Capítulo 7

Desenvolvimento de aplicações com colaboração síncrona utilizando o padrão arquitetural REST

Laurentino Augusto Dantas[1,2], Maria da Graça C. Pimentel[2]

[1]Instituto Federal de Mato Grosso do Sul (IFMS) – Naviraí-MS, Brazil

[2]Universidade de São Paulo (USP) – São Carlos-SP, Brazil

laurentino.dantas@ifms.edu.br, mgp@icmc.usp.br

Abstract

In this tutorial, we discuss the development of applications that support synchronous collaboration using the REST architecture. After presenting related work, we introduce the Model for Supporting Synchronous Collaboration in REST API (MoSCoR) that guides the adaptation of REST systems to enable real-time collaboration. Next, we discuss how MoSCoR was created to implement synchronous collaboration functionalities in the ESPIM platform authoring tool. As a didactic example for the tutorial, we exemplify the development of a product registration application that employs REST and is implemented using Spring Boot in the back-end and Angular in the front-end. In this application, we use a WebSocket as a communication channel to notify all users, via broadcast, whenever a new record is added. Finally, as an alternative to MoSCoR, we indicate examples of libraries that can be used to create collaborative applications.

Resumo

Neste tutorial, discutimos o desenvolvimento de aplicações com suporte à colaboração síncrona utilizando a arquitetura REST. Depois de apresentar trabalhos relacionados, introduzimos o Modelo de Suporte à Colaboração Síncrona em API REST (MoSCoR) que orienta a adaptação de sistemas REST para possibilitar a colaboração em tempo real. A seguir, discutimos como o MoSCoR que foi criado para a implementação de funcionalidades de colaboração síncrona na ferramenta de autoria da plataforma ESPIM. Como exemplo didático para o tutorial, exemplificamos o desenvolvimento de uma aplicação de cadastro de produto que emprega REST e é implementada utilizando Spring Boot no back-end e Angular no front-end. Nessa aplicação, utilizamos um WebSocket como canal de comunicação para notificar todos os usuários, via broadcast, sempre que um novo registro é adicionado. Por fim, como alternativa ao MoSCoR, indicamos exemplos de bibliotecas que podem ser utilizadas para criação de aplicações colaborativas.

7.1. Introdução

Na web, a colaboração síncrona remota é uma opção cada vez mais acessível a usuários. Por exemplo, em 2024 o *Google Workspace* conta com mais de três bilhões de usuários. Dentre as aplicações do *Google Workspace* que permitem colaboração síncrona, o *Google Docs* tem mais de um bilhão de usuários mensais ativos, enquanto *Google Sheets* e o *Google Slides* têm, respectivamente, 900 milhões e 800 milhões de usuários ativos mensalmente.[1] Apesar de nem todos esses usuários utilizarem recursos de colaboração síncrona, essas estatísticas ilustram o alcance potencial de ferramentas de colaboração apoiadas em interações em tempo real. De fato, considerando explicitamente colaboração síncrona, o *Google Meet* tem 300 milhões de usuários ativos mensalmente.

Editores colaborativos síncronos são um exemplo típico de aplicações que implementam a colaboração síncrona, permitindo que múltiplos usuários editem documentos simultaneamente em tempo real. A edição colaborativa em tempo real (RTCE) é uma característica desejável para muitos sistemas, pois permite que múltiplos usuários trabalhem simultaneamente em um mesmo documento ou projeto sem conflitos ou perdas de dados. Um editor colaborativo em tempo real fornece uma interface de edição onde um grupo de usuários, em diferentes locais, pode visualizar e editar o mesmo documento simultaneamente, com todas as modificações sendo propagadas e exibidas em tempo real para todos os participantes (Bath et al., 2022). Os editores colaborativos em tempo real (RCE) baseiam-se na replicação de dados compartilhados para garantir que as alterações feitas por um usuário sejam imediatamente propagadas para os demais. A literatura descreve diversos algoritmos de sincronização que asseguram a consistência dos dados entre as várias réplicas de um documento, permitindo que múltiplos usuários editem simultaneamente sem conflitos (Alsulami and Cherif, 2017). Com base nesses algoritmos, foram propostos vários tipos de RCEs para edição de texto (Nédelec et al., 2013) e *rich text* (Litt et al., 2022a), imagens (Bath et al., 2022), objetos JSON (Jungnickel and Herb, 2016), documentos PDF (Katayama et al., 2013) e objetos 3D (Salvati et al., 2015), entre outros.

O padrão arquitetural REST, proposto para a web por Fielding (2000), é adotado atualmente por mais de 83% dos desenvolvedores de software (Postman, 2023). Considerando os benefícios proporcionados pela RTCE, aplicações REST que atualmente oferecem edição individual poderiam ser refatoradas para suportar a colaboração em tempo real, permitindo que múltiplos usuários trabalhem simultaneamente e de forma eficiente no mesmo documento.

A partir de uma revisão sistemática envolvendo algoritmos e trabalhos que apoiam RCEs (Dantas et al., 2024), propusemos o Modelo de Suporte à Colaboração Síncrona em API REST (MoSCoR) — um conjunto de princípios, restrições e regras — para orientar a refatoração de aplicações arquitetural REST para que se tornem aplicações com suporte à colaboração síncrona (Dantas, 2024).

A relevância da aplicação do MoSCoR no contexto RESTful é destacada pelo seu alinhamento com a arquitetura da World Wide Web e pela sua ampla adoção. Ao estender o padrão arquitetural REST, o MoSCoR facilita a integração de funcionalidades colaborativas em aplicações web, sem a necessidade de mudanças estruturais complexas e preservando as características básicas do sistema.

[1] `https://explodingtopics.com/blog/google-workspace-stats`

No restante deste texto, discutimos conceitos fundamentais (Seção 7.2), apresentamos o MoSCoR (Seção 7.3) e registramos como o sistema ESPIM motivou a sua criação (Seção 7.4). Como exemplo didático para este tutorial, detalhamos na Seção 7.5 o desenvolvimento de uma aplicação de cadastro de produto: a aplicação emprega REST, é implementada utilizando *Angular* no *frontend* e *Spring Boot* no *backend*, e adota um *WebSocket* como canal de comunicação para notificar todos os usuários, via *broadcast*, sempre que um novo registro é adicionado. Na Seção 7.6, como exemplo de biblioteca disponível para a construção de aplicações colaborativas síncronas na web, fazemos referência ao framework *Yjs*. Apresentamos nossas considerações finais na Seção 7.7.

7.2. Fundamentos

7.2.1. Representational State Transfer (REST)

O padrão arquitetural *Representational State Transfer* (REST), formalizado na tese de doutorado de Fielding (2000) sobre seu trabalho no World Wide Web Consortium (W3C), desempenha um papel crucial na evolução e no funcionamento da web (Fielding et al., 2017). REST é um padrão arquitetural que define um conjunto de restrições e práticas para a criação de serviços web escaláveis e interoperáveis, tendo sido desenvolvido com base na experiência de Fielding (2000) em especificações como o *Hypertext Transfer Protoco*l (HTTP), a definição da sintaxe genérica dos *Uniform Resource Identifiers* (URIs) e a proposta de padrões para *Relative Uniform Resource Locators* (URLs).

Aplicações web que implementam o padrão arquitetural REST, chamadas serviços web RESTful, seguem os princípios fundamentais do padrão arquitetural e restrições correspondentes, indicados no Quadro 1, e utilizam protocolos web convencionais como o HTTP. Essas APIs consistem em um conjunto de *endpoints*, sendo cada *endpoint* associado a uma funcionalidade específica implementada em um processo de negócios. APIs RESTful são geralmente acessíveis pelos métodos de requisição GET, POST, PUT e DELETE do protocolo HTTP e invocadas por meio de URIs.

Na arquitetura básica de uma API REST ilustrada na Figura 7.1, a aplicação cliente acessa a API REST a partir de um *endpoint*. Devido ao fato de sistemas RESTful não manterem uma conexão permanente entre o cliente e o servidor e todas as requisições partirem do cliente, eles não oferecem suporte à colaboração síncrona. Em um ambiente colaborativo síncrono, é crucial que as ações realizadas por um usuário sejam disseminadas para os outros usuários por meio de um canal de comunicação bidirecional que mantenha uma conexão permanente entre o cliente e o servidor. Além disso, para garantir a consistência entre os diversos usuários que editam um documento simultaneamente, é imperativo o uso de algoritmos de controle de simultaneidade (Gadea, 2021).

7.2.2. Concorrência, resolução de conflitos e sincronização em tempo real

Um aspecto crucial na edição colaborativa é a gestão de operações concorrentes, a resolução de conflitos e a sincronização em tempo real em ambientes distribuídos. A colaboração síncrona demanda algoritmos de controle de consistência otimista para viabilizar a colaboração em tempo real. Gadea (2021) observa que o design e a implementação desses algoritmos apresentam desafios significativos. Ao longo de mais de três décadas de pesquisa em edição colaborativa, diversas soluções técnicas e teóricas foram desenvolvi-

Quadro 1: Princípios de Engenharia de Software (ESw); Princípios REST e Restrições correspondentes (adaptado de (Fielding, 2000))

Princípios ESw	**Princípios REST**	**Restrições dos Princípios REST**
Cliente-Servidor	*Client-Server Architecture*	Separação de responsabilidades entre cliente e servidor para permitir a escalabilidade independente.
Visibilidade, Confiabilidade e Escalabilidade	*Statelessness*	Cada requisição do cliente para o servidor deve conter todas as informações necessárias para entender e processar o pedido.
Eficiência	*Cacheability*	Respostas devem ser explicitamente rotuladas como cacheáveis ou não-cacheáveis para evitar operações repetitivas.
Generalidade	Uniform Interface	Interface uniforme entre componentes, simplificando a arquitetura e a interação.
Escalabilidade	*Layered System*	O sistema pode ser composto de camadas hierárquicas, sendo que cada camada não precisa ter conhecimento sobre as demais além da interface.
Extensibilidade	*Code on Demand* (opcional)	Servidores podem transferir código executável para clientes sob demanda, permitindo funcionalidades sob demanda.

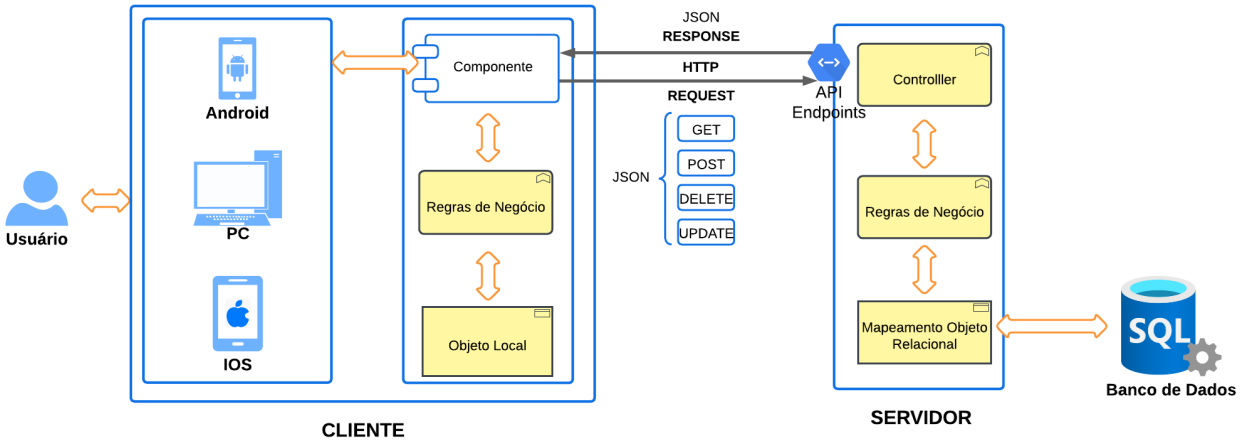

Figura 7.1: Representação básica da arquitetura REST

das, resultando em duas principais famílias de abordagens para o controle de consistência otimista com preservação de intenções: Transformação Operacional (OT - *Operational Transformation*) (Ellis and Gibbs, 1989) e Tipos de Dados Replicados Comutativos (CmRDTs - *Commutative Replicated Data Types*) Oster et al. (2006). Ainda nesse contexto, Shapiro et al. (2011) observam que um algoritmo específico CmRDT é referenciado como *Conflict-free Replicated Data Type* (CRDT).

O conjunto de três estudos conduzidos por Sun et al. (2020a,b,c) evidencia que tanto OT quanto CRDT apresentam características que os qualificam para diferentes contextos de aplicação em sistemas de edição colaborativa. As análises comparativas dos autores abrangem aspectos fundamentais como manutenção de consistência, correção, complexidade temporal e espacial, além da implementação e suporte à coedição ponto a ponto. Os resultados obtidos refutam as colocações de outros autores relativametne à superioridade universal dos CRDTs em relação aos OTs. Sun et al. (2020a,b,c) demonstram que as diferenças entre as duas abordagens são ortogonais e dependem intrinsecamente dos requisitos específicos e do contexto operacional em questão. Adicionalmente, a noção de que os CRDTs são particularmente adequados para ambientes de coedição ponto a ponto foi demonstrada ser infundada, indicando que tal percepção carece de respaldo empírico robusto. Consequentemente, as decisões referentes à adoção de OT ou CRDT devem ser fundamentadas em uma análise criteriosa das necessidades específicas do sistema e das condições operacionais, ao invés de se basearem em pressupostos de superioridade técnica geral. Assim, Sun et al. (2020a,b,c) enfatizam a relevância de uma abordagem contextualizada na seleção da metodologia mais apropriada para a manutenção da consistência em ambientes colaborativos.

7.2.3. Aplicações colaborativas: arquiteturas, algoritmos e aplicações

A edição colaborativa em tempo real tem sido um campo de estudo amplamente explorado, com diversas abordagens e soluções propostas para diferentes contextos e tipos de dados. A partir de uma revisão sistemática sobre edição colaborativa síncrona em tempo real, identificamos três temas recorrentes: arquitetura e infraestrutura, algoritmos de sincronização e gestão de conflitos, e aplicações específicas (Dantas et al., 2024).

