UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO CIÊNCIAS DA COMPUTAÇÃO

Fábio Oliveira de Abreu

Framework para gerenciamento de dados client-side com IndexedDB

Florianópolis
2022

Fábio Oliveira de Abreu

Framework para gerenciamento de dados client-side com IndexedDB

Trabalho Conclusão do Curso de Graduação em ciências da computação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para a obtenção do título de Bacharel em Ciência da computação
Orientador: Prof. Dr. Ricardo Pereira e Silva

Florianópolis

2022

Fábio Oliveira de Abreu

Framework para gerenciamento de dados client-side com IndexedDB

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Ciências da Computação, e aprovado em sua forma final pelo Programa de Graduação em Ciências da Computação do Departamento de Informática e Estatística, Centro Tecnológico da Universidade Federal de Santa Catarina.

Florianópolis, 04 de janeiro de 2022.

_____

Prof. Jean Everson Martina, Dr.
Coordenador do Curso

Banca Examinadora:

_____

Prof. Ricardo Pereira e Silva, Dr
Orientador
Universidade Federal de Santa Catarina – UFSC

_____

Prof. Rafael de Santiago, Dr
Avaliador
Universidade Federal de Santa Catarina – UFSC

_____

Prof Frank Augusto Siqueira, Dr
Avaliador
Universidade Federal de Santa Catarina – UFSC

Agradeço primeiro e principalmente a minha família, sem eles nada disso seria possível. Além deles, sou muito grato a meus colegas de curso e de trabalho aos quais sua capacidade sempre me serviu de inspiração. Também agradeço aos professores que acreditaram em mim e ofereceram tanto suporte.

RESUMO

Com a evolução das aplicações web e principalmente a popularização dos PWAs - *Progressive Web Apps*, é cada vez mais necessário o gerenciamento de grandes quantidades de dados dentro dos navegadores. Nesse sentido, o IndexedDB se destaca como uma API, nativa dos navegadores modernos, de baixo nível e alta performance. Entretanto, a manipulação dos dados por meio dela é complexa, exigindo que o desenvolvedor gaste grande quantidade de tempo com conceitos pouco usuais da API do IndexedDB e também de programação assíncrona.

Nesse sentido, esse projeto busca criar uma forma alternativa para que desenvolvedores possam gerenciar esses dados usando o IndexedDb, criando um framework para gerenciamento de dados client-side. Seu objetivo primário é simplificar o uso do IndexedDb, mas o escopo inclui o desenvolvimento de novas características, como validação baseada em tipos, extensibilidade através de herança e composição, e também relações com o banco de dados utilizando o padrão ODM - *Object Document Mapper*.

Dessa forma, novos projetos, ao utilizar a ferramenta proposta, poderão concentrar esforços na criação do software e suas funcionalidades, com garantia de estrutura e integridade dos dados providas pelo framework. Ao final, o framework desenvolvido deverá ser disponibilizado gratuitamente, através do gerenciador de pacotes NPM - *Node package manager*, para uso por desenvolvedores, em seus projetos, mundialmente.

Palavras-chave: PWA - Progressive web application,. API - Application program interface, IndexedDB, Gerenciamento de dados.

ABSTRACT

With the evolution of web applications and the popularization of PWAs - *Progressive Web Apps*, management of large amounts of data inside browsers has become increasingly essential. In that sense, the IndexedDB web API shines as a modern browser native, low level and highly performant solution. However the manipulation of data through this tool is complex, requiring high developer time, and therefore cost, to handle unusual concepts from IndexedDB API, in addition to the asynchronous nature of the instrument, making it efficient for development in its root state.

In that manner, this project aims to create an alternative form, so that developers can manage data with IndexedDb. The framework's main goal is to simplify the use of IndexedDb, but its scope also included the development of new characteristics, such as type based validation, extensibility using composition and inheritance and database relations using the ODM - *Object Document mapper* pattern.

In this way, new projects, when utilizing the proposed framework, should be able to concentrate their effort in the creation of software and its functionalities itself, with assurance of structure and integrity of data provided by the framework. In the end, the developed framework should be freely available, through the package manager NPM - *Node Package Manager*, so that developers can use it, in their projects, worldwide

Keywords: PWA - Progressive web application. API - Application program interface, IndexedDB, Data Management.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

PWA - Progressive Web App

NPM - Node Package Manager

SQL - Structured Query Language

NOSQL- Not only SQL

API - Application Program Interface
JSON - Javascript Object Notation
ODM - Object Document Mapper

