Aplicação de um contrato inteligente blockchain para o mercado imobiliário brasileiro

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

Aplicação de um contrato inteligente blockchain para o mercado imobiliário brasileiro

Resumo (ou Sumário Executivo)

Este trabalho investigou a aplicação de contratos inteligentes (*smart contracts*) baseados em blockchain para o mercado imobiliário brasileiro, com o objetivo de reduzir a burocracia, aumentar a segurança jurídica e otimizar os processos de compra e venda de imóveis. Utilizando a plataforma Ethereum e a linguagem Solidity, foi desenvolvido o sistema WebSmartOffice, que simula as operações de um cartório, desde a assinatura até a execução dos contratos. A metodologia envolveu a construção de uma aplicação monolítica com backend em Kotlin, integração com blockchain via Web3j e frontend em Angular. Para adaptação à realidade nacional, o modelo proposto prevê a utilização de cartórios como validadores da rede, garantindo a integridade e a autenticidade dos contratos armazenados. Os resultados apontam que a aplicação de *smart contracts* pode reduzir custos, minimizar riscos de fraude, automatizar transações e fortalecer a segurança dos registros imobiliários. Conclui-se que a adoção desta tecnologia é viável e pode representar um avanço significativo no sistema brasileiro de registro de imóveis.

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. Introdução

Um dos investimentos mais seguros e com alta rentabilidade é o mercado imobiliário, tanto no segmento residencial quanto no comercial. No entanto, como qualquer tipo de investimento, esse mercado apresenta vantagens e desvantagens, enfrentando desafios como a necessidade de inclusão de terceiros para verificação de informações, custos relacionados à administração, segurança dos investimentos, burocracia e transparência na posse da propriedade. Nesse contexto, as tecnologias da Indústria 4.0, em especial o uso de blockchain e contratos inteligentes (smart contracts), surgem como alternativas promissoras para a modernização do setor. A aplicação dessas tecnologias pode simplificar o processo de compra e venda de imóveis, reduzir a burocracia, eliminar a necessidade de intermediários e aumentar a segurança das transações, prevenindo, assim, a ocorrência de fraudes (Ahmad, Alqarni, Almazroi e Alam, 2020).

O blockchain pode ser definido como um sistema de registro compartilhado, seguro e de crescimento contínuo, no qual todos os participantes mantêm uma cópia dos registros, que só podem ser alterados mediante consenso entre as partes envolvidas. Estruturado como uma cadeia de blocos, cada bloco contém um conjunto de transações ou dados, possuindo um identificador único denominado hash, além de incluir o hash do bloco anterior, garantindo, dessa forma, a integridade sequencial das informações (Bashir, 2020). A introdução dessa tecnologia no mercado imobiliário visa justamente mitigar riscos de fraude, aumentando a confiabilidade e a transparência dos registros de propriedade.

Entre as plataformas baseadas em blockchain, o Ethereum destaca-se por sua natureza descentralizada, de código aberto e por oferecer suporte à criação de contratos inteligentes (smart contracts) e aplicações descentralizadas (dApps). Além de funcionar como uma máquina de estados única e uma máquina virtual, o Ethereum sincroniza e armazena as mudanças de estado do sistema, utilizando a criptomoeda Ether (ETH) para medir e restringir os custos operacionais das transações e execuções de código. Neste trabalho, a plataforma Ethereum será empregada para a implementação e execução dos contratos inteligentes que validarão as transações imobiliárias (Antonopoulos, 2018).

Os contratos inteligentes, por sua vez, são definidos como conjuntos de promessas expressas em formato digital, com protocolos que asseguram o cumprimento automático

das obrigações acordadas entre as partes. Em termos práticos, consistem em programas autoexecutáveis que operam na rede blockchain, ativando suas cláusulas automaticamente quando as condições previamente estabelecidas são atendidas, sem a necessidade de intervenção de terceiros, como advogados ou corretores. A imutabilidade, a transparência e a segurança desses contratos são garantidas pelas propriedades intrínsecas do blockchain, que impedem alterações posteriores ao seu registro. (Antonopoulos, 2018)

A implementação de contratos inteligentes é realizada por meio de linguagens de programação de alto nível, sendo a mais utilizada a linguagem Solidity, projetada especificamente para o desenvolvimento de smart contracts que operam na Ethereum Virtual Machine (EVM). Essa linguagem possibilita a definição de regras, lógicas de negócio, armazenamento de dados e interações financeiras de forma segura e imutável na blockchain.

Com o uso do Solidity, desenvolvedores podem criar contratos autoexecutáveis para viabilizar transações descentralizadas sem a necessidade de intermediários. O código-fonte dos contratos é compilado, gerando um bytecode que é interpretado e executado pela EVM, ambiente de execução responsável pelo processamento dos contratos na rede Ethereum.

Após a etapa de compilação, é necessário criar uma transação específica para implantar o contrato na blockchain, o que resulta na geração de um endereço único, utilizado em operações futuras, como o envio de fundos ou a invocação de funções. Um conceito central nesse processo é o GAS, unidade de medida que representa o esforço computacional necessário para executar operações na rede. Os usuários devem pagar, em Ether (ETH), o valor correspondente ao consumo de gas, remunerando os validadores e garantindo a continuidade e segurança da rede (ANTONOPOULOS, 2018).