Diversos trabalhos se concentraram na concepção de arquiteturas escaláveis e eficientes para suportar a edição colaborativa, como levantado por Dantas et al. (2024). Por exemplo, Bath et al. (2022) desenvolveram um aplicativo web utilizando uma arquitetura cliente-servidor com WebGL e WebSocket, garantindo renderização interativa e sincronização em tempo real para edição de imagens *raster* e vetoriais. De forma similar, Nédelec et al. (2016) utilizaram WebSockets para a criação do CRATE, um sistema para a edição colaborativa de narrativas literárias. Inoue et al. (2012) e Ozono et al. (2012) desenvolveram o WFE e o WFE-S, sistemas de edição colaborativa de páginas web com suporte a ambientes de computação em nuvem, destacando a importância da escalabilidade.

Entre os autores utilizam CRDT, Nédelec et al. (2016) empregaram CRDT na criação do editor CRATE para a edição de narrativas literárias e (Litt et al., 2022a) na criação de seu editor *rich text*. Ainda, Kleppmann (2020) apresenta um algoritmo para operações de movimentação de elementos em CRDTs de listas, e Litt et al. (2022b) tratam da edição de texto rico. Vários outros estudos propõem o uso de CRDT (Galesky and Rodrigues, 2023; Yanakieva et al., 2023; Bauwens et al., 2023; Da and Kleppmann,

2024; Laddad et al., 2022; Jeffery and Mortier, 2023). Especificamente em termos de frameworks, exemplos são o *YJS* (Jahns, 2018b), para compartilhamento de dados, e *Hocuspocus* (Jahns, 2018a), para edição colaborativa.

Entre os estudos que envolvem OT, Kuiter et al. (2021) desenvolveram o variED, um editor baseado em uma infraestrutura distribuída que utiliza OT para gerenciar operações concorrentes, essencial para o desenvolvimento de software orientado a linhas de produtos. Alsulami and Cherif (2017) adaptaram algoritmos de OT para redes oportunísticas, ajustando-os à natureza assíncrona das comunicações. Jungnickel and Herb (2016) aplicaram OT à edição simultânea de objetos JSON, enquanto Fechner et al. (2015) utilizaram OT para a edição colaborativa de dados geográficos no Ethermap.

Frente à demanda atual por ferramentas colaborativas em tempo real e visando permitir que vários usuários colaborem simultaneamente para construir e editar ontologias, Hemid et al. (2024) desenvolveram uma ferramenta web para colaboração em tempo real, compatível com GitLab, GitHub e Bitbucket, integrada a um banco de dados em tempo real baseado em OT. Visando aprimorar a colaboração entre equipes de ciência de dados, Wang et al. (2024) propuseram um modelo de edição colaborativa com três níveis de proteção para evitar erros na edição colaborativa em tempo real em notebooks computacionais utilizados por cientistas de dados. As medidas permitiram o compartilhamento imediato de edições e estado de *runtime*, melhorando o contexto compartilhado, explicações, custos de comunicação e reprodutibilidade. Virdi et al. (2023) descrevem a metodologia detalhada para a análise tecnológica e a implementação de um protótipo de editor de código colaborativo abordando a arquitetura de design.

Vários trabalhos focaram em aplicações específicas para a edição colaborativa, como identificado por (Dantas et al., 2024). Por exemplo, Nicolaescu et al. (2018) introduziram o SyncMeta, que facilita negociações e análises de impacto a partir de múltiplas perspectivas para stakeholders geograficamente distribuídos. Katayama et al. (2013) desenvolveram uma aplicação para edição colaborativa de documentos PDF, utilizando HTML5 e WebSocket para comunicação bidirecional. Lautamäki et al. (2012) apresentaram o CoRED, um editor colaborativo em tempo real para aplicações web Java. Shen and Sun (2002) propuseram o sistema REDUCE para edição colaborativa com destaque de texto em tempo real, enquanto Salvati et al. (2015) criaram o MeshHisto para modelagem colaborativa de objetos 3D. Thum et al. (2009) descreveram o SLIM, um ambiente de modelagem colaborativa síncrona focado em modelos de sistemas de software, utilizando OT para gerenciar operações concorrentes e garantir consistência e integridade do modelo. Wei et al. (2009) criaram um ambiente colaborativo de edição científica para a química, incorporando funcionalidades específicas para edição em tempo real de documentos científicos complexos, com controle de versões e ferramentas para análise e visualização de dados químicos. De Lucia et al. (2007) apresentaram STEVE, uma ferramenta para modelagem colaborativa síncrona em UML, com gerenciamento de versões e resolução de conflitos.

7.3. Modelo de Suporte à Colaboração Síncrona em API REST (MoSCoR)

7.3.1. Comunicação bidirecional e gerenciamento de canais

Para manter os documentos de todos os clientes sincronizados e garantir a integridade do histórico de alterações, é fundamental estabelecer um mecanismo que notifique os editores dos usuários sempre que ocorrerem modificações.

Deve-se criar um canal de comunicação que mantenha cliente e servidor conectados, permitindo a troca contínua de mensagens em ambas as direções, essencial para manter os editores sincronizados. Quando ocorrem alterações, o servidor pode notificar imediatamente todos os outros editores conectados, assegurando que todos os usuários tenham acesso às últimas atualizações.

Quando um cliente se conecta a um canal predeterminado, ele pode trocar mensagens bidirecionalmente com outros clientes no mesmo canal, proporcionando uma troca contínua e instantânea de dados em tempo real.

No MoSCoR, os canais de comunicação bidirecional devem seguir as regras:

1. Devem ser definidas diretrizes claras para a criação e formatação das mensagens de cada canal, bem como para o tratamento dos dados recebidos pelo canal, tanto pelo servidor quanto pelos clientes.
2. As estruturas das mensagens trocadas em cada canal devem ser reconhecidas por todos, com regras associadas para sua geração.
3. As regras para o tratamento dos dados recebidos devem incluir processos como validação, filtragem e encaminhamento adequado das informações, garantindo a integridade e segurança do sistema.

O Quadro 2 explicita, com base no discutido nesta seção, o relacionamento entre princípios de Engenharia de Software e entre princípios e restrições para Edição Colaborativa em Tempo Real.

7.3.2. Controle de histórico, de versões e sincronização de documentos

Para suporte à sincronização, no MoSCoR foi desenvolvida um solução baseada no algoritmo Jupiter (Nichols et al., 1995), originalmente concebido para oferecer suporte à colaboração remota e tratar o controle de concorrência. No algoritmo Jupiter, que utiliza OT, o controle de concorrência é realizado por meio de um servidor centralizado que serializa as operações e as distribui em modo *broadcast*.

Para manter a serialização das operações, o MoSCoR segue o padrão de ordem de chegada no servidor, visto que manter uma estrutura para garantir qual ação foi originada primeiro é um dos grandes desafios dos algoritmos de concorrência. Desta forma, o controle de concorrência fica centralizado no servidor. Em um ambiente de edição colaborativa síncrona com poucos editores trabalhando simultaneamente, essa abordagem pode ser eficaz e simples, minimizando a complexidade do sistema. A centralização facilita a coordenação e a resolução de conflitos de maneira direta e eficiente. Embora essa estratégia possa introduzir gargalos de desempenho e um ponto único de falha, esses riscos são mitigados pela baixa quantidade de editores, tornando a centralização uma escolha prática e funcional para este cenário específico. No entanto, para garantir escalabilidade

Quadro 2: Princípios de Engenharia de Software (ESw); Princípios e Restrições para Edição Colaborativa em Tempo Real (RTCE)

Princípios de ESw para RTCE	**Princípios RTCE**	**Restrições para RTCE: MoSCoR**	Implementação RESTful
Escalabilidade e Performance	Comunicação bidirecional	Comunicação bidirecional por conexão permanente: melhor utilização de recursos de rede e resposta em tempo real	*WebSockets*, ou alternativa, para comunicação bidirecional e atualização em tempo real
Modularidade e Manutenibilidade	Gerenciamento de canais	Criação e gerenciamento de canais: estrutura organizada e gerenciável para diferentes fluxos de comunicação	*Endpoints* RESTful para criação, gerenciamento e manipulação de canais
Integração Contínua e Colaboração	Controle de concorrência	Controle de concorrência e mesclagem das alterações: manutenção da coesão do sistema e suporte a múltiplos usuários simultâneos sem conflitos	Algoritmos de controle de concorrência como Júpiter ou alternativa
Consistência e Confiabilidade	Controle de histórico e versões	Controle de histórico de edições, versões e sincronização de documentos: garantia de integridade dos dados e capacidade de recuperação de estados anteriores	*Endpoints* RESTful para gerenciamento de versões, auditoria e recuperação de histórico de edições

e robustez em cenários com um maior número de editores, deve-se investigar alternativas entre OT e CRDT como indicado por Sun et al. (2020a,b,c).

Todo documento editado tem associado a ele um registro de histórico e um número inteiro que indica a sua versão, toda atualização no programa é registrado de forma sequencial no histórico, o número da versão do programa sempre será o número sequencial da última operação realizada. Quando recebe uma operação, o servidor salva as informações no programa, atualiza o histórico e número de versão do programa, a operação então é numerada e após isso é enviada em *broadcast* para todos que estiverem editando o programa naquele momento.

Além disso, no modelo de dados devem ser implementadas classes para o processo de desfazer e refazer, além de um registro de usuários. A Figura 7.2 apresenta um diagrama de classes, associadas a um documento, que permitirão o suporte à colaboração síncrona. No modelo está previsto, além do histórico, suporte à comunicação entre os usuários, possibilidade de comentários e implementação do *Undo/Redo*.

7.3.3. Controle de concorrência e mesclagem

O controle de concorrência e mesclagem em ambientes colaborativos é extremamente complexo e importante. Entretanto, a abordagem adotada no modelo, na qual a serialização é controlada pelo servidor, a ordem das operações é definida pela ordem de chegada da operação no servidor, faz com que o processo de mesclagem seja simplificado.

Existem duas situações básicas nas quais o processo de mesclagem se faz necessário: 1) Quando duas operações são geradas no mesmo momento por dois usuários distintos; e 2) Quando um dos usuários que está editando um documento perde a conexão

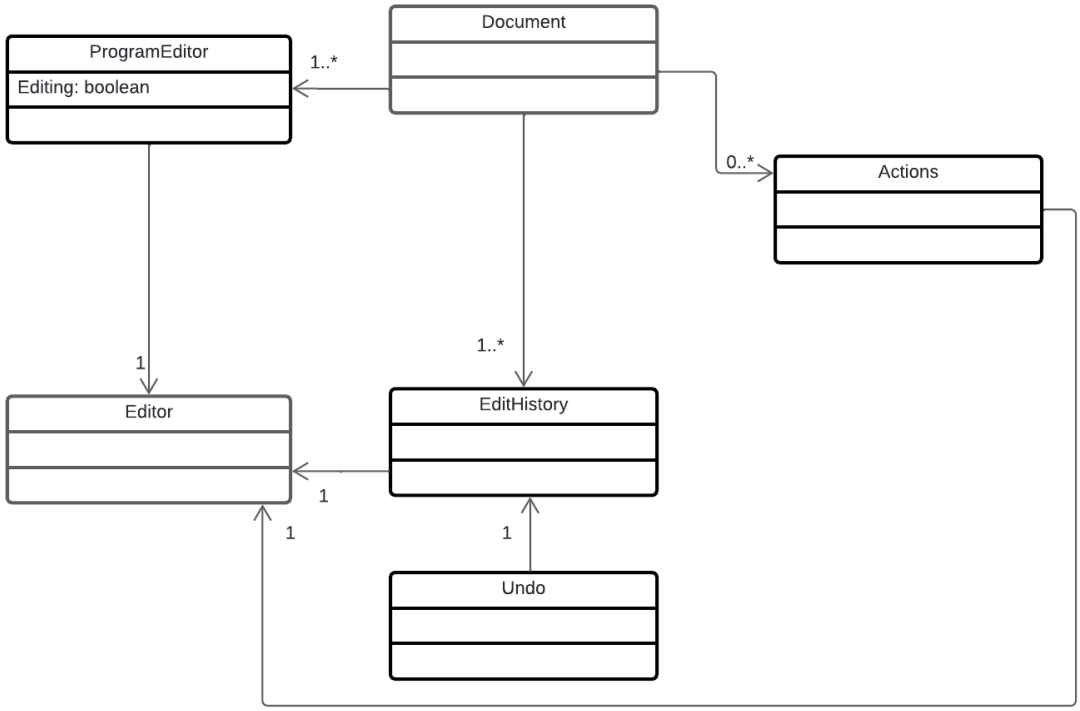

Figura 7.2: Modelo de Suporte à Colaboração Síncrona em API REST (MoSCoR): diagrama de classes: histórico, comunicação entre os usuários, comentários e *Undo/Redo*

por um tempo e realizar as alterações localmente as operações que são enviadas posteriormente ao servidor.

Na situação em que duas operações foram geradas por dois usuários distintos ao mesmo tempo, o servidor irá serializá-las e irá processá-las na ordem na qual estão serializadas, caso as duas operações não causem conflitos entre si, ambas serão executada, caso a primeira operação a ser executada torne a segunda operação inválida, a segunda operação não será executada e será gerado um erro que será enviado ao cliente que originou a operação.

No MoSCoR, está prevista a situação na qual um usuário pode continuar editando o documento quando perder a conexão com o servidor, nesses casos, quando reestabelecer a conexão com o servidor todas as alterações feitas no documento local devem ser enviadas ao servidor, para que as alterações sejam efetivadas.