ORM - Object Relational Mapper

Ajax - Asynchronous Javascript and XML

XML - Extensible markup language

BD - Banco de Dados

W3C - World Wide Web Consortium

UI - User interface

ECMA - European Computer Manufacturers Association

LISTA DE SÍMBOLOS

SUMÁRIO

1 INTRODUÇÃO

Em 2007 Steve Jobs foi a primeira pessoa a introduzir a ideia de PWA, ainda sem sigla na época, uma aplicação web que se comporte exatamente como aplicativos em celulares ou como aplicações Desktop em computadores. Mais de uma década de evolução da web se passou desde então, e finalmente os PWAs, têm crescido em popularidade, sendo utilizados em aplicações como Uber e Spotify, porém ainda longe de adoção em massa. Nesse sentido, existe o desafio de gerir dados no lado do cliente tanto para diminuir o custo com servidores, quanto para oferecer ao usuário funcionalidades também em modo offline. Curiosamente a tecnologia PWA, no atual momento, tem sido movida primariamente pela Google, rival da antiga empresa de Jobs, a qual recomenda o gerenciamento de dados, de larga escala, nesse tipo de sistema através do IndexedDB, por conta de sua performance e suporte de cerca de 98% dos navegadores usados mundialmente, apesar de sua usabilidade precária. Para evolução ainda maior desse promissor modelo de software, o acesso a ferramentas que simplifiquem o desenvolvimento, é fundamental de forma a atrair equipes e organizações para adotar esse modelo. Assim, esse projeto se propõe a trazer uma ferramenta de gerenciamento poderosa como IndexedDB, porém com uso e integração simples, contribuindo para a propagação tecnológica.

1.1 JUSTIFICATIVA

Em projetos de software é ideal que a atenção dos desenvolvedores esteja voltada da maior maneira possível aos requisitos funcionais, visto que é dessa forma que o cliente final percebe o valor no sistema. Nesse sentido, para o gerenciamento de dados em uma aplicação é esperado pelos desenvolvedores, ferramentas de uso simples de forma que o fluxo de seu recurso mais valioso, o tempo, seja direcionado aos problemas a qual esse software se propõe a solucionar. Dito isso, esse projeto pretende simplificar o uso do indexedDB construindo um framework para gerenciamento de dados oferecendo operações simplificadas mas ainda assim poderosas para ser disponibilizada para uso em projetos diversos sejam eles PWAs ou grandes

projetos web que desejam armazenar dados no lado do cliente em busca de performance, fluidez ou até redução de custos com clusters. Além disso, o framework desenvolvido se propõe a trazer novas soluções para o IndexedDb: integridade de dados através de validação, modelagem e esquematização do banco de dados, o que não acontece nos projeto similares analisados **5.0**, e também extensibilidade dos mesmos por meio de herança e composição. Vale notar, que integridade de dados, garantia que dados são salvos exatamente como intencionado, é essencial para qualquer aplicação e também é um componente fundamental na segurança da informação (Teeling, 2012).

1.2 Método de Pesquisa

1.2.1 Natureza: Aplicada

Criar uma ferramenta livre visando trazer soluções para o mundo do software. Nesse caso, esse trabalho deve reduzir o tempo gasto no desenvolvimento de código para gerenciamento de dados com indexeddb através de técnicas de abstração, trazendo uma ferramenta poderosa para o ecossistema de desenvolvimento de software web.

1.2.2 Abordagem: Qualitativo

O projeto envolve um raciocínio complexo circulando entre indutivo e dedutivo e foca nos futuros utilizadores do framework proposto.

1.2.3 Objetivo exploratório

Visando o entendimento completo do problema, o trabalho vai abordar a evolução das formas de gerenciamento de dados client-side na web bem como avaliar o estado da arte de utilização do indexedDB além de outras possíveis soluções utilizadas atualmente. Assim almejando trazer uma solução completa dentro do contexto proposto.

1.2.4 Procedimentos técnicos: Bibliográficos e estudo de caso

O problema de gerenciar dados em aplicações também é clássico no mundo do software, implementações de ferramentas para armazenar como ORMs e ODMs existem para diferentes tipos de sistemas e necessidades. Dessa forma, busca-se criar uma solução dentro

de um contexto específico, com as devidas adaptações, mas sempre que possível se referenciando em conhecimento já existente.

1.3 OBJETIVOS

Nas seções abaixo estão descritos o objetivo geral e os objetivos específicos deste TCC.

1.3.1 Objetivo Geral

O objetivo do projeto é criar um framework, que abstraia o uso do IndexedDB mantendo sua funcionalidade, performance, facilitando o gerenciamento de dados client-side. Além disso, é fundamental que o framework tenha fácil integração em projetos futuros, assim ele será disponibilizado gratuitamente no NPM - *Node Package Manager*, principal gerenciador de pacotes para esse tipo projeto, dessa forma qualquer desenvolvedor ao redor do mundo poderá começar utilizar o framework desenvolvido com um simples comando de instalação

1.3.2 Objetivos Específicos

1. Criar um framework que abstraia o gerenciamento de dados cliente-side, utilizando o indexedDB.
2. Garantir a integração fácil do framework com demais projetos de Software
3. Disponibilizar o framework final no NPM, gerenciador de pacotes de forma que o projeto esteja livre para uso geral.