Assim, este trabalho tem como objetivo desenvolver um sistema baseado em contratos inteligentes para validar transações de compra e venda de imóveis, com o propósito de otimizar os processos envolvidos, reduzir a burocracia, ampliar a eficiência e fortalecer a segurança jurídica no âmbito dos cartórios brasileiros.

2. Metodologia ou Material e Métodos

O sistema denominado WebSmartOffice, destinado ao processamento de transações de compra e venda de imóveis utilizando Smart Contracts, foi adotada uma arquitetura monolítica. A camada backend da aplicação foi implementada em Kotlin, uma linguagem de programação moderna, estática e de tipagem forte, compatível com o ecossistema Java e executada sobre a Java Virtual Machine (JVM). A escolha pelo Kotlin fundamentou-se em sua interoperabilidade com Java, bem como na robustez, segurança e ampla aceitação da JVM em sistemas de grande porte, características essenciais para garantir a escalabilidade e a confiabilidade do sistema. Adicionalmente, a vasta biblioteca de recursos disponíveis e o suporte multiplataforma foram fatores determinantes para essa decisão.

O backend foi estruturado utilizando o Spring Boot, ferramenta que facilita a configuração inicial do projeto e a integração de bibliotecas necessárias. A organização do projeto segue os princípios da Clean Architecture e as práticas de Clean Code, assegurando a correta definição e nomenclatura de classes, objetos, métodos e variáveis, além de promover um código de fácil manutenção e evolução.

Para a implementação da camada de persistência, foi utilizado o Spring Data, integrando a aplicação a um banco de dados relacional PostgreSQL. A escolha por um banco de dados SQL deve-se à sua robustez, segurança, natureza open source e ampla adoção em sistemas comerciais e pela comunidade de software livre. Considerando que a estrutura de dados do sistema seria rígida, optou-se por um banco relacional, minimizando a necessidade de alterações frequentes no modelo durante o desenvolvimento.

A autenticação e autorização de usuários foram implementadas por meio do Spring Security, tecnologia amplamente difundida, segura e de fácil assimilação, que garante o controle de acesso aos recursos do sistema.

No que se refere à gestão dos Smart Contracts, foi utilizado o Hardhat como ambiente de desenvolvimento, compilação e deploy, além da Hardhat Network para simular uma blockchain Ethereum local. A linguagem escolhida para a implementação dos contratos foi o Solidity, padrão de mercado para desenvolvimento na rede Ethereum. Devido à necessidade de valores reais para utilização das redes públicas de teste da Ethereum,

optou-se pela criação de uma rede privada local via Hardhat Network, utilizando contas Ethereum simuladas.

A integração entre os contratos inteligentes e a aplicação backend foi realizada utilizando a biblioteca Web3j, que possibilita a comunicação direta com blockchains Ethereum a partir de aplicações Java/Kotlin. Esta abordagem viabilizou a interação com os Smart Contracts em ambiente corporativo, preservando a robustez e escalabilidade da solução.

Para o desenvolvimento da camada frontend, foi utilizado o framework Angular, amplamente adotado no mercado, com vasta documentação, materiais de apoio e uma arquitetura robusta e segura. Complementarmente, foram utilizadas as tecnologias HTML, CSS, Bootstrap, JavaScript e TypeScript para construção da interface e da experiência do usuário.

O fluxo de desenvolvimento seguiu as seguintes etapas: inicialmente, foi construído o backend com todos os seus endpoints, utilizando o Swagger para a documentação e testes dos serviços. Em seguida, implementaram-se a autenticação e autorização no backend, expondo os endpoints via chamadas HTTP no padrão REST API. Posteriormente, foi desenvolvido o frontend com Angular, incluindo a criação de telas, rotas e o controle de acesso via autenticação.

Por fim, os contratos inteligentes foram implementados em Solidity, compilados utilizando o Hardhat, e integrados ao backend por meio da geração de classes Java utilizando a biblioteca Web3j, consolidando a comunicação entre as diferentes camadas do sistema.

3. Resultados e Discussão

Para o desenvolvimento do sistema WebSmartOffice, foram adotadas boas práticas de engenharia de software, tais como Clean Code (MARTIN, 2019), Arquitetura Limpa (MARTIN, 2018) e Domain-Driven Design (DDD) (EVANS, 2012). A aplicação dos princípios de código limpo contemplou a utilização de nomes autoexplicativos para variáveis e métodos, a eliminação de duplicidade de código, a criação de funções coesas, cada uma com uma única responsabilidade, bem como a inserção de comentários claros e relevantes, formatação consistente e tratamento de erros de forma estruturada conforme os princípios de Clean Code (Martin 2019).

A adoção da Arquitetura Limpa proporcionou a independência em relação a frameworks e tecnologias específicas, permitindo, por exemplo, que o WebSmartOffice não esteja restrito a um único sistema de banco de dados, o que viabiliza a sua substituição de forma simples, caso necessário. Além disso, as interfaces entre as camadas da aplicação foram claramente definidas, favorecendo a separação de responsabilidades, a legibilidade do código e a sua manutenibilidade.