Quando o servidor recebe uma lista de alterações feitas localmente por um usuário, ele tentará executar as operações a partir da versão atual do documento do servidor, caso a operação possa ser executada, o servidor irá executá-la, irá atualizar o documento no servidor e enviar as alterações por *broadcast* para os outros usuários, por outro lado, caso a operação não possa ser executada o servidor irá descartar a operação.

7.4. Motivação: estudo de caso no sistema ESPIM

O MoSCoR foi proposto para estender a ferramenta de autoria subjacente ao modelo ESPIM (*Experience Sampling and Programmed Intervention Method*) com recursos de colaboração síncrona (Dantas, 2024; Dantas et al., 2023).

7.4.1. Modelo e Sistema ESPIM

O Método *Experience Sampling and Programmed Intervention Method* (ESPIM) estende o método *Experience Sampling Method* (ESM) que tem origem na área de Psicologia (Larson and Csikszentmihalyi, 1978).

7.4.1.1. *Experience Sampling Method*

O Método de Amostragem de Experiências (*Experience Sampling Method* - ESM) foi inicialmente utilizado para investigar a experiência do tempo sozinho em adolescentes (Larson and Csikszentmihalyi, 1978). Neste método, alarmes eletrônicos distribuídos aleatoriamente orientam os participantes a registrarem seus pensamentos e sentimentos imediatos, reduzindo o viés de memória e possibilitando a coleta sistemática de autorrelatos em ambientes naturais. Além disso, o ESM envolve conceitos de coleta de dados em contextos reais e de intervenção, já que a simples programação de um alarme constitui uma interrupção no cotidiano dos usuários.

Evoluindo a partir de diários tradicionais, o ESM foi proposto e utilizado na criação de um arquivo detalhado das experiências diárias dos indivíduos através de autorrelatos sistemáticos realizados em momentos aleatórios durante a rotina normal (Larson and Csikszentmihalyi, 2014).

A literatura evidencia uma ampla variedade de plataformas que permitem a pesquisadores implementar programas de intervenção para monitorar usuários-alvo e coletar dados de experiências no ambientes natural desses usuários. A partir da análise de 239 sistemas, Henry et al. (2024) identificaram onze atributos essenciais a serem considerados na seleção de uma plataforma: localização geográfica, suporte dos desenvolvedores aos usuários da plataforma, tipos de experiências oferecidas, métodos de registro de usuários, tipos de mídia e questões, taxas de amostragem, recursos de visualização e monitoramento de dados, segurança e privacidade, custo, e continuidade no desenvolvimento da plataforma. Desse estudo, é possível identificar uma lacuna significativa: apenas cinco das 239 plataformas avaliadas oferecem aos pesquisadores a opção de criarem programas de intervenção de forma independente, i.e, sem depender de desenvolvedores. Esse é um dos requisitos fundamentais que guiaram a criação do método e do sistema ESPIM, apresentado a seguir, e que não foi considerado no estudo de Henry et al. (2024).

7.4.1.2. *Experience Sampling and Programmed Intervention Method*

Originado do protótipo SmartESM (Pimentel et al., 2016), o modelo e sistema ESPIM, abreviatura de *Experience Sampling and Programmed Intervention Method*, combina princípios do ESM e Computação Ubíqua para permitir a coleta programada de autorrela-

tos em ambientes natural por *observadores*, assim chamados por sua função de observar remotamente seus usuários-alvo (Cunha, 2019; Cunha et al., 2021).

Utilizando a plataforma web do ESPIM, um observador cria programas de intervenção que podem conter e coletar uma variedade de mídias por meio de perguntas ou orientações. Os programas são chamados de *programas de intervenção* considerando que uma intervenção corresponde ao ato de intervir, i.e., de exercer influência em determinada situação na tentativa de alterar o seu resultado.

Observadores do ESPIM são profissionais das áreas de saúde e educação, e os usuários-alvo são, respectivamente, pacientes e alunos. Os programas são apresentados aos usuários-alvo em horários pré-definidos pelos observadores por meio de uma aplicação Android para *smartphones*. Assim, o ESPIM permite desde a obtenção de dados detalhados sobre as experiências diárias das pessoas até a implementação de intervenções planejadas para influenciar o comportamento dos usuários-alvo (Cunha et al., 2021)

O ESPIM tem sido utilizado para desenvolver e aplicar programas de intervenção por meio de sua plataforma associada, que combina um software Web para o desenvolvimento de intervenções com um aplicativo para a aplicação remota desses programas (Cachioni et al., 2019; Cliquet et al., 2023, 2021; Flauzino et al., 2020; Rodrigues et al., 2021; Sanches et al., 2022; Zaine et al., 2019b,a).

7.4.2. MoSCor aplicado ao ESPIM

Para criar e dar suporte à comunicação bidirecional no ESPIM, foi utilizado o protocolo *WebSocket*, que é uma solução comum e amplamente utilizados em editores colaborativos. Também foi implementada no servidor o Gerenciamento de Canais de *WebSocket* (GCWS), que é uma rotina que tem por função criar e gerenciar o uso dos canais para permitir a troca de mensagens bidirecional entre os *frontends* e o *backend*.

Além disso, para suporte às regras voltadas à troca de mensagens por meio dos canais, foi implementado no servidor e no cliente o Gerador e Interpretador de Mensagens dos Canais de *WebSocket* que é uma rotina que tem por função criar as mensagens que serão enviadas no canal bidirecional, além de interpretar as mensagens recebidas.

Para o controle de histórico, de versões e sincronização de documentos, o modelo de dados do ESPIM foi modificado para incluir as classes do MoSCoR (Figura 7.2) resultando no modelo estendido ilustrado na Figura 7.3. Além disso, em todos os componentes foram implementadas rotinas para enviar e receber mensagens por meio de canais bidirecionais. Para cada componente, foi criado um canal e regras para esses canais.

Para o controle de alterações, foi desenvolvida uma solução baseada nos algoritmos de controle de concorrência Jupiter (Nichols et al., 1995), que emprega *Operational Transformation* (OT) e SOCT4 (Bouazza et al., 2000). Para o controle de concorrência e mesclagem, foi implementada uma solução baseada em operações delta, micro e macro versões (Kuryazov and Winter, 2014, 2015; Kuryazov D.A., 2016; Kuryazov et al., 2018).

Com base nos princípios e restrições para Edição Colaborativa em Tempo Real adotados no MoSCoR (Quatro 2), o conjunto original de requisitos do modelo e sistema ESPIM foi estendido com oito requisitos de edição colaborativa síncrona (Quadro 3).

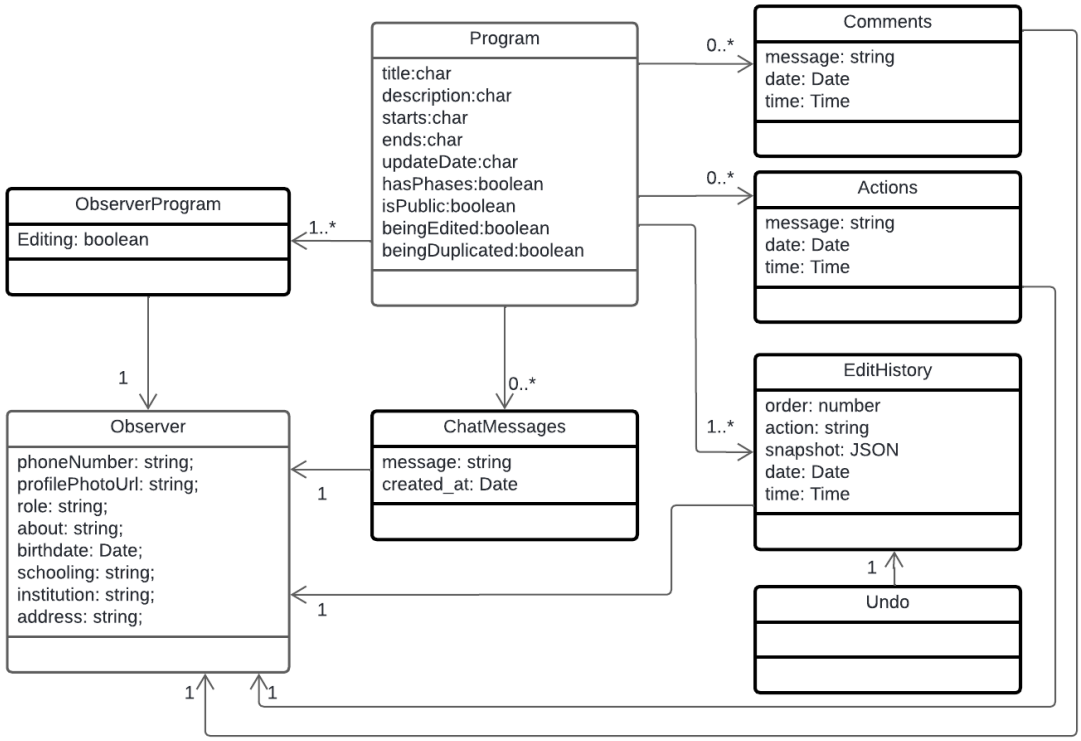

Figura 7.3: Modelo de classes da ferramenta de autoria ESPIM estendido com o MoSCoR

Quadro 3: Extensão dos requisitos funcionais (RF) do ESPIM para oferta de recursos de edição colaborativa

RF	**Descrição**
RF-RTCE1	Permitir que vários especialistas **editem** um mesmo programa ao mesmo tempo
RF-RTCE2	Manter um **registro** atualizado de todas as atualizações ocorridas em um programa
RF-RTCE3	Fazer que com que as alterações realizadas por um especialista sejam percebidas em tempo real por todos os especialistas que estão editando o mesmo programa
RF-RTCE4	Possibilitar que as alterações efetuadas em um programa possam ser **desfeitas/refeitas**
RF-RTCE5	Permitir aos especialistas **saberem** se outros especialistas estão editando o mesmo programa
RF-RTCE6	Permitir que os especialistas que estão editando um mesmo programa possam trocar **mensagens** pelo sistema
RF-RTCE7	Permitir que os especialistas possar registrar **comentários** sobre os componentes do programa
RF-RTCE8	Permitir **identificar** o autor de cada alteração em um programa

Exemplos das interface da versão ESPIM com autoria colaborativa estão ilustradas na Figura 7.4. As interfaces e as funcionalidades da ferramenta de autoria foram estendidas com ícones para:

- a edição síncrona por vários autores (RF-RTCE1, RF-RTCE8, Fig. 7.4A);

- troca mensagens *chat* (RF-RTCE2, Fig. 7.4B);
- a inclusão de comentários (RF-RTCE3, Fig. 7.4C);
- acesso ao histórico de edições, que permite visualizar todas as alterações do programa e quem as realizou, bem como desfazer alterações (RF-RTCE4 a RF-RTCE6, Fig. 7.4D).

As funcionalidades de edição colaborativa síncrona (requisitos RF-RTCE1, RF-RTCE7 e RF-RTCE8), não perceptíveis nas figuras das interfaces mas perceptíveis durante a interação dos usuários, foram implementados com base nos componentes e nos canais de comunicação independente que cada componente possui.

Figura 7.4: Interfaces de autoria da plataforma ESPIM com recursos de colaboração síncrona: uma ocorrência indicação de **A** (autores online), **B** (*chat*), **C** (comentários) e **D** (histórico de edição) em cada tela.

Como exemplo didático do uso do MoSCoR, a próxima seção apresenta a adaptação de uma aplicação de cadastro de produto para permitir autoria síncrona.

7.5. Exemplo de uma aplicação *REST* que envia mensagem por *broadcast*

Nesta seção apresentamos a implementação de uma aplicação REST de cadastro de produto que permitir o cadastro de produtos de modo síncrono por múltiplos usuários. O código está disponibilizado do Git.[2]

7.5.1. Aplicação: cadastro de produto

A implementação utiliza um *WebSocket* para estabelecer um canal de *broadcast*. O canal tem como objetivo notificar todos os usuários sempre que um novo produto for inserido no sistema. O *backend* é desenvolvido com o framework *Spring Boot*, enquanto o *frontend* é implementado utilizando o framework *Angular*.

[2] `https://github.com/augustodantas/TADS-Angular-2024`

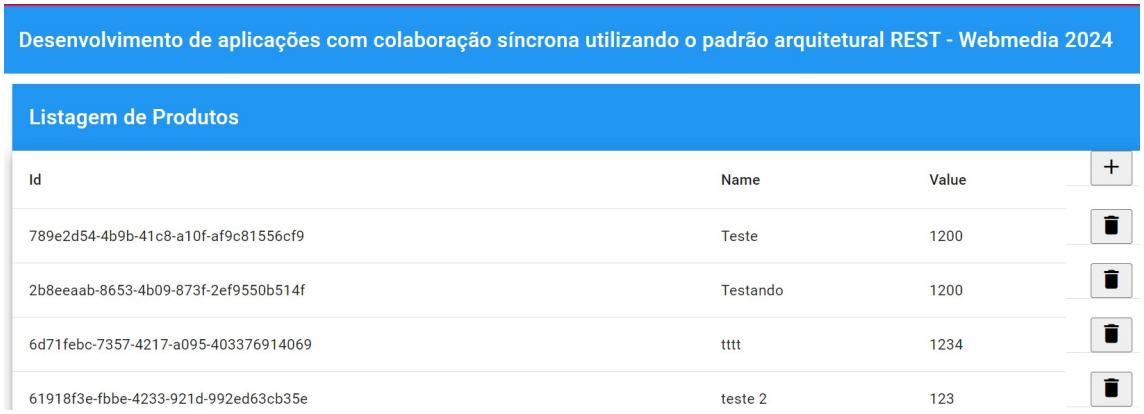

Figura 7.5: Tela que apresenta a lista de produtos cadastrados

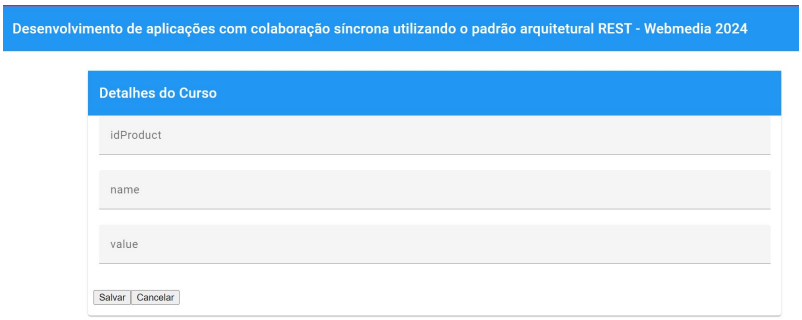

Figura 7.6: Cadastro app: Formulário de cadastro de um novo produto

Ao acessar a aplicação, o usuário visualiza a lista de produtos cadastrados no banco de dados, conforme ilustrado na Figura 7.5. Para cadastrar um novo produto, o usuário deve clicar no botão com o símbolo de adição localizado no canto superior direito da tela de listagem. Ao clicar, é exibido o formulário de cadastro, conforme a Figura 7.6.