2 Fundamentação Teórica

Este capítulo introduz assuntos teóricos importantes dentro do contexto deste trabalho, como frameworks, bancos de dados não relacionais e ODMs. Assim sendo, será feita uma abordagem direcionada trazendo partes desses assuntos que serão relevantes para justificar escolhas de implementação no capítulo posterior de desenvolvimento.

2.1 Bancos de dados não relacionais

Martin Fowler, um dos grandes nomes da engenharia de software e autor do livro *No Sql Essenciais*, argumenta que bancos não relacionais não podem ser definidos e que a melhor forma de entendê-los seria por observar características em comum entre diferentes implementações deste modelo de banco de dados.

Dessa forma, os bancos NoSQL, o que significa “Not only SQL” tem como principal atributo não seguir o modelo relacional tradicional implementado por sistemas de gerenciamento de bancos de dados relacionais, ou seja, os dados são organizados de forma não tabular. Isso implica que bancos não relacionais precisam de outras formas para modelar os dados. Nesse sentido, os modelos mais comuns são modelo de documentos, modelo chave-valor, modelo a partir de grafos e armazenamento wide column. Pode-se dizer que essa variedade de modelos para implementação confirma a ideia de Fowler sobre dificuldade de definição de um NoSQL. A tabela 2.0 e figura 2.0 abaixo ilustram esses modelos mais comuns supracitados.

Organização do Banco de dados	**Descrição**
chave-valor	Um modo simples de estruturar dados em que cada item contém chaves que indexam valores.
documento	Dados são representados por documentos de diferentes formatos sendo atualmente mais comum o formato JSON.
grafos	Organizam dados em Nodos e grafos arestas, funcionam melhor para dados com estruturas de relacionamento complexas
Wide Columns Store	Guardam dados em tabelas, linhas e colunas dinâmicas.

Tabela 2.0 Descrição dos principais tipos de modelos não relacionais a partir de NoSQL Essencial. (FONTE: Adaptado de Fowler, 2013)

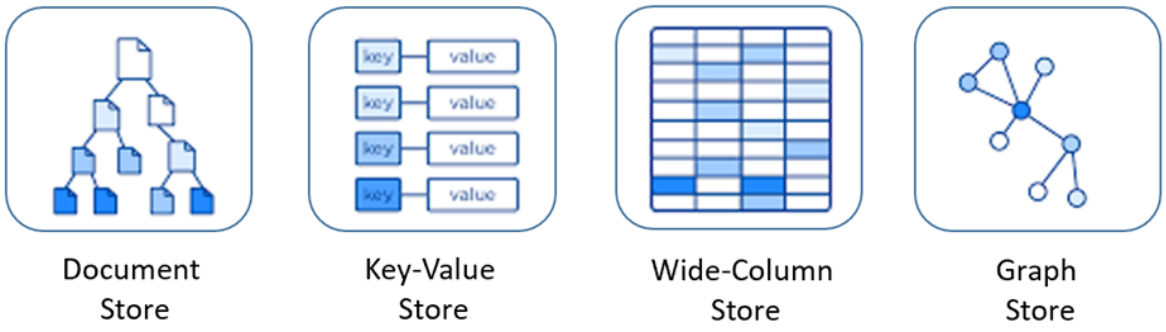

Figura 2.0 Principais tipos de modelos para bancos não relacionais. (FONTE: Docs Microsoft, 2022)

Além disso, é importante notar que existe uma outra dificuldade em definir se o tipo de um BD é chave-valor ou documentos visto que em bancos chave-valor usualmente uma chave pode indexar um objeto de valores enquanto em bancos modelados por documentos usualmente os documento são definidos por uma ID que funciona como chave para o documento. Por essa razão, Fowler costuma se referir a ambos em conjunto como tipo agregado.

Outra característica importante dos bancos NoSQL é o fato de serem "cluster-friendly" de forma que em geral esse tipo de banco deve escalar melhor, se

comparado a bancos relacionais, ao utilizar múltiplos computadores em conjunto em oposto de um único supercomputador para processamento. Por conseguinte, no próximo tópico será abordado o IndexedDB e a razão da escolha por uma arquitetura de dados não relacional dentro dos navegadores.

2.2 FRAMEWORKS

Na indústria de desenvolvimento de software, seja software comercial, público ou open-source a grande maioria dos produtos utilizam de dependências ou bibliotecas, coleções de funcionalidades já prontas, que devem auxiliar no resultado final poupando tempo e consequentemente dinheiro, além de diminuir o tamanho do código fonte, através do reuso de soluções prontas para problemas recorrentes no desenvolvimento de sistemas.