Outro ponto fundamental foi o isolamento do domínio, em conformidade com o DDD, que estabelece o domínio e suas regras de negócio como núcleo central da aplicação, independente da tecnologia utilizada. Assim, as dependências são direcionadas sempre para o centro da arquitetura, reforçando a prioridade das regras de negócio em relação aos detalhes de implementação.

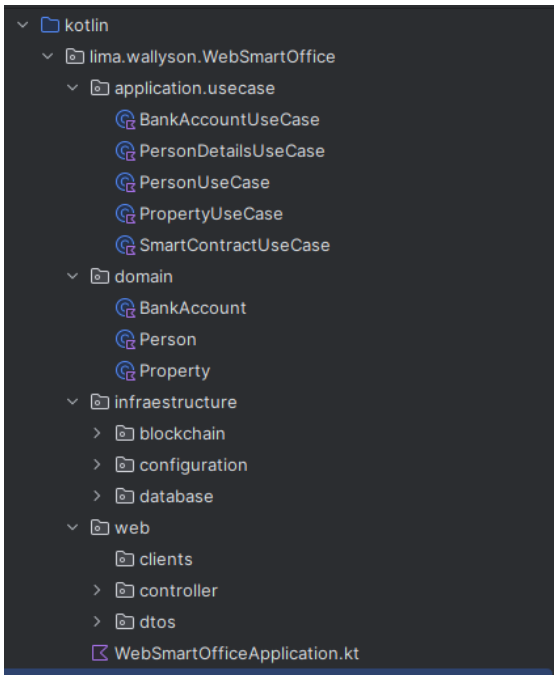


Figura 1 – Estrutura de diretórios seguindo Clean Architecture – autor (2025)

A adoção da arquitetura Clean Architecture oferece como principal vantagem a separação clara de responsabilidades entre as camadas do sistema. A lógica de negócio não depende de nenhum framework específico, uma vez que está isolada em uma camada própria. Essa abordagem permite que a aplicação seja estruturada em diretórios bem definidos, o que contribui significativamente para a organização do código, aumento da manutenibilidade, escalabilidade, testabilidade e legibilidade.

Ao utilizar o Domain-Driven Design (DDD) na API, temos a vantagem da aplicação ser orientada ao negócio, fazendo que o sistema reflita com precisão as regras e comportamentos e restrições do problema real.

Ao analisar o código apresentado na Figura 2, observa-se a aplicação de bons princípios de desenvolvimento de software, alinhados às diretrizes do SOLID. O componente **`PersonUseCase`** está adequadamente segregado na camada de aplicação, atuando como orquestrador entre serviços, repositórios e a transformação de DTOs (*Data Transfer*

Objects). Além disso, o design do código segue uma orientação ao domínio, assegurando que a lógica de negócio permaneça clara e isolada de detalhes de infraestrutura.

No que se refere às práticas de Clean Code, destaca-se a escolha de nomes expressivos e autoexplicativos: o método `getPersonByEmail` explicita de forma clara sua funcionalidade, o que contribui para a legibilidade e a facilidade de manutenção do código.A implementação demonstra ainda a aplicação de princípios fundamentais do SOLID.

O Princípio da Responsabilidade Única (SRP) é respeitado, pois o método realiza exclusivamente a operação de recuperação de uma pessoa pelo e-mail, estruturando a resposta em um DTO, enquanto delega a persistência de dados ao repositório e a lógica relacionada à conta bancária a um caso de uso específico.

Adicionalmente, observa-se o Princípio da Inversão de Dependência (DIP), uma vez que o PersonUseCase depende de abstrações, como PersonRepository, BankAccountRepository e PasswordEncoder, em vez de implementações concretas. Essa abordagem promove maior flexibilidade, testabilidade e facilita a substituição de componentes, reforçando a qualidade arquitetural do projeto.

```
@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()
        )
    }
```

Figura 2 – Boas práticas de Engenharia de Software aplicadas ao código – autor (2025)

Caso a API venha a expandir suas funcionalidades ou assumir novas responsabilidades, a arquitetura adotada favorece sua evolução e manutenção, em consonância com os princípios do SOLID, que visam tornar o software flexível, extensível e de fácil modificação.

Na implementação analisada, o diretório `application` abriga os casos de uso (use cases), responsáveis por orquestrar a lógica de negócio, seguindo o Princípio da Responsabilidade Única (SRP), ao isolar cada fluxo de trabalho em componentes específicos.

O diretório `domain` concentra os elementos centrais do domínio, como entidades e objetos de valor, alinhando-se ao Princípio da Inversão de Dependência (DIP), ao definir abstrações que independem de detalhes de infraestrutura.

