Explorando a Arquitetura de *Server Driven UI*

Calel Becassi Benis[1], **Julio Huang**[1], **Thaisla Camila Rener Romeiro**[1], **Awdren de Lima Fontão**[1]

[1]FACOM – Universidade Federal de Mato Grosso do Sul (UFMS)

{calel.becassi,julio.huang,thaisla.rener,awdren.fontao}@ufms.br

Abstract. *Server Driven UI is a concept used by software developers that aims to control the user interface from the server. Thus, it can reduce the time of the mobile application update process and customize each interface. However, there is little literature on this approach. Therefore, this work seeks to explore the concept and architecture of SDUI. Aiming to contribute to the advancement of knowledge on the topic and identify possible gaps in the components, based on the survey of documentation.*

Resumo. *O Serven Driven UI é um conceito utilizado por desenvolvedores de software que visa controlar a interface de usuário pelo servidor. Assim, conseguindo diminuir o tempo do processo de atualização de um aplicativo móvel e podendo personalizar cada interface. No entanto, há pouca literatura sobre essa abordagem. Por isso, este trabalho busca explorar o conceito e a arquitetura do SDUI. Tendo o objetivo de contribuir para o avanço do conhecimento sobre o tema e identificar possíveis lacunas dos componentes, com base no levantamento de documentação.*

1. Introdução

No desenvolvimento de aplicativos móveis (*Android* e *iOS*), é comum que as interfaces sejam definidas no aplicativo instalado no aparelho. Para o desenvolvedor lançar uma atualização do aplicativo, a nova versão requer a liberação da loja de aplicativos e os usuários necessitam atualizar manualmente. Além disso, as empresas enfrentam desafios relacionados a dependência dos usuários finais para atualizar o aplicativo e os processos burocráticos das lojas de aplicativos, que podem aumentar o tempo de desenvolvimento e implementação de novos recursos, correções de *bugs* e mudanças na interface. [Rucker 2022]

Uma alternativa para superar esses desafios é a abordagem *Server Driven UI* (Interface de Usuário Controlada pelo Servidor). Com o *Server Driven UI* (SDUI), as atualizações na interface podem ser implementadas no ambiente de produção, sem a necessidade de lançamento de uma nova versão do aplicativo. Além de enviar dados para serem exibidos na interface, o servidor também envia informações sobre como construir, exibir a interface e reagir às interações do usuário. [Sampaio 2023]

Essa abordagem oferece um maior controle sobre as atualizações dos aplicativos, permitindo a correção rápida de *bugs*, redução no tamanho da aplicação e a possibilidade de disponibilizar diferentes versões do aplicativo para grupos específicos de usuários. Além disso, permite a exploração de diversas hipóteses com um grande número de usuários e a realização de mudanças sazonais nos aplicativos. [Ramos 2020]

Como mencionado anteriormente, a adoção do *Server Driven UI* pode reduzir a dependência do usuário final para atualizar o aplicativo, já que a maior parte das atualizações ocorrem no servidor. Isso é relevante, considerando que muitos usuários não mantêm seus aplicativos atualizados devido a restrições de armazenamento ou falta de conhecimento sobre a importância das atualizações de segurança. [Ramos 2020]

Um exemplo de solução para desenvolvimento de aplicativo utilizando *Server Driven UI* é o *Beagle* [1], uma ferramenta *open source* que permite escrever o código apenas uma vez para que as telas funcionem em diferentes plataformas (*Android*, *iOS* e *Web*). Ao chamar o *Beagle*, o aplicativo recebe um JSON contendo as informações necessárias para construir a interface de usuário. Com isso, a maioria das atualizações pode ser feita diretamente no servidor, o que reduz a necessidade de passar pelo processo de aprovação nas lojas de aplicativos. Dessa forma, é possível evitar o tempo de espera de uma a duas semanas para que a atualização seja aprovada. [Ferreira 2020]

No entanto, o *Server Driven UI* apresenta desafios arquiteturais, como a definição dos componentes no servidor. Apesar da crescente popularidade do SDUI, ainda há pouca literatura sobre como projetar e implementar essa abordagem. Neste trabalho, os autores realizam uma revisão sistemática da literatura sobre o SDUI, selecionando os trabalhos que apresentam mais detalhes sobre os aspectos arquiteturais envolvidos. A partir dessa revisão, os autores identificam as possíveis lacunas existentes nos componentes da arquitetura do SDUI, e propõem soluções viáveis baseadas em boas práticas, interação da comunidade e artigos. O objetivo deste trabalho é contribuir para o avanço do conhecimento sobre o SDUI.