7.5.2. Aplicação REST: com e sem *broadcast*

Em uma aplicação baseada no padrão REST, o comportamento típico é ilustrado na Figura 7.7. Mesmo com múltiplos usuários conectados, as ações de um usuário não são comunicadas aos demais.

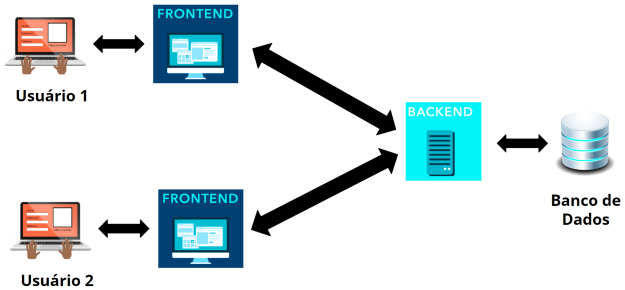

Figura 7.7: Sistema baseado em REST sem o envio de mensagem por *broadcast*

Para que uma aplicação REST suporte edição colaborativa síncrona, é necessário

implementar rotinas que permitam o envio das ações de um usuário para os demais, sincronizando assim os documentos editados. A Figura 7.8 ilustra esse processo: sempre que o *backend* recebe uma alteração, ela é comunicada aos outros usuários por meio de *broadcast*.

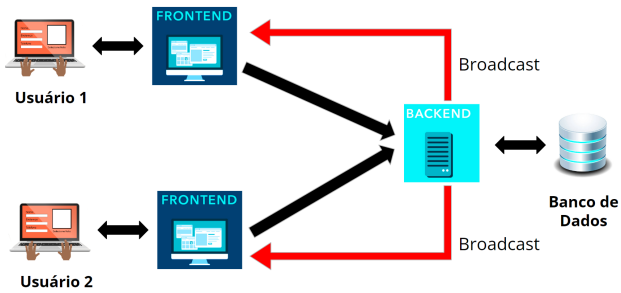

Figura 7.8: Sistema baseado em REST com envio de mensagem por *broadcast*

A aplicação descrita nesta seção seguirá o comportamento mostrado na Figura 7.8. Para implementar essa funcionalidade, será criado um canal de *broadcast* utilizando *WebSocket*. Sempre que um novo produto for inserido, todos os usuários conectados serão notificados e a lista de produtos será atualizada automaticamente.

7.5.3. Desenvolvimento do *backend Spring Boot*

O *Java Spring Framework* é um framework de software livre amplamente utilizado em aplicações empresariais para criar aplicativos de nível de produção que operam na Java Virtual Machine (JVM). A ferramenta Java *Spring Boot* complementa o *Spring Framework* ao simplificar e acelerar o desenvolvimento de aplicações web e microsserviços, oferecendo recursos como autoconfiguração, uma abordagem opinativa à configuração (isto é, que prioriza convenções pré-definidas sobre configurações personalizadas), e a capacidade de criar aplicativos independentes. Esses recursos combinados permitem configurar um aplicativo Spring com mínima instalação e configuração, tornando o processo mais ágil e eficiente (IBM, 2024).

7.5.3.1. Iniciando um projeto em *Spring Boot*

Para iniciar projeto, é necessário acessar a ferramenta *spring initializr*: uma ferramenta online, com uma interface integrada, que facilita a criação inicial de projetos *Spring Boot*.

Na interface da ferramenta *spring initializr* apresentada na Figura 7.9, deve-se preencher algumas informações do projeto e gerar o pacote que será importado para o ambiente de desenvolvimento. As informações devem ser preenchidas da seguinte forma:

- **Project**: Nessa seção é definido o gerenciador de dependências. No projeto será utilizado o Maven.
- **Language**: Nessa seção é definida a linguagem que será utilizada no projeto. Será utilizada a linguagem Java.
- **Project Metadata**: Seção onde são configuradas algumas informações do projeto:

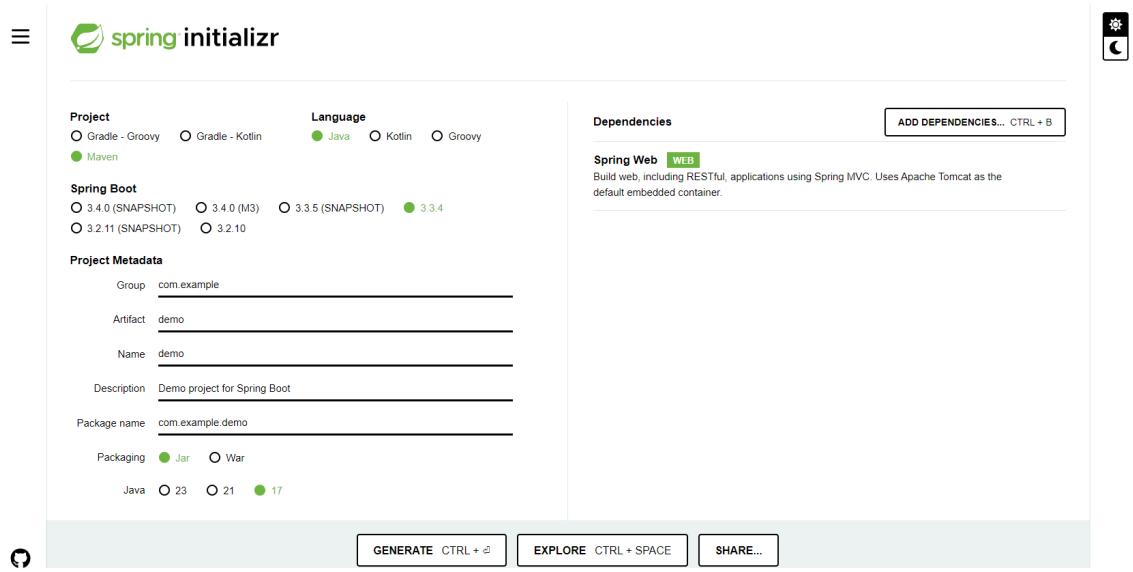

Figura 7.9: Interface do *spring initializr* utilizado para criar um projeto *Spring Boot* com as características básicas

- **Group**: Nome do domínio invertido, o que pode ser o domínio de uma empresa ou projeto o qual esteja trabalhando. Manter *com.example*
- **Artifact**:Nome do projeto. Manter **demo**
- **Name**: Será preenchido automaticamente com o nome do projeto.
- **Description**: Campo para adicionar alguma descrição do projeto.
- **Package name**: Aqui será configurado o pacote do projeto. Nesse caso o domínio + o nome do projeto.
- **Packaging**: Tipo de empacotamento do projeto. Manter *jar*
- **Java**: Versão do java que será utilizada no projeto. Utilizar a versão dezessete.

Após preencher as informações do projeto, é necessário adicionar as dependências que serão utilizadas. Existem diversas dependências que podem ser adicionadas, e variam conforme o projeto. Na seção *Dependencies* do lado direito da ferramenta, clicar no botão ADD DEPENDENCIES e adicionar a dependência *Spring Web*, que está na seção WEB.

Após finalizar o preenchimento de todos os campos, deve-se clicar no botão *GENERATE*, na parte inferior da tela, o que gera um arquivo *.zip* correspondente à configuração especificada para o projeto. Neste ponto, é necessário fazer o *download* do arquivo *.zip* para a máquina local.

Realizado o *download*, o arqruivo (*.zip*) deve ser aberto no editor de preferência de quem está implementando o projeto. Este tutorial seguirá utilizando o *Visual Studio Code* com as extensões *Extension Pack for Java* e *Spring Boot Extension Pack*.

Na Figura 7.10 é possível observar os arquivos e pastas que compõem o projeto base: toda a estrutura do projeto foi detalhada pelo *spring initializr* a partir da configuração especificada. O projeto inicial já pode ser executado: o Spring Boot traz o servidor Tomcat embutido. Tomcat é um contêiner de servlets Java, ou seja, um ambiente de exe-

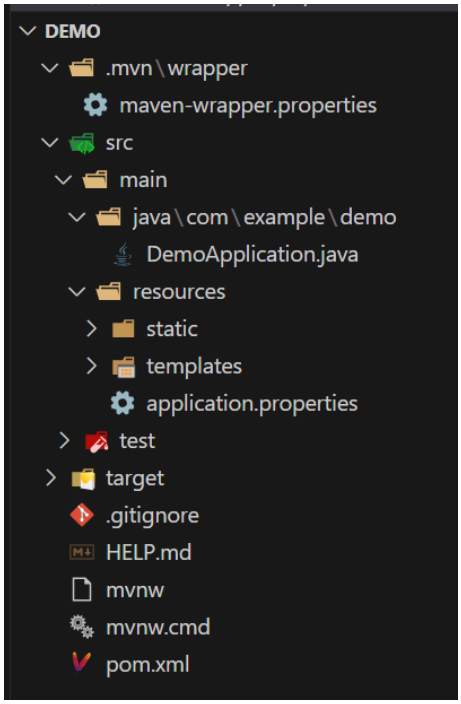

Figura 7.10: Pastas e arquivos iniciais do projeto Demo

cução que gerencia o ciclo de vida dos servlets, responsáveis por processar requisições e respostas em aplicações web. Tomcat implementa as especificações Java Servlet, JavaServer Pages (JSP) e WebSocket, permitindo a execução e o gerenciamento eficientes de aplicações web Java. A aplicação, por padrão, é inicializada na porta 8080.

O projeto irá utilizar o *Maven*, que é uma ferramenta de automação de *build* e de gerenciamento de dependências amplamente utilizada em projetos Java, incluindo aqueles que utilizam o *Spring Boot*. A principal do função *Maven* é facilitar o gerenciamento do ciclo de vida do projeto, desde a compilação do código até a geração de pacotes finais, além de garantir que todas as bibliotecas necessárias estejam disponíveis. .

Maven permite definir as bibliotecas e frameworks que o projeto necessita em um arquivo chamado `pom.xml` (*Project Object Model*). O *Maven* automaticamente faz o download das dependências correspondentes e as adiciona ao projeto, garantindo que todas as versões estejam compatíveis. Além disso, o *Maven* padroniza o processo de *build* do projeto, o que inclui compilar o código, executar testes, empacotar a aplicação e implantar o pacote gerado, facilitando a integração contínua e a entrega contínua (CI/CD).

7.5.3.2. Conectar o projeto ao Banco de dados PostgreSQL

O *Spring Boot* permite estabelecer uma conexão segura e apropriada com bancos de dados, facilitando a persistência e a recuperação de dados pela aplicação ("persistência", neste contexto, refere-se à capacidade da aplicação de armazenar dados de forma duradoura no banco de dados, garantindo que as informações permaneçam acessíveis mesmo após o término da execução do programa)

Para conectar a aplicação ao banco de dados, utilizamos a *JPA* (Java Persistence API) com o *Hibernate*, que gerencia o mapeamento objeto-relacional e simplifica as operações de persistência, assegurando integridade e eficiência no gerenciamento dos dados.

No projeto **demo** foi definido o *Maven* como ferramenta de gerenciamento de dependências. Desta forma é possível definir as dependências por meio do arquivo *pom.xml*. O *Maven* garante que as bibliotecas, necessárias às dependências definidas, estejam disponíveis durante a compilação e execução da aplicação.

As dependências do JPA e do banco de dados PostgreSQL devem ser declaradas no arquivo *pom.xml*, conforme apresentado na Figura 7.11.

pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```

Figura 7.11: Dependências declaradas no arquivo *pom.xml* para conectar o banco PostgreSQL via JPA

7.5.4. Definindo o Model

No contexto de um projeto *Spring Boot*, o **model** (modelo) representa a camada responsável por encapsular a lógica de negócios e os dados do domínio da aplicação. No *Spring Boot*, os modelos geralmente são classes Java simples (*Plain Old Java Objects* - POJOs) que representam entidades mapeadas para tabelas do banco de dados, utilizando anotações como `@Entity`, `@Table`, e `@Id`, fornecidas pela JPA. Além de modelar as entidades, essa camada pode conter validações, regras de negócio, e outras funcionalidades diretamente relacionadas aos dados.

O **model** é essencial para garantir a integridade dos dados e facilitar a interação com o banco de dados, além de servir como base para a comunicação entre as diferentes camadas da aplicação.

Para definir o modelo de dados, deve ser criada uma classe Java com os atributos do objeto. Além disso devem ser criados os métodos *setters* e *getters* da classe. A Figura 7.12, apresenta a classe model do projeto com os respectivos atributos, métodos e *annotations*.

7.5.5. Criando o `Repository`

Para possibilitar o acesso ao banco de dados, conforme detalhado na Figura 7.13 devem ser criadas a pasta e a classe `ProductRepository.Java`. Em um projeto *Spring*

Figura 7.12: Modelo da classe `Product` que será armazenada no banco de dados

Boot, uma classe `Repository` é uma interface que serve como intermediária entre a aplicação e a camada de persistência, que geralmente é um banco de dados.