Por sua vez, o diretório `infrastructure` reúne os componentes de infraestrutura, englobando configurações de integração com blockchain, arquivos de segurança implementados com Spring Security e conexões com o banco de dados. Essa separação clara entre domínio, aplicação e infraestrutura garante que as mudanças em tecnologias externas ou em mecanismos de persistência não impactem diretamente as regras de negócio, reforçando o Princípio de Aberto/Fechado (OCP), que estabelece que módulos devem ser abertos para extensão, mas fechados para modificação.

Essa estrutura modular e orientada a princípios sólidos da engenharia de software sustenta a escalabilidade, a manutenibilidade e a testabilidade da aplicação à medida que novos requisitos surgirem.

Por fim, o diretório `web` contém os controllers, responsáveis por expor os endpoints para o frontend, além dos DTOs (Data Transfer Objects) utilizados para transporte de dados entre as camadas. Essa organização modular segue os princípios da Clean Architecture e proporciona uma base sólida para o desenvolvimento de sistemas robustos e escaláveis.

Para realizar a autenticação e autorização da aplicação WebSmartOffice foi utilizado o Spring Security, na Figura 3 podemos verificar onde é criado a lógica da autenticação.

```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()
    }
```

Figura 3 – Método que realiza a autenticação no Spring Security – autor (2025)

No método `loadUserByUsername`, é realizada a busca no banco de dados por um usuário válido com o e-mail informado. Em seguida, o sistema verifica a senha armazenada (já criptografada) e define a role (perfil de acesso) do usuário. No contexto do sistema WebSmartOffice, existem apenas duas roles disponíveis: Admin e User. Após a validação, o método retorna uma instância representando o usuário autenticado.

Na Figura 4, é apresentada a configuração de autorização do Spring Security, responsável por definir as permissões de acesso às rotas da aplicação. Nessa etapa, é especificado se determinada rota requer autenticação prévia e se uma sessão de usuário deve ser criada para que o frontend possa utilizá-la durante a navegação.

```
@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()
}
```

Figura 4 – Autorização das rotas pelo Spring Security – autor (2025)

Como pode ser observado na Figura 5, determinadas requisições foram explicitamente autorizadas, como as rotas de login, logout e as relacionadas à documentação da API via Swagger, as quais não requerem autenticação. Para fins de simplificação no ambiente de desenvolvimento, as rotas com prefixo `/admin` e `/user` também foram temporariamente liberadas. Todas as demais requisições exigem que o usuário esteja autenticado.

```kotlin
@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")
    }
}
```

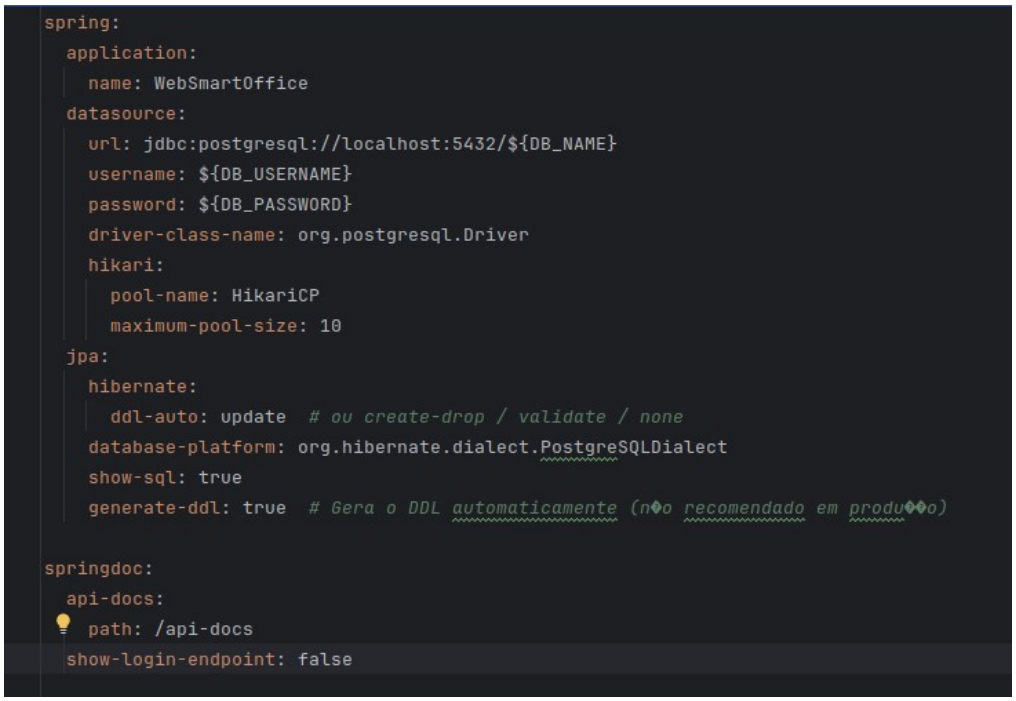
Figura 5 – Lógica da autenticação do usuário – autor (2025)

Na Figura 5, é apresentado o método responsável por expor a lógica de autenticação para o frontend, por meio da rota "`/auth/login`". Esse endpoint é responsável por validar as credenciais do usuário, gerar a sessão ou token correspondente e iniciar o contexto de segurança necessário para a navegação autenticada.