2. Fundamentação Teórica

Conhecer a arquitetura cliente-servidor é essencial para entender o funcionamento do *Server Driven UI* (SDUI). Nessa arquitetura, o cliente é a parte do sistema que interage diretamente com o usuário e é responsável por enviar as solicitações ao servidor. O servidor, por sua vez, é responsável por processar essas solicitações, executar as tarefas necessárias e fornecer uma resposta ao cliente. [Fileto 2006]

O cliente é o chamado *front-end* da aplicação, ou seja, a interface com a qual o usuário interage. Ele envia as requisições ao servidor e lida com as interações e validações dos dados informados pelo usuário. Por outro lado, o servidor é o *back-end*, o responsável por responder às solicitações do cliente e realizar as tarefas necessárias, como pesquisas, filtragens e atualizações em bancos de dados. [Fileto 2006]

A abordagem de *Back-End for Front-End* (BFF) é uma aplicação específica da arquitetura cliente-servidor que se encaixa no conceito de *Server-Driven UI*. De acordo com o Sam Newman [Newman 2015], com o *Back-End for Front-End*, o servidor consegue fornecer APIs específicas para cada interface do usuário (*front-end*), ou seja, ele cria um *back-end* dedicado para cada *front-end*, ao invés de oferecer uma API de uso geral. O *framework Beagle* utiliza o conceito BFF, como mostra na Figura 1.

[1] O *Beagle* é uma plataforma *Server-Driven* para *iOS*, *Android* e *Web* que facilita a construção de telas por meio de um *backend*, permitindo que telas e fluxos nativos sejam alterados utilizando apenas JSON.

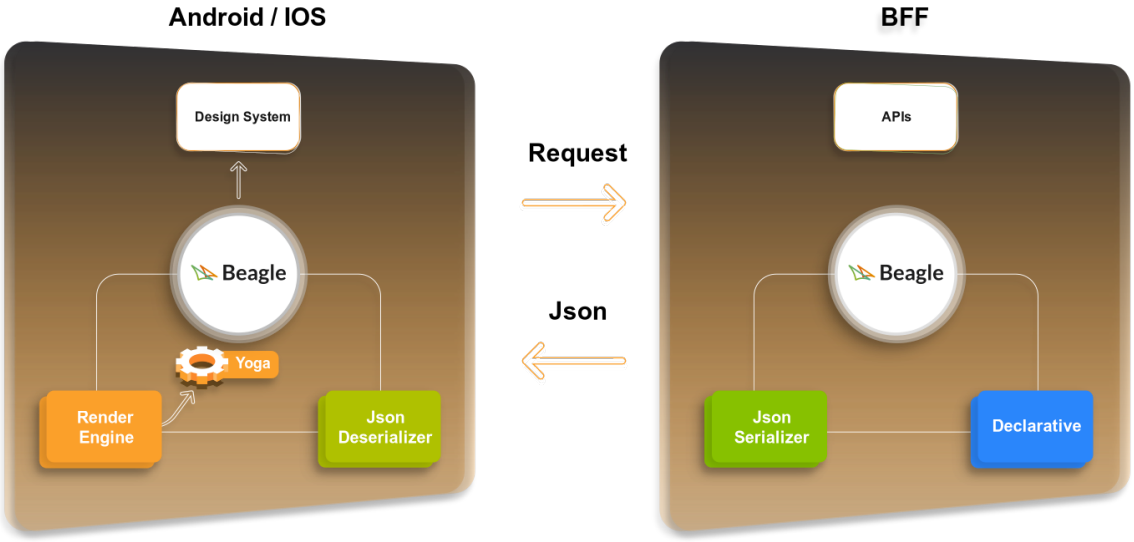

Figura 1. *Framework* que atualiza UI via *back-end*

Server-Driven UI é uma abordagem de desenvolvimento de interface do usuário que visa transferir parte da lógica de apresentação e *layout* da aplicação cliente para o servidor. Isso significa que, em vez de o aplicativo cliente ser responsável por toda a lógica de exibição, o servidor fornece informações sobre como a interface do usuário deve ser montada e renderizada no dispositivo do usuário.

Esse método tem ganhado popularidade à medida que a variedade de dispositivos e plataformas aumentam, como *desktops*, *smartphones*, *tablets* e dispositivos inteligentes em geral. Cada um desses dispositivos possui tamanhos de tela, resoluções e capacidades diferentes, o que torna desafiador para os desenvolvedores criarem uma experiência de usuário consistente e otimizada para todos os cenários.

O SDUI permite que cada *front-end* tenha uma API customizada de acordo com as suas necessidades específicas, aproveitando toda a capacidade do dispositivo utilizado. Por exemplo, dispositivos móveis podem se beneficiar de funcionalidades que somente eles possuem, como um leitor de *QRcode*. Enquanto *desktops* podem receber funcionalidades adaptadas ao seu ambiente, como um código de pareamento. [Newman 2015]

Essa abordagem traz vantagens significativas, pois torna a experiência do usuário mais rápida e eficiente, reduzindo a complexidade das APIs e garantindo um melhor desempenho para cada plataforma. Além disso, permite uma maior flexibilidade na criação de interfaces ricas e interativas, de acordo com as particularidades de cada dispositivo.