Por si só o uso de bibliotecas já traz enormes benefícios para projetos de engenharia de software, ainda assim a ideia de frameworks traz um paradigma mais extenso. Nesse sentido, um framework deve prover padrões para a organização e desenvolvimento do software, fornecendo assim uma estrutura para o programador final do produto. Contudo, um produto muitas vezes não pode se ater a um único framework de forma monolítica, pois não existe algo específico para todas necessidades do projeto. Por exemplo, um software com longo escopo poderia utilizar 4 frameworks, um para desenvolvimento da interface, outro para gerenciamento de dados, e mais dois para implementação de testes e renderização 3D. Assim cada framework seria responsável por prover ferramentas e estrutura para partes específicas de um projeto maior.

Partindo para uma definição mais formal, segundo artigo sobre processo de desenvolvimento de frameworks publicado por Ricardo Pereira e Silva e Roberto Tom Price, um framework orientado a objetos, como o desenvolvido neste projeto, é uma estrutura de classes inter-relacionadas que constitui uma implementação inacabada, no sentido que aplicações de um certo domínio partem dessa implementação para facilitar e organizar o processo de desenvolvimento (Silva e Price, 1998). Assim, um framework controla a arquitetura e o fluxo de controle de aplicações desenvolvidas por ele. Nesse sentido, a Figura **2.1**, obtida a partir do artigo mencionado, ilustra a interação de um framework e código desenvolvido pelo usuário do framework através do relacionamento entre classes do framework e aplicação.

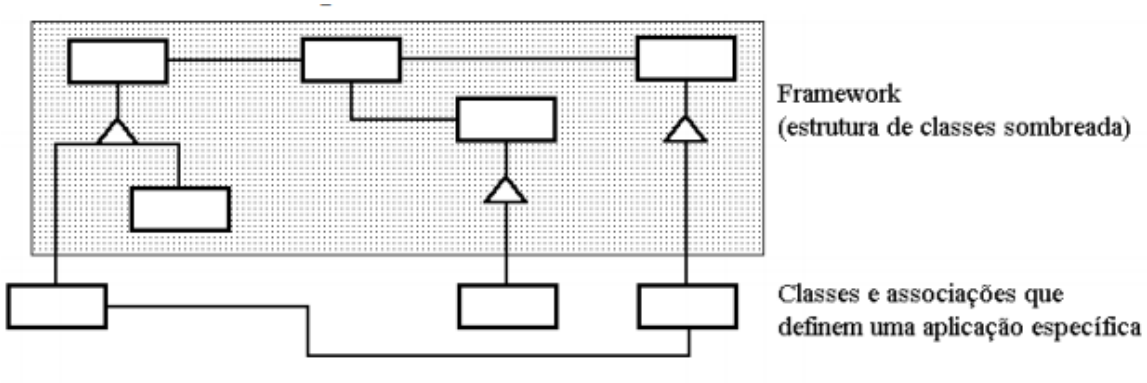

Figura 2.1 Exemplo de aplicação desenvolvida utilizando um framework
(FONTE: Silva e Price, 1998)

2.3 PROGRAMAÇÃO ASSÍNCRONA NO NAVEGADOR

O Javascript linguagem, nativa do navegador, é por natureza uma linguagem *single thread*, executa código linha por linha de forma sequencial, possui apenas uma pilha de chamadas e heap de memória. Essa situação gera um problema grave: é impossível, para o usuário, interagir com a página enquanto essa thread está executando código, de forma que a webpage fica congelada até que a execução termine. Para solucionar esse problema de blocking, as engines dessa linguagem se utilizam de uma técnica de event loop ou loop de eventos. De forma concisa, funções são despachadas da thread principal para esse loop de eventos de forma que a thread principal pode continuar sua execução delegando n funções para esse loop. Quando a thread principal acaba sua execução o loop de eventos processa as funções solicitadas, armazenadas em forma de fila, disparando um evento para cada função finalizada, de maneira que esses eventos retornam a execução para a thread principal que pode delegar novas funções ao event loop e assim sucessivamente ad infinitum. A Figura **2.2** ilustra em alto nível a interação entre a linguagem e o event loop de forma que o programa principal solicita funções de APIs da web como AJAX para realização de requisições na rede e Timeout para espera de tempo assíncrona (não bloqueante).