A utilização do Spring Security nesse contexto traz diversas vantagens para aplicações backend. Ele oferece um mecanismo robusto, flexível e extensível de autenticação e autorização, permitindo o controle refinado de acesso às rotas da API com base em roles, permissões e regras de segurança personalizadas. Além disso, o Spring Security possui integração nativa com padrões modernos de segurança, como JWT, OAuth2, sessões HTTP e criptografia de senhas com algoritmos seguros como Bcrypt.

Para criar a conexão com o banco de dados usamos o JPA e Hibernate, criando a conexão em um arquivo application.yml, salvamos o username e password do banco em uma váriavel de sistema para evitar vulnerabilidades na nossa api. No campo url passamos a string de conexão do banco, também indicamos o driver para fazer a conexão do banco no driver-class-name, este é um conceito de Injeção de dependência do Spring Data, não precisamos nos preocupar em gerenciar a conexão com o banco de dados, ele é injetado

automaticamente pelo Spring, desta forma poderíamos mudar rapidamente de banco, bastando mudar a string de conexão e o driver como mostrado na Figura 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
```

Figura 6 – Arquivo application.yml – autor (2025)

O Smart Contract desenvolvido neste trabalho foi implementado na linguagem Solidity, a principal linguagem de programação utilizada para a criação de contratos inteligentes na rede Ethereum. O contrato representa um modelo genérico de compra e venda de propriedade, podendo ser reutilizado em diversas transações desse tipo, bastando alterar os dados específicos do comprador, vendedor e do imóvel envolvido.

Como ilustrado na Figura 7, o contrato exige a definição de informações essenciais, como os endereços Ethereum do comprador e do vendedor, o valor do imóvel (em Ether), o tamanho da propriedade, o endereço do imóvel, o registro da propriedade e a escritura pública (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;
```

Figura 7 – Informações utilizadas no SmartContract – autor (2025)

O processo de utilização do Smart Contract envolve três etapas principais:

1. Compilação: O código Solidity é compilado com o uso de ferramentas como o Solc (Solidity Compiler), que geram os arquivos ABI e bytecode necessários para implantação.
2. Implantação na rede: O contrato é então enviado e registrado em uma blockchain Ethereum, neste caso estamos usando a rede HardHat.
3. Interação com o contrato: Após implantado, qualquer parte pode interagir com o contrato, desde que respeite as regras e condições definidas no código.