Em resumo, o *Server-Driven UI* com a arquitetura cliente-servidor e a aplicação do *Back-End for Front-end* oferece uma solução eficaz para criar interfaces de usuário adaptáveis e otimizadas para diferentes dispositivos, garantindo uma experiência mais consistente e satisfatória para os usuários.

3. Trabalhos Relacionados

Neste trabalho de pesquisa, foi conduzida uma análise de artigos científicos relacionados à *Server Driven UI*. Embora a disponibilidade de literatura tenha sido limitada, foram

identificadas algumas informações que contribuem para a compreensão do tema.

Carvalho et al. [Carvalho et al. 2022] se concentraram na implementação da biblioteca que utiliza o conceito de SDUI voltadas para *e-commerce*. Os estudos dos autores eram diminuir a demora no processo de atualização de aplicativos móveis. Seus estudos demonstraram que a implementação é simples e satisfatória para os programadores. Contudo a manutenção da biblioteca realizada por outros desenvolvedores se tornaram complexas. Neste trabalho o foco é na arquitetura do SDUI, diferenciando-se da implementação da biblioteca.

Segundo o estudo de Pete e Patricio [Hdgson and Echague 2019], as *Feature Flags* oferecem benefícios para a entrega de recursos em um software, tais como flexibilidade e controle. Esses benefícios também são observados no *Server Driven UI*, que possibilita a personalização e a gestão da experiência do usuário. Tanto as *Feature Flags* quanto o SDUI seguem o princípio de separar a lógica do servidor da apresentação do cliente.

Porém, enquanto as *Feature Flags* permitem aos desenvolvedores habilitar ou desabilitar recursos de forma dinâmica. O SDUI vai além, permitindo que a própria interface do usuário seja personalizada com base em informações e decisões do servidor. Isso significa que as experiências dos usuários podem ser adaptadas gradativamente, mostrando diferentes elementos de interface e layouts com base em regras de negócios definidas no servidor.

Já o Kyle Brown e o Bobby Woolf [Brown and Woolf 2016] realizaram um conjunto de pesquisas sobre padrões de implementações para arquiteturas de microsserviços. Embora o foco da pesquisa deles não tenham sido sobre *Server Driven UI*, uma das arquiteturas estudadas foi a abordagem de BFF (*Back-End for Front-End*), que também é utilizada pelo SDUI. Suas conclusões sugerem que essa abordagem pode ser vantajosa para diferentes tipos de clientes (*web* e *mobile*), pois pode gerar interfaces específicas para suas necessidades e pode tornar o desenvolvimento mais eficiente, utilizando uma única linguagem.

No decorrer deste trabalho, a arquitetura do *Server Driven UI* foi explorada, uma vez que os estudos relacionados identificados até o momento não abordaram de forma específica essa arquitetura. Tendo em vista os trabalhos apresentadas anteriormente, o foco deste estudo é explorar a arquitetura do SDUI. Com isso, identificar possíveis lacunas e soluções presentes nas aplicações desta arquitetura.

4. Estudo de Caso Exploratório

A implementação do SDUI (*Server Driven UI*) é uma abordagem que permite personalização dinâmica da interface do usuário em aplicativos móveis e *web*. No entanto, essa abordagem também pode apresentar alguns desafios e lacunas. Este estudo de caso visa explorar as lacunas que podem surgir na arquitetura do SDUI e identificar possíveis estratégias para superá-los.

4.1. Objetivo do estudo

Identificar lacunas e seus possíveis impactos na arquitetura do *Server Driven UI* e na experiência do usuário final. Desse modo, os autores irão explorar estratégias viáveis para superar essas lacunas.

4.2. Metodologia

A metodologia abordada neste trabalho é a coleta, estudo e análise de dados através de pesquisas, leituras de artigos (trabalhos relacionados) e discussões entre os autores. Buscando por elementos e fontes que tinham o uso ou demonstração do estilo de arquitetura ou padrão arquitetural utilizado pelo SDUI.

Após realizar a junção e a síntese do material existente que define a arquitetura do SDUI, os autores realizaram uma análise dos componentes em comum. Em seguida, foi mapeado o que era conceitualmente, cada componente. Com isso, levantando os pontos positivos e negativos, selecionando lacunas e estratégias de resoluções que podem ser exploradas.