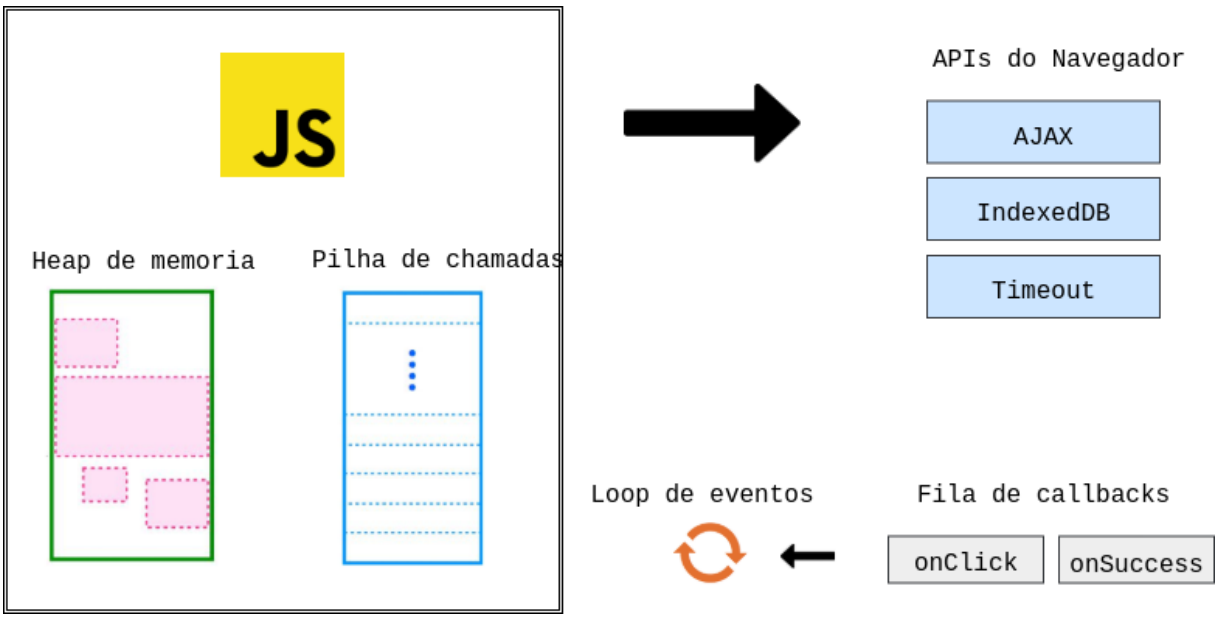

Figura 2.2 Javascript e o event Loop no navegador
(FONTE: Imagem feita pelo autor)

Existem três formas de trabalhar de maneira assíncrona dentro do navegador: a clássica através de callbacks e as mais modernas através de Promises ou async await que não deixa de ser um tipo de promise. Entretanto, para todos esses métodos o funcionamento do event loop e a natureza single thread do javascript continua a mesma.

2.3.1 Callbacks

Callbacks são a maneira clássica de lidar com problemas assíncronos no javascript, na primeira visão parece uma solução bastante eficiente, basicamente é chamado uma função e é passado como argumento da mesma uma segunda função que deve ser chamada ao final da execução da primeira. Como explicado acima, abaixo é ilustrado através da API de timeout uma chamada assíncrona através de callback. No exemplo é passada uma função anônima que pinta uma mensagem e deve ser disparada em 10s de forma que a espera não bloqueia a thread principal.

```
setTimeout(() => {console.log("Isso é assíncrono")}, 10000);
```

Figura 2.3 Exemplo básico de callbacks
(FONTE: Imagem feita pelo autor)

Embora aparente ser um método inicialmente sensato, essa técnica gera um problema chamado callback hell (inferno de callbacks) que acontece quando existem muitas funções que dependem de resultado uma da outra, gerando um código de baixa legibilidade e dificuldades para programadores.

```
a(function (resultadoDeA) {
    b(resultadoDeA, function (resultadoDeB) {
        c(resultadoDeB, function (resultadoDeC) {
            d(resultadoDeC, function (resultadoDeD) {
                e(resultadoDeD, function (resultadoDeE) {
                    f(resultadoDeE, function (resultadoDeF) {
                        console.log(resultadoDeF);
                    });
                })
            });
        })
    })
})
```

Figura 2.4 Ilustração do "callback hell"
(FONTE: Imagem feita pelo autor)

Esse motivo levou os desenvolvedores de aplicações a busca por novas soluções que serão vistas em seguida. Dessa maneira, existe o caso das APIs da web para requisições na rede. Nesse sentido, a API clássica AJAX (Javascript Assíncrono e XML) utilizava exclusivamente o padrão de callbacks enquanto o novo método para requisições Fetch API utiliza somente Promises ou async await que veremos adiante. No contexto desse projeto, o IndexedDb utiliza de callbacks para lidar com eventos assíncronos e essa natureza deve ser explorada, porém o framework final não dará suporte a esse tipo de implementação focando nos padrões mais modernos como promises.