A lógica principal da transação está encapsulada no método `buyProperty()`, que executa a venda da propriedade. O valor da transação é representado em Wei, que é a menor unidade da criptomoeda Ether. O método valida se o valor enviado corresponde ao valor estipulado da propriedade, registra o comprador por meio de `PropertyPurchased`, transfere o montante ao vendedor usando a palavra-chave `payable`, e finaliza a operação emitindo o evento `SaleCompleted`, que confirma a conclusão da venda entre as partes, como vemos na Figura 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);
}
```

Figura 8 – Método buyProperty – autor (2025)

A escolha pela utilização da linguagem Solidity e da rede Ethereum se justifica por diversos fatores. A Ethereum é a principal plataforma para execução de contratos inteligentes, oferecendo segurança, descentralização, imutabilidade e ampla adoção no mercado. Já o Solidity é uma linguagem orientada a contratos, com sintaxe inspirada em linguagens tradicionais como JavaScript e C++, possuindo vasto suporte da comunidade, documentação extensa e compatibilidade direta com o ecossistema Ethereum, o que facilita o desenvolvimento, teste e auditoria de contratos inteligentes confiáveis.

Para realizar a conexão com o frontend, foram utilizadas chamadas REST API sobre o protocolo HTTP. A interface do sistema foi desenvolvida com o Angular 17, utilizando HTML, CSS, JavaScript e TypeScript, tecnologias amplamente consolidadas no desenvolvimento de aplicações web modernas. O Angular foi escolhido por ser um framework amplamente usado, robusto e escalável, que oferece suporte nativo a componentes reutilizáveis, injeção de dependênica, roteamento, formulários reativos e integração facilita com APIs.

A estrutura do frontend foi organizada seguindo a convenção proposta pelo próprio Angular, adotando o uso de componentes reutilizáveis para promover modularidade, manutenção facilitada e reaproveitamento de código. Conforme ilustrado na Figura 9, o projeto foi dividido em diretórios específicos com responsabilidades bem definidas.

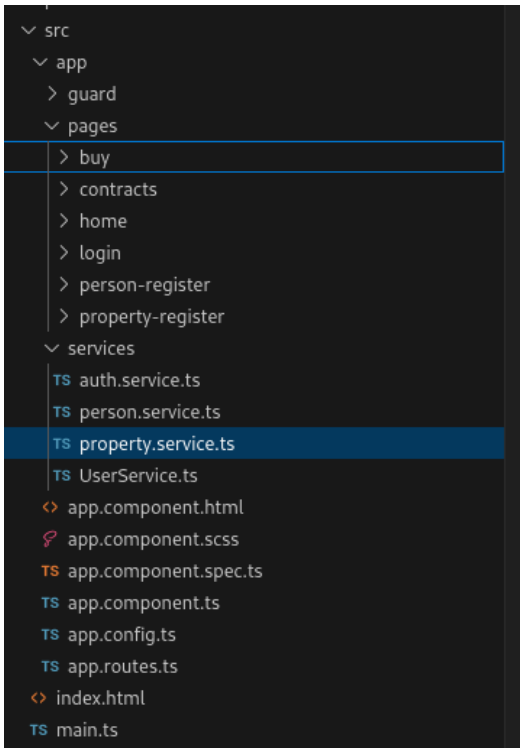


Figura 9 – Organização de estrutura de diretórios própria do Angular – autor (2025)

O diretório de páginas contém os componentes responsáveis por cada funcionalidade da aplicação, onde cada página é composta por um arquivo `.html` (estrutura visual), um arquivo `.css` (estilização) e um arquivo `.ts` (responsável pela lógica e comportamento da interface, implementado em TypeScript). Além disso, há um diretório chamado services, que centraliza a lógica de negócio do frontend e realiza a comunicação com a API backend por meio de requisições HTTP. Essa separação permite que a lógica de consumo de dados fique desacoplada da camada de apresentação, promovendo coesão e reutilização. Na raiz do projeto encontra-se o diretório `app`, que representa o núcleo da aplicação. Nele estão localizados o componente principal (`AppComponent`), o sistema de roteamento que gerencia a navegação entre as páginas, e os arquivos de configuração globais da aplicação.