4.3. Seleção de lacunas

Uma lacuna, de acordo com o dicionario online DICIO [DICIO] (Lexicógrafa responsável: Débora Ribeiro), representa uma falha, brecha, aquilo que falta em algo (real ou abstrato). Foi realizado um levantamento no *Github* de *frameworks* que utilizam a abordagem do SDUI. Os *frameworks* que possuiam mais destaque na comunidade eram o *Beagle*, o *DivKit* e o *Flet*. Dentre eles, com base na análise da documentação, o *Beagle* foi escolhido por ser o mais completo.

Com isso, a arquitetura utilizada como referência para este estudo foi a do *framework Beagle*. Para análise de componentes, os autores simplificaram a arquitetura do *Beagle* para melhor compreensão (Figura 2). Com base no levantamento de documentação realizado pelos autores, os tópicos que mais possuíam documentação e atividade da comunidade eram os componentes de Serialização, Componente UI e Cache. A partir disso foram selecionados esses três componentes afim de facilitar a análise, levantar informações que estão relacionadas a esses tópicos da arquitetura e estudar as possíveis lacunas encontradas.

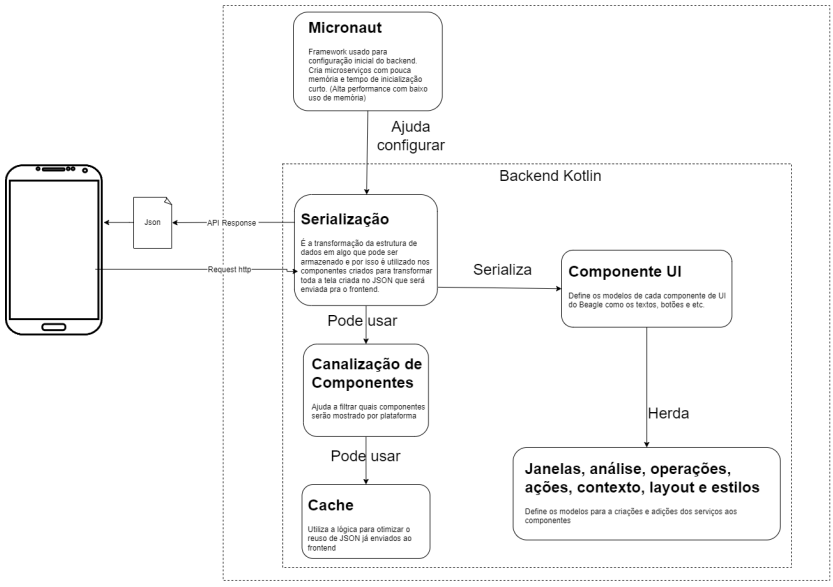

Figura 2. Simplificação da arquitetura do *framework Beagle*

4.3.1. Serialização

A serialização é um procedimento que transforma uma estrutura de dados ou um objeto em um formato que pode ser armazenado ou enviado (Figura 3). Por exemplo JSON, XML, Yaml e ProtoBuf. É importante notar que a serialização é um processo independente da aplicação, o que significa que um conjunto de dados serializado em uma plataforma específica deve ser capaz de ser convertido de volta (desserializado) em qualquer outra plataforma. Isso é fundamental para assegurar a interoperabilidade entre diferentes aplicações. [Alencar 2019]

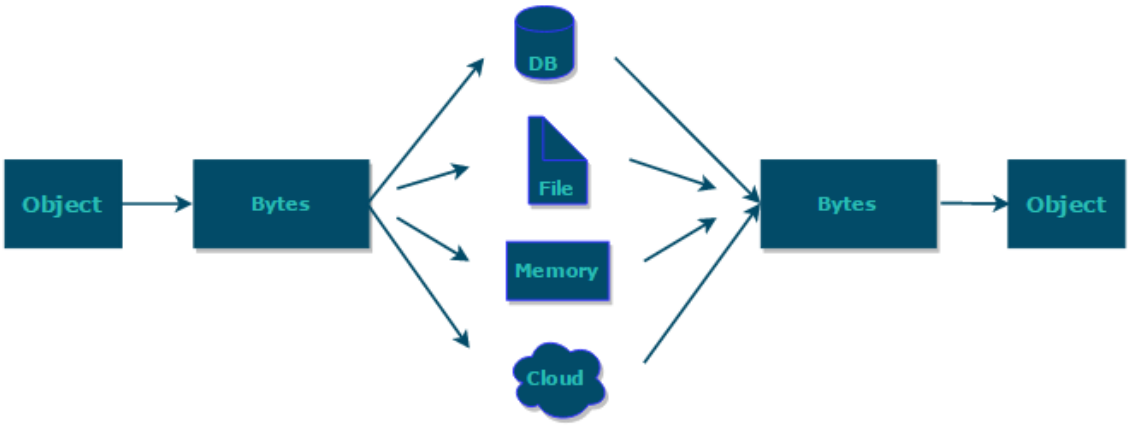

Figura 3. Processo de serialização

Arquitetura	Lacunas	Explicação
Serialização	Integridade dos dados	A integridade é um pilar para garantir um sistema seguro. A corrupção ou perda de dados pode causar problemas de funcionalidade do sistema e em alguns casos até falhas completas.
	Segurança dos dados	Como a serialização é uma sequência de bytes com informações puras, é possível ler qualquer informações do objeto serializado sem nenhuma segurança.
	Desempenho	Se o desempenho da serialização for lento, pode afetar a experiencia do usuário, pois a tela que será renderizada no cliente, depende da serialização do dado.