2.3.2 Promises

Segundo MDN Docs uma Promise é um objeto que representa a eventual conclusão ou falha de uma operação assíncrona. Em contramão de uma função que recebe callbacks a serem chamados ao fim da função, funções modernas retornam promises em que podem ser passadas callbacks de sucesso e falha exemplificados na imagem abaixo

```
const promise = busqueAlgo();
promise.then(emSucessoFaca, EmFalhaFaca);
```

Figura 2.5 Exemplo básico de promises
(FONTE: Imagem feita pelo autor)

As promises além de trazer um padrão específico para lidar com problemas assíncronos também têm como grande vantagem uma sintaxe mais amigável que evita o callback hell, no exemplo abaixo é possível observar como o código fica mais limpo com promises encadeadas se comparado a seção interior com callbacks.

```
busqueAlgo()
.then(resposta => editarResposta(resposta))
.then(respostaEditada => salvarResposta(respostaEditada))
.then(foiSalvo => {
  if (foiSalvo) {
     console.log('Alterado e salvo com sucesso')
   }
})
.catch(EmFalhaFaca);
```

Figura 2.6 Ilustração de pipelining com promises
(FONTE: Imagem feita pelo autor)

No exemplo da Figura **2.6** nota-se como promises funcionam através de uma técnica de pipelining de forma que existem vários estágios caracterizados pelo .then(). Cada estágio recebe como resposta o resultado do estado anterior de forma que existe uma relação de

ordem natural e dependência entre eles. Caso aconteça algum problema no pipeline esse será tratado na função catch final. E no cenário em que não exista uma relação de dependência natural entre promessas também é possível utilizar a função promise.all() que retorna uma promessa com o resultado de todas as funções recebidas.

2.3.3 Async Await

Por fim, a técnica mais recente é o uso do async await que é compatível e interoperacional com as promessas vistas no capítulo anterior. Pode-se dizer que o "async await" nada mais é do que um açúcar sintático, construção de sintaxe com finalidade de facilitar a leitura e escrita de código, sobre os objetos de promise.

```
async function obtemDadosEAtualiza() {
    const parametro = preparaParametro();
    const resposta = await promiseParaObterDados(parametro);

    const atualizado = atualizaDados(resposta);

    return atualizado;
}
```

Figura 2.7 Exemplo de função com async await
(FONTE: Imagem feita pelo autor)

Na imagem acima observa-se um exemplo do uso do "async await", primeiramente é declarada uma função do tipo async visto que essa funcionalidade só pode ser utilizada dentro de funções ou métodos de classe. Dessa forma, toda função ou método assíncrono retorna um valor do tipo promise que poderá então ser utilizado através do objeto promise como descrito na seção anterior ou através do próprio async await em outra função assíncrona. Continuando a palavra chave await indica uma espera não bloqueada por um resultado que será resolvida pelo event loop até que a função possa ser continuada.

Considerando as três soluções para programação assíncrona discutidas, dentro da implementação do framework proposto será feito o uso de callbacks quando necessário para interagir com o indexeDDB, porém como a intenção final é facilitar o trabalho de desenvolvedores que utilizam a ferramenta a interface para uso da mesma só dará suporte aos padrões mais modernos de promises e async await. Ou seja, o uso de callbacks será abstraído

do utilizador do framework dessa forma a possibilidade de interagir com o indexedDb somente via promises e async await será um dos atrativos do projeto desenvolvido.

2.4 ODMs

O design pattern Data Mapper foi definido por Martin Fowler, no livro Padrões de Arquitetura de Aplicações Corporativas (Fowler, 2003), como uma camada de mapeadores que movem dados entre objetos e um banco de dados de forma independente enquanto os mantêm independentes um do outro e também do mapeador em si,

Nas imagens abaixo, obtidas a partir do livro de Fowler mencionado, pode-se observar respectivamente o relacionamento entre banco / Mapper / Objeto e o exemplo da atualização de um dado no banco no padrão Data Mapper através de diagrama de sequência.

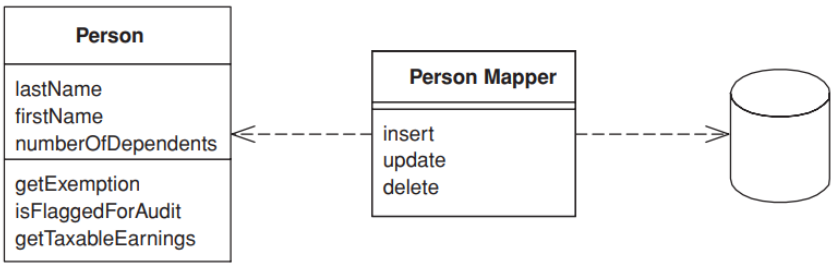

Figura 2.8 Exemplo de relacionamento entre banco de dados Mapper e objeto na memória principal. (FONTE: Padrões de arquitetura de aplicações corporativas, 2003)