A classe `Repository` abstrai a lógica de acesso aos dados, oferecendo uma interface simplificada e organizada para executar operações de leitura e escrita no banco de dados. O *Spring Data JPA*, uma extensão do *Spring*, simplifica a criação de repositórios baseados em JPA sem a necessidade de escrever consultas SQL manualmente.

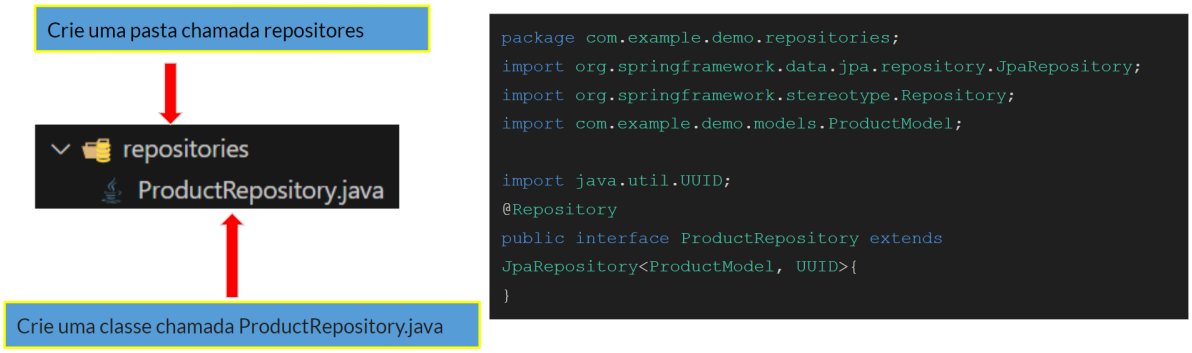

Figura 7.13: Pasta e classe `ProductRepository`

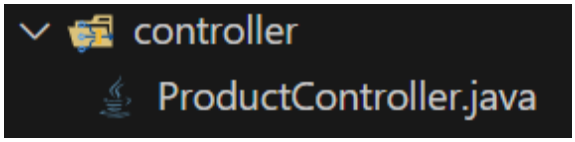

Figura 7.14: Criação de uma pasta `controller` e da classe `ProductController`

7.5.5.1. Definindo o `controller` da classe `Product`

Devem ser criadas uma pasta `controller` e uma classe `ProductController.java`, conforme apresentado na Figura 7.14. Um *controller* é uma classe responsável por gerenciar as solicitações HTTP recebidas pelo aplicativo e determinar as respostas apropriadas.

O *controller* faz parte da camada de apresentação da aplicação, seguindo o padrão *MVC* (*Model-View-Controller*). Nesse padrão, o *Model* representa os dados ou a lógica de negócios, a *View* define com esses dados são apresentados ao usuário, e o *Controller* recebe as entradas do usuário, processa a lógica necessária e retorna uma resposta correspondente.

Na classe *controller* são definidos os *endpoints* REST. Na aplicação serão definidos os *endpoints* para listar, inserir, excluir e atualizar os dados do banco de dados.

Na Figura 7.15 é apresentada a definição da classe e as injeções de dependência. Injeções de dependência são um padrão de design que visa reduzir o acoplamento entre os componentes da aplicação. Nesse caso, o `ProductController` depende de `ProductRepository` para acessar os dados dos produtos no banco de dados. Em vez de o `ProductController` criar uma instância de `ProductRepository` diretamente, o *Spring Boot* injeta essa dependência automaticamente utilizando a anotação `@Autowired`.

Na Figura 7.16 são apresentados os métodos para recuperar um registro do banco de dados e para listar dos registros (*GetOne* e *GetAll*).

O método *Post* para inserir registros no banco de dados é apresentado na Figura 7.17. O comando *service.execute()* é utilizado para notificar o canal de *WebSocket* sobre a realização de uma nova inserção, garantindo que os clientes conectados recebam atualizações em tempo real.

Para finalizar, a classe *controller* e os verbos HTTP padronizados no REST são apresentados nas Figuras 7.18 e 7.19. A Figura 7.18 apresenta o método *Delete* para apagar um registro do banco de dados, enquanto a Figura 7.19 descreve o método *Update* que realiza a alteração dos dados.

7.5.5.2. Criar um *WebSocketMessageBroker*

No contexto do *Spring Boot*, um *WebSocketMessageBroker* é um componente que facilita a comunicação em tempo real entre clientes e servidores usando o protocolo WebSocket. Ele atua como um intermediário que gerencia a troca de mensagens entre diferentes partes de uma aplicação, permitindo a comunicação bidirecional e assíncrona entre o cliente

```java
@RestController
public class ProductController {

    @Autowired
    ProductRepository productRepository;

    @Autowired
    ProcessorService service;

    public record ProductRecordDto(@NotBlank String name, @NotNull BigDecimal value) {
    }
```

Figura 7.15: Definição da classe e injeção das dependências na classe *ProductController*

```java
@GetMapping("/api/products")
public ResponseEntity<List<ProductModel>> getAllProducts() {
    return ResponseEntity.status(HttpStatus.OK)
            .body(productRepository.findAll());
}

/**
 * @param id
 * @return
 */
@GetMapping("/api/products/{id}")
public ResponseEntity<Object> getOneProduct(@PathVariable(value = "id") UUID id) {
    Optional<ProductModel> productO = productRepository.findById(id);
    if (productO.isEmpty()) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
                .body("Product not found.");
    }
    return ResponseEntity.status(HttpStatus.OK)
            .body(productO.get());
}
```

Figura 7.16: Métodos *GetOne* e *GetAll* da classe *ProductController*, utilizados para recuperar um registro e a lista de registro do banco de dados

```java
@PostMapping("/api/products")
public ResponseEntity<ProductModel> saveProduct(@RequestBody
    @Valid ProductRecordDto productRecordDto) {

    var productModel = new ProductModel();
    BeanUtils.copyProperties(productRecordDto,productModel);
    service.execute();
    return ResponseEntity.status(HttpStatus.CREATED)
            .body(productRepository.save(productModel));
}
```

Figura 7.17: Método que controla o *Post* da classe *ProductController*

```java
@DeleteMapping("/api/products/{id}")
public ResponseEntity<Object> deleteProduct(@PathVariable(value = "id") UUID id) {
    Optional<ProductModel> productO = productRepository.findById(id);
    if (productO.isEmpty()) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
                .body("Product not found.");
    }
    productRepository.delete(productO.get());
    return ResponseEntity.status(HttpStatus.OK)
            .body("Product deleted successfully.");
}
```

Figura 7.18: Método que controla o *Delete* da classe *ProductController*

```java
@PutMapping("/api/products/{id}")
public ResponseEntity<Object> updateProduct(@PathVariable(value = "id") UUID id,
        @RequestBody @Valid ProductRecordDto productRecordDto) {
    Optional<ProductModel> productO = productRepository.findById(id);
    if (productO.isEmpty()) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
                .body("Product not found.");
    }
    var productModel = productO.get();
    BeanUtils.copyProperties(productRecordDto,
            productModel);
    return ResponseEntity.status(HttpStatus.OK)
            .body(productRepository.save(productModel));
}
```

Figura 7.19: Método que controla o *Update* da classe *ProductController*

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-websocket</artifactId>
</dependency>
```

Figura 7.20: Instalação da dependência do WebSocket no arquivo pom.xml

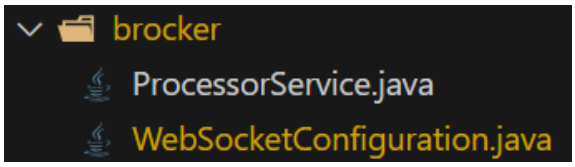

Figura 7.21: Criação da pasta broker que irá centralizar as classes responsáveis por gerenciar o WebSocket

(como um navegador) e o servidor. O *WebSocketMessageBroker* permite que o servidor envie dados para o cliente de forma ativa e em tempo real, ou seja, sem que o cliente precise solicitar explicitamente as informações. Além disso, o *broker* gerencia o roteamento das mensagens baseadas nos destinos (ou canais) definidos, e os clientes podem se inscrever em tópicos específicos para receber mensagens relevantes, utilizando o protocolo STOMP (*Simple Text Oriented Messaging Protocol*) sobre WebSocket para definir o formato e as regras de comunicação.

No *Spring Boot*, a configuração de um *WebSocketMessageBroker* geralmente envolve a habilitação de um *broker* simples, que permite o roteamento de mensagens para destinos como `/topic` (para mensagens de *broadcast*) e `/queue` (para mensagens direcionadas). Por exemplo, um *endpoint* STOMP pode ser registrado usando a anotação `@EnableWebSocketMessageBroker`, permitindo que os clientes se conectem ao servidor utilizando *WebSockets*. Esse *endpoint* pode ser configurado com um *fallback* para *SockJS*, que atua como um mecanismo alternativo de transporte, garantindo suporte em navegadores que não têm *WebSocket* nativo ou em ambientes onde as conexões WebSocket são restritas. Dessa forma, a comunicação em tempo real permanece funcional, assegurando uma experiência consistente para todos os usuários.

O uso de prefixos de destino, como `/app`, ajuda a diferenciar as mensagens que precisam ser processadas pelo servidor. Em resumo, o *WebSocketMessageBroker* no *Spring Boot* é uma ferramenta poderosa para construir aplicações que necessitam de comunicação em tempo real, suportando a atualização instantânea e bidirecional de dados entre o servidor e os clientes.

Para implementar o WebSocket no projeto deste tutorial, o primeiro passo é instalar as dependências no arquivo `pom.xml`, conforme está apresentado na Figura 7.20.

7.5.6. Desenvolvimento do *frontend Angular*

Para criar um projeto *Angular*, é necessário ter instalado na máquina o **Node.js**. Para instalar o *Angular* CLI é necessário executar o comando:

npm install g angular/cli

Após a instalação do *Angular* Cli, para criar um novo projeto é necessário executar o seguinte comando:

ng new nome-do-projeto

Deve-se substituir nome-do-projeto pelo nome que se deseja dar ao projeto. Um projeto *Angular* é composto por várias pastas e arquivos, conforme apresentado na Figura 7.22. O conteúdo inicial de um projeto *Angular* é como segue:

- **e2e/**: Esta pasta contém o código de testes end-to-end (e2e) para a aplicação. Esses testes são usados para simular interações do usuário e garantir que a aplicação funcione corretamente de ponta a ponta.
- **node_modules/**: Esta pasta é gerada automaticamente e contém todas as dependências do projeto *Angular* instaladas via *npm*. Essas dependências incluem bibliotecas e pacotes necessários para a execução e desenvolvimento da aplicação.
- **src/**: Esta é a pasta principal onde o código-fonte da aplicação é armazenado. Ela contém subpastas e arquivos que formam a estrutura básica da aplicação.
  - **app/**: Contém os módulos, componentes, serviços e outros arquivos relacionados à lógica da aplicação. O arquivo *app.module.ts* é o módulo raiz da aplicação.
  - **assets/**: Contém arquivos estáticos como imagens, fontes, e outros recursos que não são processados pelo *Angular*.
  - **environments/**: Contém arquivos de configuração para diferentes ambientes (por exemplo, desenvolvimento e produção). Permite que a aplicação utilize configurações diferentes dependendo do ambiente em que está sendo executada.
  - **index.html**: O arquivo HTML principal da aplicação. É o ponto de entrada do aplicativo e contém a estrutura básica da página.
  - **main.ts**: O ponto de entrada principal da aplicação *Angular*. Este arquivo é responsável por inicializar o aplicativo e carregá-lo no navegador.
  - **polyfills.ts**: Este arquivo contém scripts que fornecem suporte para navegadores mais antigos, garantindo que a aplicação *Angular* funcione corretamente em diferentes ambientes.
  - **styles.css**: O arquivo de estilos global da aplicação. Aqui podem ser definidos estilos que serão aplicados em toda a aplicação.
  - **angular.json**: Arquivo de configuração principal do projeto *Angular*. Ele define como o aplicativo deve ser construído, os caminhos para os arquivos e outras configurações importantes.
  - **package.json**: Contém as informações do projeto, como nome, versão, scripts disponíveis, e as dependências do projeto. Também é usado pelo *npm* para instalar pacotes e bibliotecas.
  - **tsconfig.json**: Arquivo de configuração do TypeScript, especificando opções de compilação e comportamentos relacionados ao TypeScript.

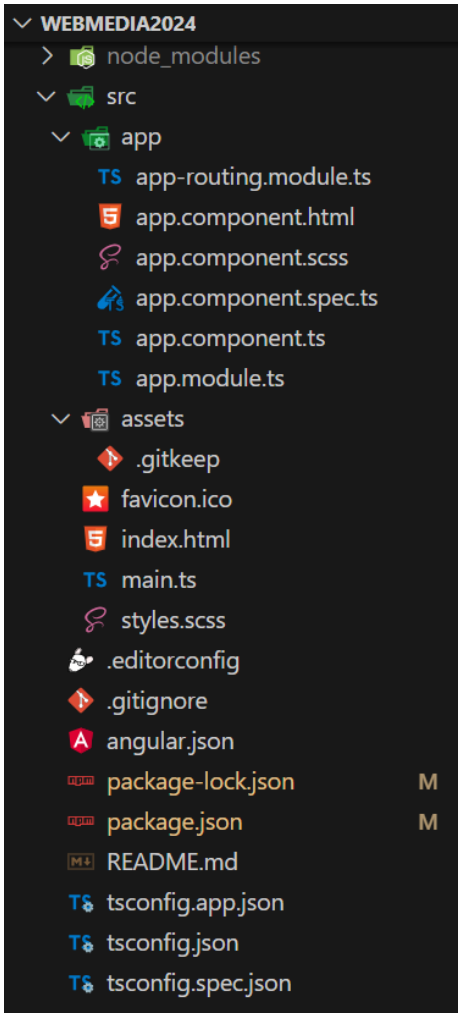

Figura 7.22: Arquivos e pastas do projeto *Angular*

– **tslint.json**: Arquivo de configuração para o *TSLint*, que é uma ferramenta de análise estática usada para verificar a qualidade e conformidade do código TypeScript.

No projeto deste tutorial utilizamos *Angular Material*, uma biblioteca de componentes de interface de usuário (UI) projetada para o framework *Angular*, que adere às diretrizes do *Material Design* do Google. A biblioteca oferece uma ampla gama de componentes prontos para uso, como botões, formulários, ícones, tabelas, que facilitam a criação de interfaces de usuário modernas, responsivas e esteticamente agradáveis. Para a instalação do *Angular Material* deve ser utilizado o seguinte comando:

ng add angular/material

A biblioteca *Angular Material* foi desenvolvida para se integrar perfeitamente em aplicações *Angular*, garantindo que os componentes mantenham consistência e sigam as melhores práticas de usabilidade. A Figura 7.23 apresenta o resultado da instalação do *Angular Material*.

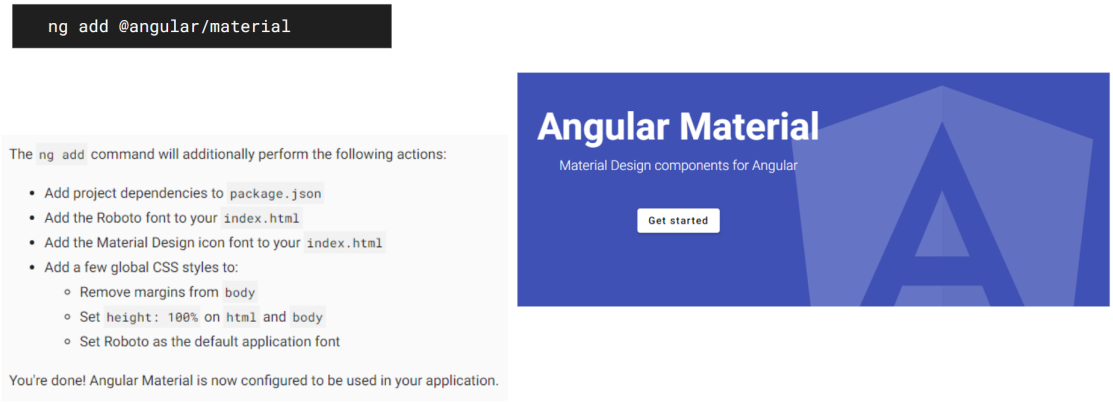

Figura 7.23: Resultado da instalação do *Angular Material* no projeto

A Figura 7.24 detalha o código que deve ser inserido no arquivo *app.module.ts*: estão especificados todos os *imports* necessário ao funcionamento do projeto.

O código HTML do arquivo *app.component.html* é apresentado na Figura 7.25. Esse conteúdo especifica apenas uma barra de ferramentas com o título do tutoriaa e o elemento <*router-outlet*>. O `<router-outlet>` é uma diretiva fundamental no *Angular* que atua como um espaço reservado (placeholder) para a renderização de componentes com base na configuração de rotas da aplicação. Ao utilizar o sistema de roteamento do *Angular*, o `<router-outlet>` permite que diferentes componentes sejam carregados dinamicamente em resposta a mudanças na URL, facilitando a navegação entre diferentes partes da aplicação sem a necessidade de recarregar a página.

7.5.6.1. Criação do módulo *products*

Para estruturar o projeto a ser desenvolvido, será criado um módulo chamado *products* a partir do seguinte comando:

ng g m products –routing

Os módulos em *Angular* ajudam a organizar e estruturar uma aplicação. Um módulo é uma coleção de componentes, diretivas, pipes e serviços que são agrupados para fornecer funcionalidades específicas.

Também devem ser criados os componentes para produto, um formulário para edição do produto e um arquivo de serviços para produtos, além do modelo de dados para o objeto produto. Devem ser executados os seguintes comandos:

ng g c products/products

ng g i products/models/product

ng g s products/service/products

ng g c products/product-form