Tabela 1. Lacunas da Serialização

Uma forma de preservar a **integridade dos dados** durante o processo de serialização, é realizando *backups* periodicamente. Essa periodização pode ser toda semana, todo dia ou até mesmo toda hora dependendo da complexidade da aplicação. *Backups* são importantes para manter a integridade pois com ele pode-se garantir que mesmo

se os dados forem perdidos ou alterados indevidamente, ainda haverá uma cópia segura armazenada em um local seguro. Por isso é de suma importância manter os dados de *backups* atualizados frequentemente.

Outra forma de manter a integridade dos dados, é com geração de *hash* antes do processo de serialização, que se comparar com o *hash* gerado após o processo de desserialização, deve ser o mesmo. *Hashes* são resumos de arquivos ou dados de qualquer tamanho para um tamanho fixo. Essa representação pode ser comparada a um tipo de ”impressão digital”do arquivo ou dado. Ou seja, um mesmo arquivo que gerou o *hash* ”X”antes da serialização, após a desserialização deve gerar o mesmo *hash* ”X”, caso contrário o arquivo foi corrompido ou alterado durante o processo. [Rajeev and G.Geetha 2012]

Já quando se trata da **segurança desses dados**, além da integridade é importante manter a confidencialidade dos mesmos. Sendo assim uma alternativa segura é a criptografia. A criptografia converte os dados legíveis em algo sem sentido/ ilegível mas com a capacidade de recuperar os dados originais. Diferente inclusive do *hash* que não é possível obter os dados de onde ele foi resumido.

O **desempenho da serialização** depende de vários fatores, incluindo a linguagem de programação, a biblioteca utilizada, o tamanho e a complexidade dos dados a serem serializados, e o hardware em que o código está sendo executado. O formato em que os dados são serializados também faz diferença, alguns são mais eficientes em termos de espaço e tempo do que outros.

O formato JSON, por exemplo, é fácil de ler, mas pode ser menos eficiente em termos de tamanho e velocidade do que formatos binários mais compactos, como o *Protobuf*[2]. O protocolo de *buffers*, chamado de *Protobuf* é um exemplo de serialização de dados mais eficiente (Figura 4). [Krare 2018]

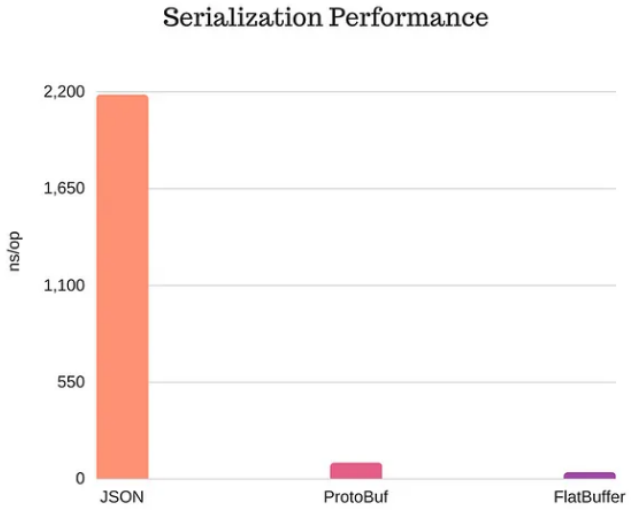

Figura 4. Performance Serialização

[2]*Protocol Buffers* (*Protobuf*) é um formato de serialização de dados desenvolvido pela Google. Armazena de forma eficiente e compacta dados estruturados em formato binário, permitindo uma transferência mais rápida através de ligações de rede.

É importante medir e analisar a serialização em seu contexto específico para determinar as melhores práticas de otimização.

4.3.2. Componentes UI

Os componentes UI, ou *Widgets*, no contexto do *Beagle*, são elementos pré-construídos que simplificam o desenvolvimento de aplicativos, eliminando a necessidade de criar componentes personalizados sempre que precisar de funcionalidades comuns em uma interface de usuário. Esses componentes básicos são peças fundamentais para a construção de interfaces de usuário e aplicativos móveis centrados no usuário, tornando o processo de desenvolvimento mais eficiente e conveniente.

Com isso os autores estudaram mais sobre o assunto, para poder identificar possíveis lacunas, como mostrado na Tabela 2.