Essa estrutura modular, fundamentada na arquitetura baseada em componentes do Angular (ANGULAR, 2025) e alinhada aos princípios da Clean Architecture (MARTIN, 2017), contribui de forma significativa para a escalabilidade, a manutenibilidade e a clareza do

projeto frontend. A segmentação da aplicação em componentes reutilizáveis e independentes favorece a separação de responsabilidades, tornando o código mais organizado, previsível e facilitando sua evolução em contextos de expansão funcional ou manutenção contínua.

A tecnologia de *Smart Contracts* permite que contratos sejam firmados diretamente entre contratante e contratado, eliminando a necessidade de intermediários, como instituições bancárias e cartórios. Como exemplificado no trabalho de (Ahmad, Alqarni, Almazroi e Alam, 2020), é possível realizar operações, como compra e venda de imóveis ou aluguel, de forma direta entre as partes. Contudo, ao considerar a realidade brasileira, a completa ausência de intermediários seria uma proposta utópica, dada a obrigatoriedade legal da atuação dos cartórios. Dessa forma, a arquitetura proposta foi pensada considerando o sistema brasileiro de cartórios.

No modelo sugerido, a execução dos *Smart Contracts* seria realizada pelos cartórios. A rede Ethereum funcionaria como uma rede distribuída, onde, a cada novo contrato firmado, um nó seria adicionado à rede. Assim, cada cartório possuiria sua própria cópia dessa rede distribuída, conforme ilustrado na Figura 10.

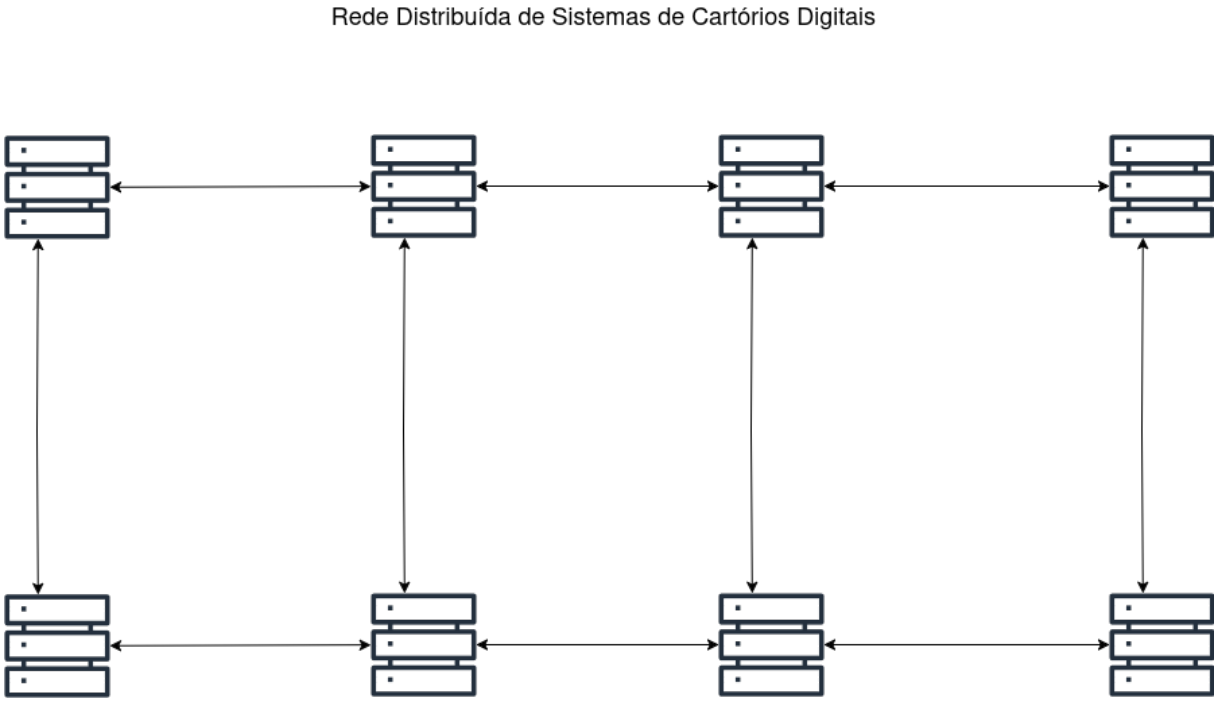


Figura 10 – Sistema de Rede Ethereum distribuído entre cartórios – autor (2025)

Para cada tipo de operação realizada em cartório, seria necessário desenvolver um modelo específico de contrato. Por exemplo, contratos de compra e venda de imóveis, contratos de aluguel ou de compra de veículos utilizariam modelos padronizados, alterando apenas informações específicas das partes envolvidas.

No sistema WebSmartOffice — um aplicativo projetado para simular a operação de um cartório —, o usuário realiza a autenticação, visualiza imóveis disponíveis, seleciona um deles e assina o contrato. A seguir, o contrato é implementado (*deployed*) com as informações fornecidas e inserido na rede Ethereum. Quando o usuário aciona a opção "pagar", o contrato é efetivamente executado: ocorre a verificação do saldo do comprador, a transferência de valores entre as contas Ethereum e, em seguida, a transferência de posse do bem. O contrato é então finalizado, e um bloco contendo o registro da transação é adicionado à rede.

Devido a limitações de tempo e escopo, no WebSmartOffice todas essas etapas (seleção do imóvel, transferência de valores e transferência de posse) são realizadas no próprio sistema. No entanto, em uma implementação realista, haveria um sistema especializado para compra e venda de imóveis, o cartório de notas seria responsável pela criação do *Smart Contract*, e uma instituição bancária gerenciaria a transferência de valores.