```typescript
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { TestaBidingComponent } from './testa-biding/testa-biding.component';
import { FormsModule } from '@angular/forms';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import {MatToolbarModule} from '@angular/material/toolbar';
import { HttpClientModule } from '@angular/common/http';


@NgModule({
  declarations: [
    AppComponent,
    TestaBidingComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    BrowserAnimationsModule,
    MatToolbarModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Figura 7.24: Código do *app.module.ts*

```html
<mat-toolbar color="primary">
    <span>Desenvolvimento de aplicações com colaboração síncrona utilizando o
        padrão arquitetural REST - Webmedia 2024</span>
</mat-toolbar>
<router-outlet></router-outlet>
```

Figura 7.25: Código HTML do *app.component.html*

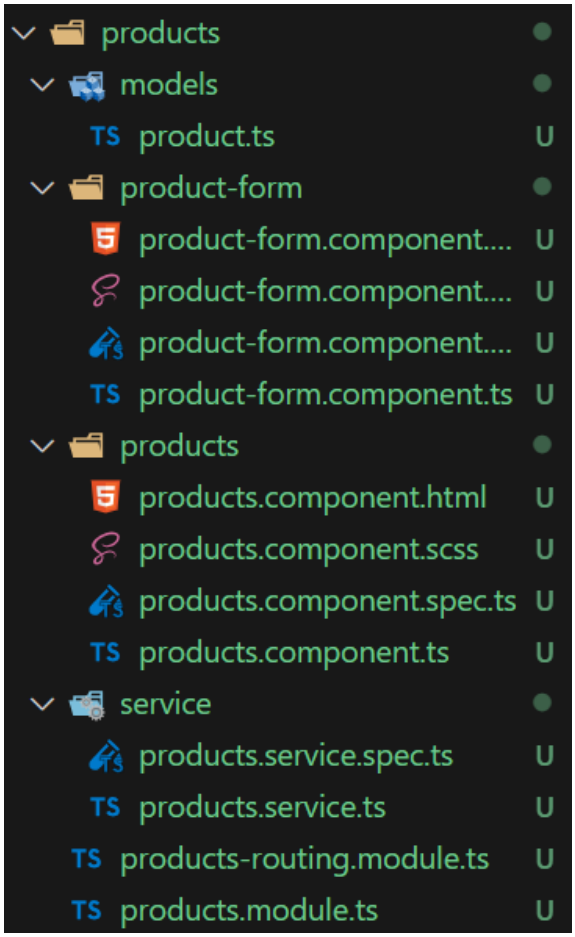

Figura 7.26: Pastas e arquivos do Módulo *products* após a criação de todos os serviços, classes, modelos e componentes

Após a execução de todos os comandos, o módulo *products* deve estar semelhante ao que é apresentado na Figura 7.26.

Em *Angular*, **service** é uma classe que encapsula a lógica de negócios e fornece funcionalidades que podem ser compartilhadas entre diferentes componentes da aplicação. *Services* são usados para realizar tarefas como manipulação de dados, comunicação com APIs externas, gerenciamento de estados e execução de operações que não estão diretamente relacionadas à apresentação da interface do usuário. Ao utilizar a injeção de dependência, o *Angular* permite que os componentes acessem os *services* de forma eficiente, promovendo a reutilização de código e a separação de responsabilidades.

Após a criação do módulo *products*, é necessário ajustar as rotas para que o *Angular* consiga executá-lo. O arquivo `app-routing.module.ts` define as rotas da aplicação, associando caminhos de URL a componentes específicos que devem ser carregados quando essas rotas são acessadas. A Figura 7.27 especifica o arquivo *app.routing.ts*.

Também é necessário ajustar as rotas do *products-routing* conforme a Figura 7.28.

O arquivo *products.module* deve ser preenchido conforme o que está descrito na Figura 7.29. A Figura 7.30 especifica o conteúdo do arquivo *products.component.html*.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  {path: '',pathMatch: 'full', redirectTo : 'products'},
  {path: 'products',
  loadChildren: () => import ('./products/products.module').then(m => m.ProductsModule)}
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Figura 7.27: Conteúdo do arquivo *app-routing* para gerenciar as rotas do projeto.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { ProductsComponent } from './products/products.component';
import { ProductFormComponent } from './product-form/product-form.component';

const routes: Routes = [
  {path: '', component: ProductsComponent},
  {path: 'new', component: ProductFormComponent}

];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class ProductsRoutingModule { }
```

Figura 7.28: Conteúdo do arquivo *products-routing* para gerenciar as rotas no módulo products.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import {MatTableModule} from '@angular/material/table';
import {MatCardModule} from '@angular/material/card';
import {MatToolbarModule} from '@angular/material/toolbar';
import {MatIconModule} from '@angular/material/icon';
import {MatFormFieldModule} from '@angular/material/form-field';
import {MatInputModule} from '@angular/material/input';
import { ProductsRoutingModule } from './products-routing.module';
import { ProductsComponent } from './products/products.component';
import { ProductFormComponent } from './product-form/product-form.component';
import { ReactiveFormsModule } from '@angular/forms';
@NgModule({
  declarations: [
    ProductsComponent,
    ProductFormComponent
  ],
  imports: [
    CommonModule,
    ProductsRoutingModule,
    MatTableModule,
    MatCardModule,
    MatToolbarModule,
    MatIconModule,
    ReactiveFormsModule,
    MatFormFieldModule,
    MatInputModule
  ]
})
export class ProductsModule { }
```

Figura 7.29: Conteúdo do *products.module.ts*

```
<mat-card>
    <mat-toolbar color="primary">
        Detalhes do Curso
    </mat-toolbar>
    <mat-card-content>
        <form [formGroup]="form">
            <mat-form-field  class="full-width">
                <input matInput placeholder="idProduct" formControlName="idProduct">
            </mat-form-field>

            <mat-form-field  class="full-width">
                <input matInput placeholder="name" formControlName="name">
            </mat-form-field>

            <mat-form-field  class="full-width">
                <input matInput placeholder="value" formControlName="value">
            </mat-form-field>
        </form>
    </mat-card-content>
    <mat-card-actions>
        <button mat-button (click)="onSubmit()">Salvar</button>
        <button mat-button (click)="onCancel()">Cancelar</button>
    </mat-card-actions>
</mat-card>
```

Figura 7.30: Conteúdo do *products.component.html*

7.5.6.2. Implementação do canal do WebSocket

Para que o *frontend* possa monitorar o canal de *broadcast* do *backend*, deve-se instalar as bibliotecas e implementar as rotinas. Para instalar o *SockJS* e o *StompJS* para *Angular*, devem ser executados os seguintes comandos:

npm i sockjs-client

npm i stomp/stompjs

Para que o *frontend* possa monitorar o *WebSocket* e gerenciar as mensagens recebidas, é necessário implementar o serviço `WebSocketConnector`. Esse serviço será responsável por gerenciar todas as comunicações realizadas por meio do WebSocket e poderá ser injetado em componentes e serviços que necessitam utilizar essa forma de comunicação. Dessa maneira, o `WebSocketConnector` centraliza a lógica de comunicação, facilitando a manutenção e a reutilização do código em diferentes partes da aplicação. A implementação do `WebSocketConnector` está descrita na Figura 7.31.

Após a definição do `WebSocketConnector`, toda a aplicação pode utilizar a comunicação via *WebSocket*. A Figura 7.32 apresenta o conteúdo final HTML do arquivo *product-form.component.html*. Já o conteúdo do *product-form.component.ts* é apresentado na Figura 7.33. O conteúdo do *products.service.ts* está listado na Figura 7.34 e a Figura 7.35 apresenta o conteúdo do *products.component.ts*.

```
import * as SockJS from 'sockjs-client';
import { Stomp } from "@stomp/stompjs";
export class WebSocketConnector {
    private stompClient: any;
    constructor(private webSocketEndPoint: string, private topic: string,
        private onMessage: Function, private callbackError?: Function) {
        const errorCallback = callbackError || this.onError;
        this.connect(errorCallback);
    }
    private connect(errorCallback: Function) {
        console.log("Starting a WebSocket connection");
        const ws = new SockJS(this.webSocketEndPoint);
        this.stompClient = Stomp.over(ws);
        this.stompClient.connect({}, (frame : any) => {
            this.stompClient.subscribe(this.topic, (event: any) => {
                this.onMessage(event);
            });
        }, errorCallback.bind(this));
    };
    private onError(error: any) {
        console.log("Error while connect: " + error);
        setTimeout(() => {
            console.log("Trying to connect again...");
            this.connect(this.onError);
        }, 3000);
    }
}
```

Figura 7.31: `WebSocketConnector`, service que centraliza a lógica de comunicação via WebSocket.

```
<mat-card>
    <mat-toolbar color="primary">
        Detalhes do Curso
    </mat-toolbar>
    <mat-card-content>
        <form [formGroup]="form">
            <mat-form-field  class="full-width">
                <input matInput placeholder="idProduct" formControlName="idProduct">
            </mat-form-field>

            <mat-form-field  class="full-width">
                <input matInput placeholder="name" formControlName="name">
            </mat-form-field>

            <mat-form-field  class="full-width">
                <input matInput placeholder="value" formControlName="value">
            </mat-form-field>
        </form>
    </mat-card-content>
    <mat-card-actions>
        <button mat-button (click)="onSubmit()">Salvar</button>
        <button mat-button (click)="onCancel()">Cancelar</button>
    </mat-card-actions>
</mat-card>
```

Figura 7.32: Implementação do *product-form.component.html*

7.6. Biblioteca alternativa: YJS

Várias bibliotecas têm sido disponibilizadas pela comunidade para apoiar a construção de aplicações colaborativas utilizando CRDT (*Convergent Replicated Data Type*).

O *Yjs*[3] é uma framework que permite a comunicação entre clientes por meio do compartilhamento de tipos de dados. *Yjs* emprega algoritmos CRDT para que as alterações feitas por um cliente sejam automaticamente distribuídas para os colaboradores e mescladas sem conflitos (Jahns, 2018b).