Arquitetura	Lacunas	Explicação
Componente UI	Reutilização	Sem garantir reutilização dos componentes pode gerar um alto custo, esforço e tempo no futuro.
	Compatibilidade do componente	Ao fazer alterações nos componentes UI no *back-end*, é importante garantir que sejam compatíveis com a versão atual do *front-end*. Alterações que quebram a compatibilidade podem resultar em erros ou em comportamentos inesperados do aplicativo.
	Segurança do componente	Componentes vulneráveis e desatualizados podem permitir que um invasor execute um código malicioso em um dispositivo ou roube informações confidenciais.

Tabela 2. Lacunas do Componente UI

O Italo Aurelio [Aurelio 2020] destaca a importância da **reutilização** de componentes, alertando que a falta desse compromisso pode resultar em custos elevados, esforços extras e atrasos no futuro. O objetivo central é equilibrar utilidade e complexidade na criação de componentes reutilizáveis. Como uma peça de carro que é projetada uma vez e aplicada em muitos veículos, a reutilização promove otimização, consistência e eficiência. Em resumo, a criação de componentes reutilizáveis impulsiona melhorias, economia de custos e agilidade no desenvolvimento de projetos.

Ao criar componentes reutilizáveis, a escolha entre um componente rígido e flexível depende da complexidade e das necessidades do projeto. Componentes rígidos são menos customizáveis, mais simples de criar e manter, enquanto componentes flexíveis são mutáveis, adaptáveis e oferecem mais opções de personalização. Inicialmente, é aconselhável começar com componentes rígidos e adicionar flexibilidade conforme necessário, equilibrando as necessidades do projeto. Os componentes flexíveis são mais adequados

para atender a diversas necessidades, mas a complexidade do desenvolvimento deve ser considerada ao tomar essa decisão [Aurelio 2020].

Fazer alterações nos componentes UI no *back-end* requer atenção para garantir a **compatibilidade do componente** com a versão atual do *front-end*, de acordo com Alberto [Coelho 2023] é importante a comunicação entre *front-end* e *back-end* no desenvolvimento de projetos de tecnologia. A colaboração eficaz entre essas equipes assegura a implementação adequada das funcionalidades do sistema e a integração harmoniosa das partes envolvidas. Alterações incompatíveis podem resultar em erros ou comportamento inesperado do aplicativo, afetando a experiência do usuário. Portanto, manter uma comunicação eficiente entre *front-end* e *back-end* é essencial para garantir maior eficiência, colaboração e evitar retrabalho de um projeto de tecnologia.

Em seu artigo, o autor Pradiptamukherjee [Pradiptamukherjee 2021] cita que os componentes desatualizados e vulneráveis podem criar brechas para invasores executarem códigos maliciosos ou roubarem informações confidenciais. É importante utilizar bibliotecas e *frameworks* confiáveis e atualizados, pois componentes problemáticos podem ser definidos como *software* de terceiros ou plataformas com falhas públicas. A complexidade de codificar funcionalidades como transações, localização e chat leva muitos sites a recorrerem a aplicativos de terceiros, mas a falta de atualização regular pode prejudicar a **segurança do componente**. Componentes com vulnerabilidades conhecidas estão entre as principais ameaças listadas pelo OWASP (*Open Web Application Security Project*) para aplicativos da web, comprometendo a segurança de muitos sites que os utilizam.

4.3.3. Cache

A cache é uma técnica de armazenamento temporário de dados frequentemente acessados, utilizada nas arquiteturas de aplicações modernas, para melhorar a eficiência e a velocidade de acesso a esses dados em um sistema. O foco deste estudo é o *Server Caching*, ou ”*caching* no servidor”, exemplificado na Figura 5.

