations, in WebSmartOffice all these stages (property selection, fund transfer, and transfer of ownership) are carried out within the system itself. However, in a realistic implementation, there would be a specialized system for property purchases and sales; the notary office would be responsible for creating the smart contract, and a financial institution would handle the transfer of funds.

Adapting the model to the Brazilian context, it would be possible to design smart contracts with a defined validity period, with the transfer of funds executed automatically upon the expiration of this period. The contract would store the user’s banking details, and the operation could be performed in an official digital currency, such as DREX (Digital Real), using payment systems like PIX. This approach would offer several advantages: reduction of notary costs, greater legal security (by eliminating the need for manual transfers, which are susceptible to fraud), paper savings, reduced bureaucracy, and increased resilience of records, since all notary offices would maintain a complete copy of the network. In the event of data loss at one notary office, it would be sufficient to restore the records from any other notary office in the network. Moreover, the forgery of contracts would be significantly hindered, as it would require compromising the entire notary Ethereum network to alter a valid block.

Figures 11 and 12 show, in WebSmartOffice, the contract signing process and the transaction hash stored after its execution.


localhost:4200/buy/property
WebSmartOffice
Dashboard
Cadastrar Usuário
Cadastrar Propriedade
Listar Contratos
Comprar Imóvel
Logout
Bem-vindo ao WebSmartOffice
Gerencie seu escritório de forma eficiente e segura

Propriedades

Registro: e2014d5f-1310-41f3-9d7f-a0e0db3c1e3d

CPF prop:72411264089

Escritura: e2014d5f-1310-41f3-9d7f-a0e0db3c1e3d

Preço: R$ 610,000.00

Tamanho: 65

Endereço Completo: Sao Paulo, SP.

Assinar Contrato Pagar

Propriedades

Registro: 657bf804-bb02-4a0f-bd4f-bf8aef51b82e

CPF prop:72411264089

Escritura: 657bf804-bb02-4a0f-bd4f-bf8aef51b82e

Preço: R$ 750,000.00

Tamanho: 75

Endereço Completo: Fortaleza, CE.