O *Yjs* adota uma arquitetura descentralizada segundo o modelo *Peer-to-Peer* (P2P), na qual cada participante atua simultaneamente como cliente e servidor. Isso significa que os participantes podem compartilhar recursos diretamente entre si sem a necessidade de um servidor centralizado para intermediar as comunicações ou o compartilhamento de dados. Os autores argumentam que sua estrutura é altamente escalável, suportando um número ilimitado de usuários e sendo adequada para documentos de grande porte.

Relativamente à tecnologia empregada para permitir a comunicação, um tutorial ilustra o uso de *Yjs* com Websocket.[4] Outra integração exemplificada é com o banco de dados *in memory* Redis.[5]

[3] https://github.com/yjs/yjs
[4] https://github.com/yjs/y-websocket
[5] https://github.com/yjs/y-redis

```
@Component({
  selector: 'app-product-form',
  templateUrl: './product-form.component.html',
  styleUrls: ['./product-form.component.scss']
})
export class ProductFormComponent {
  form : FormGroup;

  constructor(private formBuilder : FormBuilder,
    private productService : ProductsService,
    private location: Location){

    this.form = this.formBuilder.group({
      idProduct: [null],
      name: [null],
      value: [null]
    })

  }

  onSubmit(){
    this.productService.save(this.form.value).subscribe(volta => {this.onCancel()});
  }

  onCancel(){
    this.location.back();
  }
```

Figura 7.33: Conteúdo do *product-form.component.ts*

```typescript
import { Injectable } from '@angular/core';
import {HttpClient} from '@angular/common/http'
import { Product } from '../models/product';
import { WebSocketConnector } from './web.socket.connector';
import { interval } from 'rxjs';
import { TimeInterval } from 'rxjs/internal/operators/timeInterval';
@Injectable({
  providedIn: 'root'
})
export class ProductsService {
  private readonly API = '/api/products';

  constructor(private httpClient: HttpClient) {
  }
  list() {
    return this.httpClient.get<Product[]>(this.API);
  }
  save(record : Product){
    return this.httpClient.post<Product>(this.API,record);
  }
  delete(idProduct : String){
    return this.httpClient.delete(this.API+"/" +idProduct);
  }

}
```

Figura 7.34: Código fonte do *service products.service.ts*

```
export class ProductsComponent implements OnInit {
  products : Observable<Product[]>;
  displayedColumns = ['idProduct', 'name', 'value',"actions"];
  //private webSocket: WebSocket;
  private readonly API = '/api/products';
  private webSocketConnector: WebSocketConnector;
  constructor(private productService: ProductsService,
    private router: Router,
    private route: ActivatedRoute){
    this.webSocketConnector = new WebSocketConnector(
      'http://localhost:8080/api/socket',
      '/statusProcessor',
      this.espera.bind(this)
    );
    this.products = this.productService.list();
  }
  ngOnInit(): void {
  }
  onAdd(){
    this.router.navigate(['new'],{relativeTo: this.route});
  }
  onDelete(idProduct : string){
    this.productService.delete(idProduct).subscribe(  );
  }
  espera(message : any = {}){
    console.log(message);
    this.products = this.productService.list();
    console.log("Foi....");
  }
```

Figura 7.35: Descrição do conteúdo do *products.component.ts*

O *Yjs* oferece suporte a diversas aplicações[6] e editores de texto colaborativos empregados na web atualmente. Assim, os autores argumentam que aplicações que empregam o framework permitem, entre outros, edição offline, controle de versões, funções de desfazer/refazer, cursores compartilhados e ciência de quais usuários estão online e colaborando ao mesmo tempo. Por exemplo, recursos para informar usuários que utilizam a aplicação que há outros editores participando da colaboração são demonstrados em um dos tutoriais.[7]

Entre os tutoriais disponíveis para introdução ao *Yjs*, George (2024)[8] ilustra o uso do *Yjs* para construir um aplicativo colaborativo em tempo real usando CRDT com o *Angular*. Nesse caso, o autor utiliza *Yjs* para tornar colaborativo um editor amplamente utilizado na Web, o *Codemirror*.[9]

Outra demonstração de integração com editor web é com o *Quill*[10] Editor.[11] Além do *Codemirror* e do *Quill*, exemplos de integração com outros editores também estão disponíveis.[12]

Ainda em termos do framework *Yjs*, o *Hocuspocus* (Jahns, 2018a) é um framework que oferece um servidor para sincronização de documentos em tempo real, projetado para trabalhar em conjunto com o *Yjs*.

7.7. Considerações Finais

A literatura demonstra de forma clara que o trabalho em grupo colaborativo aumenta a produtividade, além de acarretar diversos outros benefícios (Ackerman et al., 2013; Kamel and Davison, 1998), mesmo que com problemas (Ma et al., 2023).

Na web, a colaboração remota síncrona está se tornando cada vez mais acessível aos usuários. Editores colaborativos em tempo real são exemplos típicos de aplicações que implementam essa modalidade de colaboração, permitindo que múltiplos usuários editem documentos simultaneamente.

Considerando protocolos e mecanismos da web, a especificação do padrão arquitetural REST definiu restrições a partir de conceitos e princípios de Engenharia de Software (Fielding, 2000). O Modelo de Suporte à Colaboração Síncrona em API REST (MoSCoR) estende o conjunto de restrições do padrão arquitetural REST com restrições para colaboração síncrona com base em requisitos associados a princípios de Engenharia de Software para edição colaborativa em tempo real (RTCE).

A aplicação de motivação registrada neste tutorial mostra que foi bem sucedida a extensão do modelo de autoria do sistema ESPIM, que é uma aplicação do tipo cliente-servidor baseada no padrão arquitetural REST, para que a aplicação permitisse edição colaborativa em tempo real. Ao orientar a extensão do padrão arquitetural original da aplicação, o MoSCoR facilitou a integração de funcionalidades colaborativas no sistema

[6] https://github.com/yjs/yjs
[7] https://docs.yjs.dev/getting-started/adding-awareness
[8] https://medium.com/blocksurvey/tutorial-how-to-build-a-real-time-collaborative-app
[9] https://github.com/yjs/y-codemirror
[10] https://quilljs.com/
[11] https://docs.yjs.dev/getting-started/a-collaborative-editor
[12] https://github.com/yjs/yjs-demos

sem a necessidade de mudanças estruturais complexas e sem a necessidade de alterar as características básicas do sistema.

Este tutorial detalhou a implementação de uma aplicação colaborativa de cadastro de produto. A aplicação de cadastro utilizou a mesma abordagem MoSCor utilizada no sistema ESPIM.

Embora tenha sido desenvolvido com o foco na extensão do modelo ESPIM, o MoSCoR se torna relevante para desenvolvedores em geral, dado que o padrão REST é atualmente o mais utilizado no desenvolvimento de sistemas web, e a capacidade de oferecer colaboração síncrona é uma funcionalidade desejável nesses sistemas.

O tutorial também apontou a disponibilidade de recursos como o *Yjs*, um framework que facilita que aplicações web sejam convertidas em aplicações colaborativas com o uso de CRDT, bem como contextualizou diversos projetos desenvolvidos com base no *Yjs*.

O MoSCoR e o *Yjs* são abordagens que buscam o mesmo fim, mas que são diferentes em sua essência. O *Yjs* oferece uma solução pronta que pode ser utilizada pelo desenvolvedor, enquanto o MoSCoR descreve o caminho que deve ser percorrido.

Referências

Mark S Ackerman, Juri Dachtera, Volkmar Pipek, and Volker Wulf. 2013. Sharing knowledge and expertise: The CSCW view of knowledge management. *Computer Supported Cooperative Work (CSCW)* 22 (2013), 531–573.

Noha Alsulami and Asma Cherif. 2017. Collaborative editing over opportunistic networks: State of the art and challenges. *International Journal of Advanced Computer Science and Applications* 8, 11 (2017), 264–276.

Ulrike Bath, Sumit Shekhar, Julian Egbert, Julian Schmidt, Amir Semmo, Jürgen Döllner, and Matthias Trapp. 2022. CERVI: collaborative editing of raster and vector images. *The Visual Computer* 38, 12 (2022), 4057–4070.

Jim Bauwens, Kevin De Porre, and Elisa Gonzalez Boix. 2023. [Short paper] Towards improved collaborative text editing CRDTs by using Natural Language Processing. In *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data* (Rome, Italy) *(PaPoC '23)*. Association for Computing Machinery, New York, NY, USA, 51–55. https://doi.org/10.1145/3578358.3591330

Abdelmajid Bouazza, Pascal Molli, et al. 2000. Unifying coupled and uncoupled collaborative work in virtual teams. In *ACM CSCW'2000 workshop on collaborative editing systems, Philadelphia, Pennsylvania, USA*. 6p.

Meire Cachioni, Isabela Zaine, Tássia Monique Chiarelli, Lilian Ourém Batista Vieira Cliquet, Kamila Rios da Hora Rodrigues, Bruna Carolina Rodrigues da Cunha, Leonardo Fernandes Scalco, Brunela Della Maggiori Orlandi, Maria da Graça C. Pimentel, and Samila Sathler Tavares Batistoni. 2019. Aprendizagem ao longo de toda a vida e letramento digital de idosos: um modelo multidisciplinar de intervenção com o apoio

de um aplicativo. *Revista Brasileira de Ciências do Envelhecimento Humano* 16, 1 (2019), 18–24.

Lilian Ourém Batista Vieira Cliquet, Maria da Graça Campos Pimentel, Samila Sathler Tavares Batistoni, Kamila Rios da Hora Rodrigues, Isabela Zaine, and Meire Cachioni. 2021. Idosos on-line: desenvolvimento de intervenção educativa em letramento digital. *Velho-ser: um olhar interdisciplinar sobre o envelhecimento humano* (2021), 45–50.

Lilian Ourém Batista Vieira Cliquet, Maria da Graça Campos Pimentel, Samila Sathler Tavares Batistoni, Kamila Rios da Hora Rodrigues, Isabela Zaine, and Meire Cachioni. 2023. Use of smartphones by older adults: characteristics and reports of students enrolled at a University of the Third Age (U3A). *PerCursos* 24 (2023), 1–30.

Bruna Carolina Rodrigues Cunha, Kamila Rios Da Hora Rodrigues, Isabela Zaine, Elias Adriano Nogueira da Silva, Caio César Viel, and Maria Da Graça Campos Pimentel. 2021. Experience sampling and programmed intervention method and system for planning, authoring, and deploying mobile health interventions: design and case reports. *Journal of Medical Internet Research* 23, 7 (2021), e24278.

Bruna Carolina Rodrigues da Cunha. 2019. *ESPIM: um modelo para guiar o desenvolvimento de sistemas de intervenção a distância*. Tese de Doutorado em Ciências de Computação e Matemática Computacional. Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos. `https://doi.org/10.11606/T.55.2019.tde-29082019-090151` Acesso em: 01 de outubro de 2024.

Liangrun Da and Martin Kleppmann. 2024. Extending JSON CRDTs with Move Operations. In *Proceedings of the 11th Workshop on Principles and Practice of Consistency for Distributed Data* (Athens, Greece) *(PaPoC '24)*. Association for Computing Machinery, New York, NY, USA, 8–14. `https://doi.org/10.1145/3642976.3653030`

Laurentino Augusto Dantas. 2024. *Autoria Colaborativa de Intervenções Programadas e Amostragem de Experiências: Um estudo de caso com o ESPIM*. (**submetida**) Tese de Doutorado em Ciências de Computação e Matemática Computacional. Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos.

Laurentino Augusto Dantas, Joab Cavalcante da Silva, Raphael Christian dos Santos Oliveira, Lucas Fidelis Pereira, Kamila Rios da Hora Rodrigues, and Maria da Graça Campos Pimentel. 2023. Descrição do processo de implementação de recursos para autoria colaborativa síncrona na plataforma ESPIM. *Caderno Pedagógico* 20, 6 (2023), 2244–2269.

Laurentino Augusto Dantas, Joab Cavalcante da Silva, and Maria da Graça C. Pimentel. 2024. Desenvolvimento de Editores Colaborativos em Tempo Real: Revisão Rápida. In *Workshop de Revisões Sistemáticas de Literatura em Sistemas Multimídias e Web - WebMedia 2024*. SBC, 124–142. No prelo.

Andrea De Lucia, Fausto Fasano, Giuseppe Scanniello, and Genny Tortora. 2007. Enhancing collaborative synchronous UML modelling with fine-grained versioning of software artefacts. *Journal of Visual Languages & Computing* 18, 5 (2007), 492–503.

C. A. Ellis and S. J. Gibbs. 1989. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data* (Portland, Oregon, USA) *(SIGMOD '89)*. Association for Computing Machinery, New York, NY, USA, 399–407. https://doi.org/10.1145/67544.66963

Thore Fechner, Dennis Wilhelm, and Christian Kray. 2015. Ethermap: real-time collaborative map editing. In *Proceedings of the 33rd ACM Conference on Human Factors in Computing Systems*. 3583–3592.

Roy Thomas Fielding. 2000. REST: architectural styles and the design of network-based software architectures. *Doctoral dissertation, University of California* (2000).

Roy T. Fielding, Richard N. Taylor, Justin R. Erenkrantz, Michael M. Gorlick, Jim Whitehead, Rohit Khare, and Peyman Oreizy. 2017. Reflections on the REST architectural style and "principled design of the modern web architecture"(impact paper award). In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 4–14. https://doi.org/10.1145/3106237.3121282

Karina de Lima Flauzino, Maria da Graça Campos Pimentel, Samila Sathler Tavares Batistoni, Isabela Zaine, Lilian Ourém Batista Vieira, Kamila Rios da Hora Rodrigues, and Meire Cachioni. 2020. Letramento Digital para Idosos: percepções sobre o ensino-aprendizagem. *Educação & Realidade* 45 (2020), 1–17.