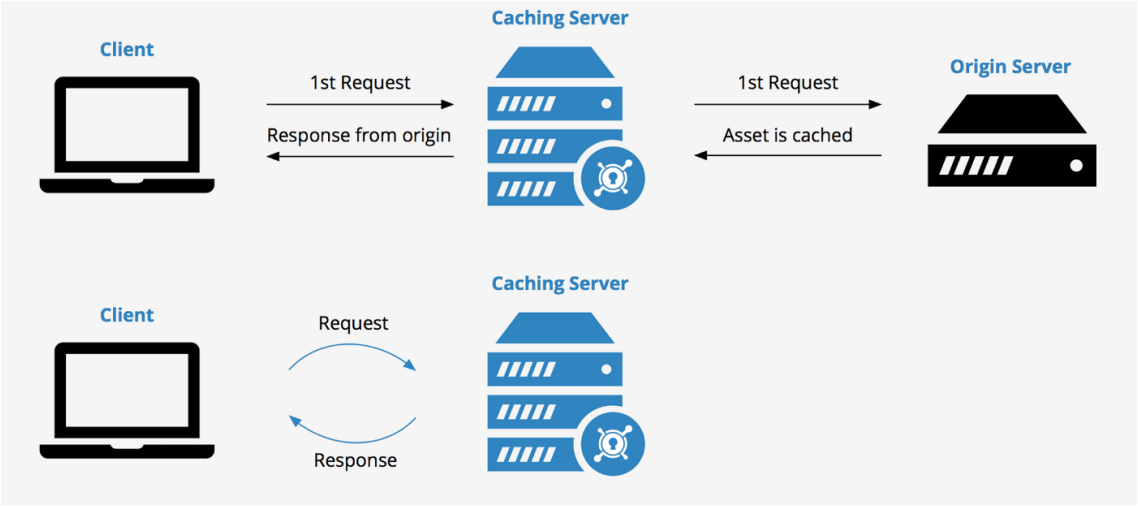

Figura 5. *Server Caching*

Arquitetura	Lacunas	Explicação
Cache	Atualização da cache	Visto que a capacidade de armazenamento do cache é sempre limitada (ao tamanho do disco e/ou da memória) e que as informações podem estar sendo atualizadas a todo o momento, uma grande questão que deve-se considerar é quando e como atualizar as informações contidas na cache.
	Integridade dos dados	O conteúdo é considerado obsoleto e não íntegro quando o tempo que ele está na cache é maior que o tempo definido como "fresco". Em geral, o conteúdo expirado não pode ser usado para responder às solicitações do cliente.

Tabela 3. Lacunas da Cache

Como a arquitetura do SDUI é orientada a servidor, escolher uma estratégia de **atualização de cache** é de suma importância. Uma cache precisa ser atualizada para manter a confiabilidade do sistema, pois assim ela cumpre seu principal papel que é melhorar o desempenho e rapidez no acesso aos dados de forma consistente. Ao selecionar a estratégia apropriada, a utilização da capacidade de armazenamento é otimizada, o atraso na obtenção de dados diminui e a carga sobre os dispositivos também, como o banco de dados. Existem várias estratégias que podem ser utilizadas para atualização da cache de forma eficiente, como *Cache-aside*, *Write-through*, *Write-back* e *Refresh-ahead*. Algumas delas serão detalhadas abaixo:

Na estratégia *Write-through*, conforme o Norman [Jouppi 1993] explica, cada vez que ocorre uma operação de gravação (*write*) nos dados no sistema, a atualização é refletida tanto na cache quanto no armazenamento principal. Alguns pontos-chaves relacionados são:

- Consistência de dados em tempo real onde as gravações são propagadas instantaneamente para o armazenamento em cache e para o armazenamento principal, garantindo que os dados em cache estejam sempre atualizados.
- Garantia de **integridade de dados** onde a estratégia *Write-through* é particularmente útil em cenários onde a consistência de dados é crítica e qualquer discrepância entre os dados em cache e os dados no armazenamento principal pode ser prejudicial.

Já na estratégia de *Write-back*, as operações de escrita são inicialmente registradas na cache e somente depois são propagadas para o armazenamento principal de forma assíncrona. Sua principal característica é adiar a atualização do armazenamento principal, o que pode resultar em um melhor desempenho em operações de gravação, mas também introduz desafios relacionados à consistência de dados. Alguns pontos-chaves relacionados a essa estratégia são:

- As operações de escrita assíncronas, onde as operações de gravação são registradas na cache sem esperar pela confirmação do armazenamento principal. Isso permite um melhor desempenho de gravação, pois as gravações são concluídas na cache de forma rápida.

- Risco de perda de dados temporários devido à natureza assíncrona das operações de gravação na cache. Existe um risco potencial de perda de dados temporários, caso ocorra uma falha antes que as gravações sejam efetivamente propagadas para o armazenamento principal.

É importante notar que a escolha entre *Write-through* e *Write-back*, depende das necessidades específicas do sistema, dos requisitos de desempenho e da tolerância a possíveis atrasos nas gravações.

5. Conclusão

Este trabalho explorou o conceito de *Server Driven UI* e sua arquitetura, afim de encontrar elementos importantes (Serialização, Componentes UI e Cache), suas lacunas e seus possíveis impactos. O SDUI pode oferecer benefícios significativos em termos de flexibilidade e manutenção de interfaces de usuário. As análises e estudos de caso apresentados aqui destacam a relevância do SDUI no contexto atual da tecnologia, onde manter a aplicação sempre atualizada é fundamental para corrigir eventuais *bugs* e manter a segurança do sistema.

Ao longo deste estudo, foram revisados trabalhos relacionados, embora a literatura sobre o SDUI ainda seja relativamente limitada. Os resultados dessa pesquisa mostraram algumas formas de tapar possíveis lacunas, como a importância dos *backups* para manter a integridade dos dados durante uma serialização, criptografia para manter o sigilo e segurança desses dados e escolher um bom formato para transmissão visando o desempenho no processo de serialização.