Adaptando o modelo para o contexto brasileiro, poderíamos prever *Smart Contracts* com prazo de validade determinado, sendo a transferência de valores executada automaticamente ao término do prazo. O contrato armazenaria os dados bancários do usuário, e a operação poderia ser realizada em moeda digital oficial, como o DREX (Real Digital), utilizando sistemas de pagamento como o PIX. Essa abordagem traria diversas vantagens: redução de custos cartorários, maior segurança jurídica (uma vez que eliminaria a necessidade de transferências manuais, que são suscetíveis a fraudes), economia de papel, diminuição da burocracia e maior resiliência dos registros — visto que todos os cartórios manteriam uma cópia completa da rede. Em caso de perda de dados em um cartório, bastaria restaurá-los a partir de qualquer outro cartório da rede. Além disso, a falsificação de contratos seria dificultada, já que seria necessário corromper toda a rede Ethereum dos cartórios para alterar um bloco válido.

As Figuras 11 e 12 demonstram, no WebSmartOffice, a assinatura de um contrato e o hash da transação armazenado após sua execução.

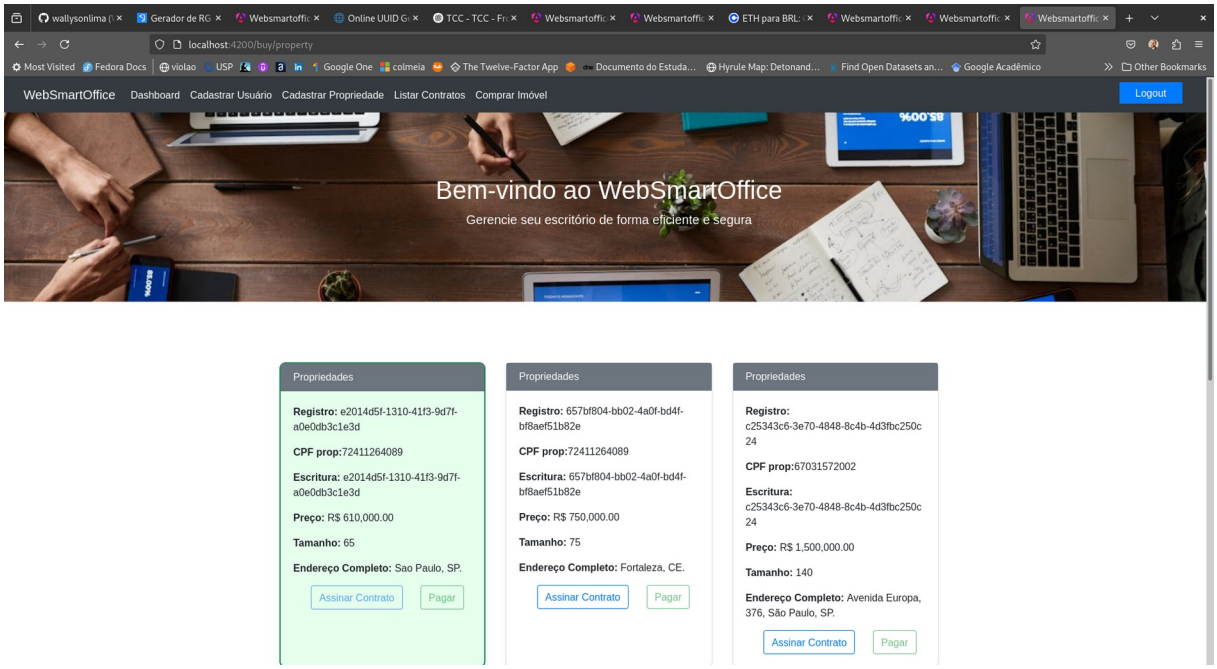


Figura 11 – Assinatura e Pagamento de um contrato de compra de imóvel – autor (2025)

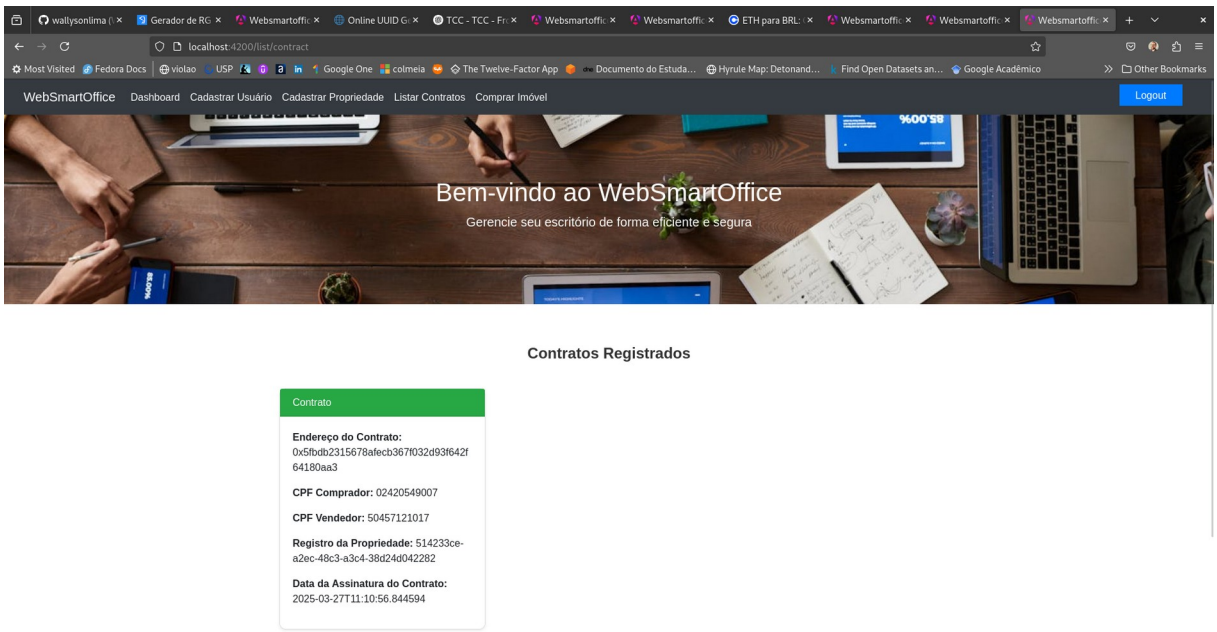


Figura 12 – Exemplo de um Hash de Contrato Salvo – autor (2025)

4. Conclusão(ões) ou Considerações Finais

Esta pesquisa permitiu identificar as tecnologias necessárias para implementar transações de compra e venda de imóveis utilizando *Smart Contracts*. Foi proposta uma arquitetura adaptada à realidade brasileira, apontando as vantagens que essa abordagem traria para o sistema de cartórios: aumento da segurança jurídica, redução de custos, diminuição da burocracia e automatização dos processos de compra e venda de imóveis. Também foram utilizadas e integradas diversas tecnologias estudadas ao longo do MBA em Engenharia de Software, como desenvolvimento de APIs, *frontend*, autenticação/autorização, *blockchain*, Ethereum e Solidity, entre outras.

Como sugestão para trabalhos futuros, propõe-se a implementação de outros tipos de contratos — como contratos de aluguel de imóveis, compra de veículos, aquisição de terras ou sociedades empresariais. Para isso, bastaria definir um modelo específico de contrato para cada tipo de operação. Também se vislumbra a possibilidade de desenvolver uma arquitetura baseada em microsserviços, com módulos dedicados para compra e venda de imóveis, operações bancárias e gestão cartorária.

5. Agradecimento

Agradeço ao meu orientador, Alexander Chaves, pela orientação e pela escolha do tema de pesquisa. Agradeço também à minha família, sem a qual este trabalho não teria sido possível, especialmente à minha mãe, que sempre me incentivou nos estudos, e à minha esposa, que compreendeu as abdicações necessárias durante o desenvolvimento deste projeto. Por fim, expresso minha gratidão ao sistema público de educação, responsável por toda a minha formação acadêmica.

6. Referências

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.