Cristian Gadea. 2021. *Architectures and Algorithms for Real-Time Web-Based Collaboration*. Ph. D. Dissertation. Université d'Ottawa/University of Ottawa.

Leonardo de Freitas Galesky and Luiz Antonio Rodrigues. 2023. Efficient CRDT Synchronization at Scale using a Causal Multicast over a Virtual Hypercube Overlay. In *Proceedings of the 11th Latin-American Symposium on Dependable Computing* (Fortaleza/CE, Brazil) *(LADC '22)*. Association for Computing Machinery, New York, NY, USA, 84–88. https://doi.org/10.1145/3569902.3569948

Alen George. 2024. Tutorial: How to Build a Real-Time Collaborative App Using CRDT in Angular. https://medium.com/blocksurvey/tutorial-how-to-build-a-real-time-collaborative-app-using-crdt-in-ang Accessed: 2024-10-10.

Ahmad Hemid, Waleed Shabbir, Abderrahmane Khiat, Christoph Lange, Christoph Quix, and Stefan Decker. 2024. OntoEditor: Real-Time Collaboration via Distributed Version Control for Ontology Development. In *European Semantic Web Conference*. Springer, 326–341.

L. Henry, E. Hansen, J. Chimoff, K. Pokstis, M. Kiderman, R. Naim, J. Kossowsky, M. Byrne, S. Lopez-Guzman, K. Kircanski, D. Pine, and M. Brotman. 2024. Selecting an

Ecological Momentary Assessment Platform: Tutorial for Researchers. *J Med Internet Res* 26 (2024), e51125. https://doi.org/10.2196/51125

IBM. 2024. O que é Java Spring Boot? http://https://www.ibm.com/br-pt/topics/java-spring-boot Acessado em 20/08/2024.

Ryota Inoue, Yudai Kato, Takushi Goda, Tadachika Ozono, Shun Shiramatsu, and Toramatsu Shintani. 2012. A real-time collaborative mechanism for editing a web page and its applications. In *2012 Fifth International Symposium on Parallel Architectures, Algorithms and Programming*. IEEE, 186–193.

Kevin Jahns. 2018a. HocusPocus - Collaborative editing. https://tiptap.dev/docs/hocuspocus/guides/collaborative-editing

Kevin Jahns. 2018b. YJS - A CRDT framework with a powerful abstraction of shared data. https://github.com/yjs/yjs

Andrew Jeffery and Richard Mortier. 2023. AMC: Towards Trustworthy and Explorable CRDT Applications with the Automerge Model Checker. In *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data* (Rome, Italy) *(PaPoC '23)*. Association for Computing Machinery, New York, NY, USA, 44–50. https://doi.org/10.1145/3578358.3591326

Tim Jungnickel and Tobias Herb. 2016. Simultaneous editing of JSON objects via operational transformation. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. 812–815.

Nabil N Kamel and Robert M Davison. 1998. Applying CSCW technology to overcome traditional barriers in group interactions. *Information & Management* 34, 4 (1998), 209–219.

Shin-Ya Katayama, Takushi Goda, Shun Shiramatsu, Tadachika Ozono, and Toramatsu Shintani. 2013. A fast synchronization mechanism for collaborative web applications based on HTML5. In *2013 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. IEEE, 663–668.

Martin Kleppmann. 2020. Moving elements in list CRDTs. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '20)*. ACM, Article 4, 6 pages. https://doi.org/10.1145/3380787.3393677

Elias Kuiter, Sebastian Krieter, Jacob Krüger, Gunter Saake, and Thomas Leich. 2021. variED: an editor for collaborative, real-time feature modeling. *Empirical Software Engineering* 26, 2 (2021), 24.

Dilshodbek Kuryazov and Andreas Winter. 2014. Representing model differences by delta operations. In *2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations*. IEEE, 211–220.

Dilshodbek Kuryazov and Andreas Winter. 2015. Towards Model History Analysis Using Modeling Deltas. *Softwaretechnik-Trends Band 35, Heft 2* (2015).

Dilshodbek Kuryazov, Andreas Winter, and Ralf Reussner. 2018. Collaborative modeling enabled by version control. (2018).

Sobirov J.Sh. Kuryazov D.A., Jumanazarov B.B. 2016. Software Model Version Control And Collaboration. https://cyberleninka.ru/article/n/software-model-version-control-and-collaboration. *XXI* (2016), 6–3.

Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks, and Joseph M. Hellerstein. 2022. Keep CALM and CRDT On. *Proc. VLDB Endow.* 16, 4 (Dec. 2022), 856–863. https://doi.org/10.14778/3574245.3574268

Reed Larson and Mihaly Csikszentmihalyi. 1978. Experiential correlates of time alone in adolescence 1. *Journal of Personality* 46, 4 (1978), 677–693.

Reed Larson and Mihaly Csikszentmihalyi. 2014. The experience sampling method. In *Flow and the foundations of positive psychology*. Springer, 21–34.

Janne Lautamäki, Antti Nieminen, Johannes Koskinen, Timo Aho, Tommi Mikkonen, and Marc Englund. 2012. CoRED: browser-based Collaborative Real-time Editor for Java web applications. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*. 1307–1316.

Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter van Hardenberg. 2022a. Peritext: A CRDT for Collaborative Rich Text Editing. *Proc. ACM Hum.-Comput. Interact.* 6, CSCW2, Article 531 (Nov. 2022), 36 pages. https://doi.org/10.1145/3555644

Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter van Hardenberg. 2022b. Peritext: A CRDT for Collaborative Rich Text Editing. *Proc. ACM Hum.-Comput. Interact.* 6, CSCW2, Article 531 (nov 2022), 36 pages. https://doi.org/10.1145/3555644

Renkai Ma, Yue You, Xinning Gui, and Yubo Kou. 2023. How Do Users Experience Moderation?: A Systematic Literature Review. *Proc. ACM Hum.-Comput. Interact.* 7, CSCW2, Article 278 (Oct. 2023), 30 pages. https://doi.org/10.1145/3610069

Brice Nédelec, Pascal Molli, and Achour Mostefaoui. 2016. Crate: Writing stories together with our browsers. In *Proceedings of the 25th International Conference Companion on World Wide Web*. 231–234.

Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. 2013. LSEQ: an adaptive structure for sequences in distributed collaborative editing. In *Proceedings of the 2013 ACM Symposium on Document Engineering* (Florence, Italy) *(DocEng '13)*. Association for Computing Machinery, New York, NY, USA, 37–46. https://doi.org/10.1145/2494266.2494278

David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. 1995. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology (UIST '95)*. ACM, 111–120.

Petru Nicolaescu, Mario Rosenstengel, Michael Derntl, Ralf Klamma, and Matthias Jarke. 2018. Near real-time collaborative modeling for view-based web information systems engineering. *Information Systems* 74 (2018), 23–39.

Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. 2006. Data consistency for P2P collaborative editing. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*. 259–268.

Tadachika Ozono, Robin ME Swezey, Shun Shiramatsu, Toramatsu Shintani, Takushi Goda, Yudai Kato, and Ryota Inoue. 2012. Differential Synchronizaiton Mechanism for a Real-Time Collaborative Web Page Editing System WFE-S. In *IIAI International Conference on Advanced Applied Informatics*. IEEE, 242–247.

Maria da Graça Campos Pimentel, AC Rocha, BC Cunha, AF Orlando, O Machado Neto, C Viel, E Antunes, and I Zaine. 2016. Apoio ao envelhecimento no lugar por meio de amostragem de experiências e de intervenção programada. *Medicina* 49, 2 (2016), 11–12.

Postman. 2023. 2023 State of the API Report. Accessed: 2024-06-21.

Kamila Rodrigues, Isabela Zaine, Brunela Orlandi, and Maria da Graça Pimentel. 2021. Ensinando configurações do smartphone e aplicações sociais para o público 60+ por meio de aulas semanais e intervenções remotas. In *Anais do XII Workshop sobre Aspectos da Interação Humano-Computador para a Web Social*. SBC, 25–32. https://doi.org/10.5753/waihcws.2021.17541

Gabriele Salvati, Christian Santoni, Valentina Tibaldo, and Fabio Pellacini. 2015. Meshhisto: Collaborative modeling by sharing and retargeting editing histories. *ACM Transactions on Graphics (TOG)* 34, 6 (2015), 1–10.

Rodolfo S Sanches, Moacir A Ponti, and Kamila R Rodrigues. 2022. Evasao Universitária e Estratégias para Retençao de Alunos com Base em Intervençoes Remotas. In *Anais Estendidos do XXI Simpósio Brasileiro de Fatores Humanos em Sistemas Computacionais*. SBC, 84–87.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. Springer, 386–400.

Haifeng Shen and Chengzheng Sun. 2002. Highlighting: a gesturing communication tool for real-time collaborative systems. In *International Conference on Algorithms and Architectures for Parallel Processing 2002*. IEEE, 180–187.

Chengzheng Sun, David Sun, Agustina Ng, Weiwei Cai, and Bryden Cho. 2020c. Real Differences between OT and CRDT under a General Transformation Framework for Consistency Maintenance in Co-Editors. *Proc. ACM Hum.-Comput. Interact.* 4, GROUP, Article 06 (jan 2020), 26 pages. https://doi.org/10.1145/3375186

David Sun, Chengzheng Sun, Agustina Ng, and Weiwei Cai. 2020a. Real Differences between OT and CRDT in Building Co-Editing Systems and Real World Applications. arXiv:1905.01517 [cs.DC] https://arxiv.org/abs/1905.01517

David Sun, Chengzheng Sun, Agustina Ng, and Weiwei Cai. 2020b. Real Differences between OT and CRDT in Correctness and Complexity for Consistency Maintenance in Co-Editors. *Proc. ACM Hum.-Comput. Interact.* 4, CSCW1, Article 21 (may 2020), 30 pages. https://doi.org/10.1145/3392825

Christian Thum, Michael Schwind, and Martin Schader. 2009. SLIM—A lightweight environment for synchronous collaborative modeling. In *12th International Conference Model Driven Engineering Languages and Systems, MODELS 2009*. Springer, 137–151.

Khushwant Virdi, Anup Lal Yadav, Azhar Ashraf Gadoo, and Navjot Singh Talwandi. 2023. Collaborative Code Editors-Enabling Real-Time Multi-User Coding and Knowledge Sharing. In *2023 3rd International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)*. IEEE, 614–619.

April Yi Wang, Zihan Wu, Christopher Brooks, and Steve Oney. 2024. "Don't Step on My Toes": Resolving Editing Conflicts in Real-Time Collaboration in Computational Notebooks. *arXiv preprint arXiv:2404.04695* (2024).

Ruipeng Wei, Ruisheng Zhang, Chen Zhao, Dongmei Yue, and Lian Li. 2009. Design and Implementation of Scientific Collaborative Editing Environment on Chemistry. In *International Conference on Grid and Cooperative Computing'2009*. IEEE, 188–192.

Elena Yanakieva, Philipp Bird, and Annette Bieniusa. 2023. A Study of Semantics for CRDT-based Collaborative Spreadsheets. In *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data* (Rome, Italy) *(PaPoC '23)*. Association for Computing Machinery, New York, NY, USA, 37–43. https://doi.org/10.1145/3578358.3591324

Isabela Zaine, Priscila Benitez, Kamila Rios da Hora Rodrigues, and Maria da Graça Campos Pimentel. 2019a. Applied behavior analysis in residential settings: use of a mobile application to support parental engagement in at-home educational activities. *Creative Education* 10, 8 (2019), 1883–1903.

Isabela Zaine, David Frohlich, Kamila Rios da Hora Rodrigues, Bruna Carolina Rodrigues da Cunha, Alex Fernando Orlando, Leonardo Fernandes Scalco, and Maria Da Graça Campos Pimentel. 2019b. Promoting Social Connection and Deepen Relations in Older People: Design of Media Parcels towards facilitating Time-based Media Sharing. *Journal of Medical Internet Research* 21, 10 (2019).

Bio

Laurentino Augusto Dantas é formado em Processamento de Dados pela Universidade Norte do Paraná (1993) e em Direito pela Universidade Paranaense (2007), possui especializações em Engenharia de Software (Universidade Norte do Paraná, 1996) e em Docência para a Educação Profissional, Científica e Tecnológica (Instituto Federal do Mato Grosso do Sul, 2016). Detém mestrado em Ciência da Computação pela Universidade Federal de Santa Catarina (2001) e atualmente atua como professor EBTT no Instituto Federal de Educação, Ciência e Tecnologia de Mato Grosso do Sul, além de ser doutorando no Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo. Suas áreas de interesse incluem Desenvolvimento Web, Interação Humano-Computador, Tecnologias da Informação e Comunicação na Educação, Ambientes Virtuais de Aprendizagem, Jogos Educacionais, Empreendedorismo, Inovação Tecnológica, Comércio Eletrônico e Envelhecimento Ativo.

Maria da Graça Campos Pimentel é Professora Sênior no ICMC-USP. Foi Professora Titular na USP de 2011 a 2021, tendo ingressado na carreira docente em 1987 depois de atuar em empresas de desenvolvimento de software por quatro anos. Graduada em Ciências da Computação pela UFSCar, com mestrado e Livre-docência pela USP e doutorado pela University of Kent, realizou estágio sabático no Georgia Tech. Exerceu cargos como chefia do Departamento de Ciências da Computação do ICMC, presidência da Comissão de Cultura e Extensão do ICMC, coordenação do curso de Bachalerado em Ciências da Computação e coordenação do Programa de Pós-graduação em Ciências Matemáticas e de Computação do ICMC. Atuou como assessora do Comitê de Área na CAPES. Foi vice-chair e treasurer da ACM SIGWEB, além de coordenar a Comissão Especial em Sistemas Multimídia e Web da SBC. Suas pesquisas abrangem Web, Multimídia, Interação Humano-Computador e Tecnologia Assistiva. Recebeu três prêmios pelo projeto Meninas Programadoras.