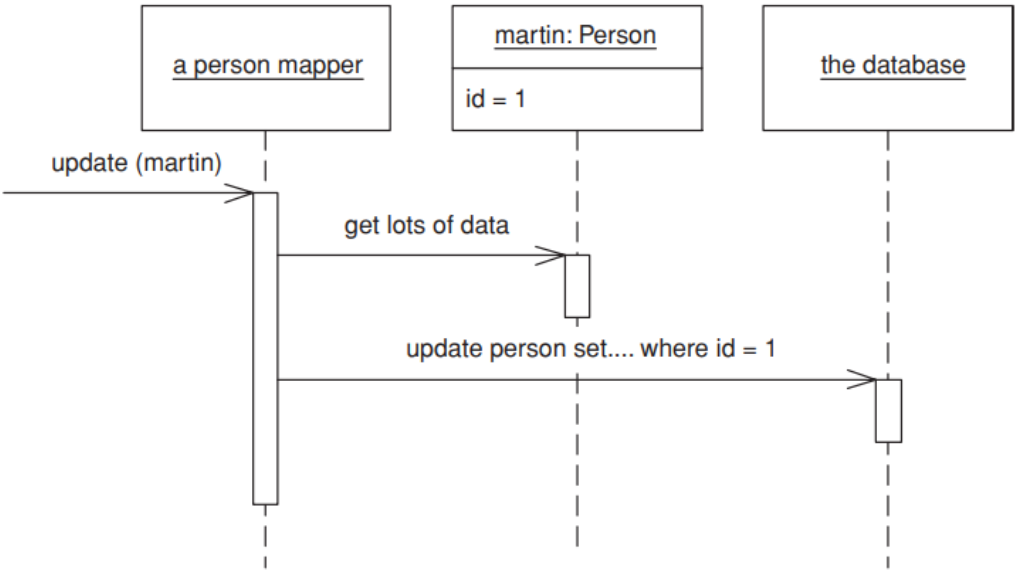

Figura 2.9 Exemplo de uso do padrão Data Mapper para atualizar dados em banco relacional

(FONTE: Padrões de arquitetura de aplicações corporativas, 2003)

Nesse sentido, um ODM (*Object Document Mapper)* nada mais é que uma implementação do conceito de *Data Mapper*, sobre o conceito de gerenciamento de dados dentro de um banco não relacional do tipo agregado (chave-valor ou documento), trazendo algumas funcionalidades extras relevantes dentro desse contexto. Assim, uma ODM provê armazenamento, remoção e atualização de dados em memória e no BD. Suporte para coleções e associações entre documentos, cascade para persistência na remoção de dados, unidade de trabalho de forma que mudanças possam ser executadas em uma única chamada de forma a otimizar o uso de recursos.

3 CONTEXTO TECNOLÓGICO

Essa seção descreve parte do contexto tecnológico em que o framework desenvolvido está inserido. Nesse sentido, passando por modos de aplicação que podem se beneficiar do trabalho desenvolvido como PWA, além de tecnologias utilizadas, por exemplo, linguagem de programação e empacotador de módulos, finalizando a discussão com o NPM, distribuidor de pacotes em que o framework desenvolvido estará disponível.

3.1 PWAs

Segundo MDN web docs (MDN Web docs, 2022), website oficial da mozilla para desenvolvimento e padrões de projeto na web, PWAs possuem inúmeras vantagens do ponto de vista usuário incluindo serem instalados, progressivamente aprimorados, responsivos, linkáveis, descobrível, independente a rede, e seguros.

PWAs são definidos em Microsoft Docs (Microsoft docs 4, 2022) da seguinte maneira, Os Aplicativos Web Progressivos não se limitam a exibir sites em telas móveis, os PWAs são baseados em tecnologias web padrão, entre navegadores, que permitem que eles sejam instalados e executados em vários ambientes diferentes. Simplificando essa definição um PWA é uma aplicação adaptável capaz de ser usado em diferentes plataformas, com ou sem conexão à internet utilizando um navegador como base para execução, podendo ser instalado de forma similar a uma aplicação nativa ou encontrado na web em formato de website. Dessa forma um usuário poderia por exemplo acessar a mesma aplicação no celular ou desktop sendo que essa mesma aplicação pode ser disponível em loja de aplicativos como Microsoft Store ou Google Play ou como um link para ser acessado no navegador.

Também é importante considerar que em termos de desenvolvimento a diferença de um PWA e uma aplicação web tradicional está na presença do *service worker* e arquivo manifest.json. O arquivo manifesto é um documento no formato JSON, que deve ser referenciado no código HTML da aplicação. Esse arquivo indica que a página pode ser instalada no sistema operacional e traz informações como nome do aplicativo, imagem utilizada como ícone e cores de tema. Já os *service workers* oferecem novas funcionalidades para o desenvolvedor (IRIANTO 2021), como funcionar como um proxy entre rede e

navegador, serve de cache para respostas da rede e também traz capacidade de lidar com notificações push. Vale notar que os *service workers* trabalham em uma thread separada a principal do javascript, funcionando assim de forma não bloqueante, porém não tem acesso ao DOM **(seção 3.3)** e demais APIs do navegador.

No contexto desse trabalho o conceito de PWA é uma grande motivação e caso de uso para o framework desenvolvido.