Assinar Contrato Pagar

Propriedades

Registro: c25343c6-3e70-4848-8c4b-4d3fbc250c24

CPF prop:67031572002

Escritura: c25343c6-3e70-4848-8c4b-4d3fbc250c24

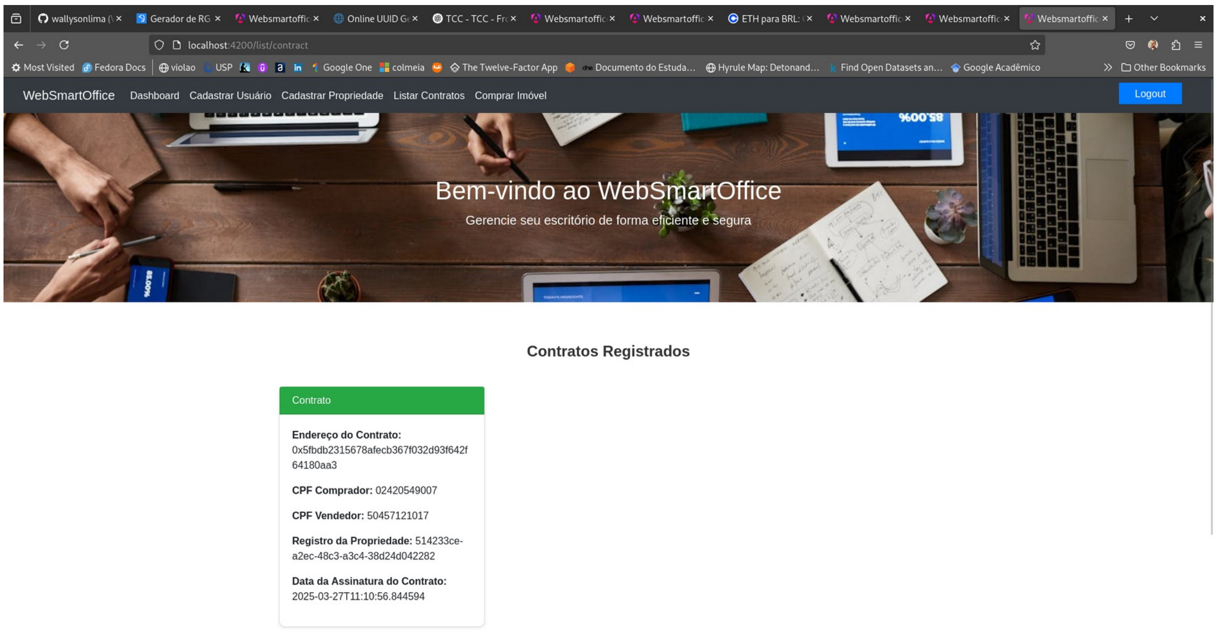
Preço: R$ 1,500,000.00

Tamanho: 140

Endereço Completo: Avenida Europa, 376, São Paulo, SP.

Assinar Contrato Pagar

Figure 11 – Signing and Payment of a Property Purchase Contract – author (2025)


localhost:4200/list/contract
WebSmartOffice
Dashboard
Cadastrar Usuário
Cadastrar Propriedade
Listar Contratos
Comprar Imóvel
Logout
Bem-vindo ao WebSmartOffice
Gerencie seu escritório de forma eficiente e segura

Contratos Registrados

Contrato

Endereço do Contrato: 0x5fbdb2315678afecb367f032d93f642f64180aa3

CPF Comprador: 02420549007

CPF Vendedor: 50457121017

Registro da Propriedade: 514233ce-a2ec-48c3-a3c4-38d24d042282

Data da Assinatura do Contrato: 2025-03-27T11:10:56.844594

Figure 12 – Example of a Saved Contract Hash – author (2025)

4. Conclusion(s) or Final Considerations

This research made it possible to identify the technologies required to implement property purchase and sale transactions using smart contracts. An architecture adapted to the Brazilian context was proposed, highlighting the advantages that this approach would bring to the notary system: increased legal security, cost reduction, reduced bureaucracy, and automation of property purchase and sale processes.

Various technologies studied throughout the MBA in Software Engineering were also used and integrated, such as API development, frontend, authentication/authorization, blockchain, Ethereum, Solidity, among others.

As a suggestion for future work, it is proposed to implement other types of contracts — such as rental agreements for real estate, vehicle purchase contracts, land acquisition, or business partnerships. To achieve this, it would only be necessary to define a specific contract model for each type of transaction. There is also the possibility of developing an architecture based on microservices, with dedicated modules for property purchases, banking operations, and notarial management.

5. Acknowledgment

I would like to thank my advisor, Alexander Chaves, for his guidance and for selecting the research topic. I also thank my family, without whom this work would not have been possible, especially my mother, who has always encouraged my studies, and my wife, who understood the necessary sacrifices during the development of this project. Finally, I express my gratitude to the public education system, which has been responsible for my entire academic education.

6. Referencies

GOOREN, Minko; GOOGLE INC. 2023. *Angular: Up & Running*. 2. ed. Sebastopol, CA, EUA: O’Reilly Media Inc.

MARTIN, Robert C. 2018. Arquitetura limpa: o guia do artesão para estrutura e design de software. 1. ed. São Paulo, SP, BR: Alta Books..

MARTIN, Robert C. 2019. Código limpo: habilidades práticas do Agile software. 1. ed. São Paulo, SP, BR: Alta Books.

Barghuthi, N.; Karamitsos, I.; Papadaki, M. 2018. Design of the Blockchain Smart Contract: A Use Case for Real State. British University of Dubai, Dubai, UAE.

EVANS, Eric. 2012. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. 1. ed. Boston, MA, EUA: Addison-Wesley Professional.

Bashir, Imran. 2020. Mastering Blockchain. 3ed. Packt Publishing Ltd. Birmingham, BHX, UK.

Antonopoulos, Andreas. 2018. Mastering Ethereum. 1ed. O’Reilly Media Inc. California, CA, EUA.

Ahmad, I.; Alam, L. ;Almazroi, A. A.; Alqarni, M. A. 2020. Real State Management via a Decentralized Blockchain Platform.

SOLIDITY. 2025. *Solidity Documentation*. Disponível em: https://docs.soliditylang.org. Acesso em: 05 jul. 2025.

Julie, E. G; Margret, M. K. 2023. Smarter and resilient smart contracts applications for smart cities environment usin blockchain technology.