Nos componentes UI é importante garantir a reutilização dos componentes, obter eficiência na comunicação entre os desenvolvedores *front-end* e *back-end*, para garantir a compatibilidade das versões e utilizar bibliotecas confiáveis e atualizadas.

Em cache demonstrou-se a relevância de se planejar e pensar na atualização da mesma, isso se mostra fundamental para assegurar a consistência e a confiabilidade dos dados da aplicação. Além disso é importante garantir a integridade dos dados escolhendo uma estratégia de atualização de cache que atenda as necessidades específicas do sistema.

Este estudo também concentrou-se na importância da segurança e integridade dos dados no contexto do SDUI, abordando os riscos de segurança associados a essa abordagem e propondo estratégias para mitigá-los, garantindo a integridade e confidencialidade das aplicações SDUI.

Para trabalhos futuros, é interessante continuar a pesquisa e o desenvolvimento explorando a implementação e colocando em prática as soluções abordadas neste estudo. Com mais pesquisas e trabalhos relacionados a SDUI, a compreensão e a experiência prática serão aprimoradas, proporcionando benefícios tanto para os desenvolvedores quanto para os usuários finais.

Referências

Alencar, T. (2019). O que é serialização/serialization? `https://medium.com/@thaisdalencar/o-que-%C3%A9-serializa%C3%A7%C3%A3o-serialization-fc0887bd0970`. Acessado em 12/10/2023.

Aurelio, I. (2020). O que é um componente reutilizável? https://segredo.dev/o-que-e-um-componente-reutilizavel/. Acessado em 23/10/2023.

Brown, K. and Woolf, B. (2016). Implementation patterns for microservices architectures. https://hillside.net/plop/2016/papers/proceedings/papers/brown.pdf. Acessado em 14/09/2023.

Carvalho, G., Papst, N., and Frango, I. (2022). AplicaÇÃo de server driven ui ˜ para interfaces mobile ios de e-commerce. https://adelpha-api.mackenzie.br/server/api/core/bitstreams/296ad9bb-f15b-4632-9325-1a1a81d1274b/content. Acessado em 14/09/2023.

Coelho, A. (2023). A importância da comunicação entre front end e back end na tecnologia. https://awari.com.br/a-importancia-da-comunicacao-entre-front-end-e-back-end-na-tecnologia. Acessado em 31/10/2023.

DICIO. Significado de lacuna. https://www.dicio.com.br/lacuna/. Acessado em 02/11/2023.

Ferreira, U. (2020). Conheça o beagle: A evolução do server-driven ui. https://medium.com/@Uziasf/conhe%C3%A7a-o-beagle-a-evolu%C3%A7%C3%A3o-do-server-driven-ui-2a57cb8d81ff. Acessado em 15/08/2023.

Fileto, R. (2006). Sistemas cliente-servidor. http://www.inf.ufsc.br/˜r.fileto/Disciplinas/BD-Avancado/Aulas/03-ClienteServidor.pdf. Acessado em 03/08/2023.

Hdgson, P. and Echague, P. (2019). Feature flag best practices. https://www.split.io/wp-content/uploads/OReilly_and_Split_Feature_Flag_Best_Practices.pdf. Acessado em 05/10/2023.

Jouppi, N. P. (1993). Cache write policies and performance. https://dl.acm.org/doi/pdf/10.1145/173682.165154. Acessado em 02/11/2023.

Krare, K. (2018). Json vs protocol buffers vs flatbuffers. https://codeburst.io/json-vs-protocol-buffers-vs-flatbuffers-a4247f8bda6f. Acessado em 06/11/2023.

Newman, S. (2015). Pattern: Backends for frontends. https://samnewman.io/patterns/architectural/bff/. Acessado em 03/08/2023.

Pradiptamukherjee (2021). What is components with known vulnerability ? https://www.geeksforgeeks.org/what-is-components-with-known-vulnerability/. Acessado em 23/10/2023.

Rajeev, S. and G.Geetha (2012). Cryptographic hash functions: A review. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=85abc4805adb741b0f8c962794d2ab4dac975c5f. Acessado em 06/11/2023.

Ramos, V. (2020). Entendendo o server driven ui. https://vitor-ramos.medium.com/entendendo-o-server-driven-ui-adb469add630. Acessado em 12/07/2023.

Rucker, S. (2022). What is server-driven ui? https://www.judo.app/blog/server-driven-ui. Acessado em 15/08/2023.

Sampaio, R. (2023). Backend driven content: why we developed a server driven ui framework at nubank. https://building.nubank.com.br/server-driven-ui-framework-at-nubank/. Acessado em 12/07/2023.