3.2 LINGUAGEM TYPESCRIPT

Mais de 20 anos após sua introdução, o Javascript é hoje uma das linguagem-cross plataforma mais difundida já criadas. Começando como uma pequena linguagem de script para adicionar interatividade trivial a páginas web. Após muita evolução, atualmente mantida pela companhia ECMA International, a linguagem hoje é padrão nos navegadores e é usada extensivamente no desenvolvimento de aplicações web tanto em front end como no backend. Apesar de toda evolução manter grandes projetos escaláveis em javascript continua em desafio, nesse sentido foi desenvolvida a linguagem typescript pela Microsoft um superconjunto de javascript, de forma que todo javascript válido é também typescript válido. O objetivo do typescript segundo a própria documentação é ser um checador de tipos estático ou em outras palavras uma ferramenta que antes que seu código rode verifique a corretude da tipagem do programa. O código escrito em typescript com tipagem forte é convertido para javascript e assim executa no navegador.

```
type validatedInputString = string & { __brand: "hasValidation" };
const validateUserInput = (input: string) => {
  const simpleValidatedInput = input.replace(/\</g, "<");
  return simpleValidatedInput as ValidatedInputString;
};
const hasLargeName = (name: ValidatedInputString): boolean => {
  return name.length < 10;
};
```

Figura 3.0 Exemplo de código em typescript
(FONTE: Imagem feita pelo autor)

```
"use strict"
const validateUserInput = (input) => {
  const simpleValidatedInput = input.replace(/\</g, "<");
  return  simpleValidatedInput
}
const hasLargeName = (name) => {
  return name.length <  10;
}
```

Figura 3.1 Resultado em javascript da tradução do código da figura anterior
(FONTE: Imagem feita pelo autor)

Para um projeto como esse que trata de gerenciamento de dados, a tipagem será uma grande vantagem pois vamos prover uma interface para que programadores criem seus tipos de dados no IndexedDB. Vale notar que a linguagem de desenvolvimento não irá afetar o uso da ferramenta posteriormente pela grande comunidade javascript na web, grandes frameworks como Vue e Svelte são desenvolvidos com typescript e ainda assim incluídos em diversos projetos que usam apenas javascript base.

3.3 DOM E O NAVEGADOR

Em navegadores modernos as páginas web são estruturadas através documentos no formato HTML5 (*Hypertext Markup Language*) e estilizados através de CSS3 (*Cascading Style Sheets*), sendo o javascript a linguagem de programação executável dentro do navegador capaz de trazer interatividade para as páginas além de interagir com a estrutura HTML5 e CSS de um documento. Nesse sentido, essa cooperação entre estrutura da página e linguagem de programação acontece através do DOM (*Document Object Model*) que é, em síntese, uma uma representação orientada a objetos da página da web, que organiza a página web formando uma estrutura de dados do tipo árvore. Ele também provê uma série de funções para acessar e manipular o documento web programaticamente. No contexto do Indexeddb essa API da web funciona de forma coordenada com a DOM, sendo também programável através de Javascript, como a API do IndexedDb funciona de forma assíncrona através de transações,

eventos do DOM são utilizados para notificar o término, sucesso ou possível falha de uma operação realizada.

3.4 ROLLUP

Em síntese, o Rollup é um empacotador de módulos estáticos para aplicações modernas.

Internamente é capaz de montar um grafo de dependências a partir de múltiplos pontos de entrada e então combinar os módulos que compõem um projeto em um ou mais pacotes, que são arquivos estáticos em que o conteúdo do projeto pode ser servido.

Por padrão o Rollup é capaz de entender arquivos Javascript CSS, JSON, HTML além de Typescript. Porém muitos outros tipos de arquivos podem ser interpretados pelo Rollup incluídos no grafo de dependência e por consequência na aplicação desenvolvida. Para isso, utilizamos loaders de forma em que é possível adicionar capacidade ao empacotador de trabalhar em novos formatos. Contudo, nesse projeto o uso de loaders não será necessário, ainda assim será utilizado um plugin para extensão das funcionalidades do Rollup, esse plugin é chamado vite-plugin-dts e tem como função adicionar arquivos com informações sobre tipagem ao pacote final, dessa forma mesmo aplicações javascript que utilizem o framework podem se beneficiar da tipagem estática definida no código fonte escrito em typescript. Esses arquivos para tipagem criados pelo empacotador possuem extensão .d.ts.

3.5 NPM

O NPM - *Node Package Manager* é um repositório online para publicação de projetos *open source*, além disso também é uma ferramenta de linha de comando para interagir com esse repositório que auxilia com instalação, manutenção de dependência e também controle de versão. Hoje o NPM tem o maior número de software registrados se comparado com projetos similares, sendo mais de 1.3 milhão de pacotes e esse número cresce todo dia. Vale notar que o uso do NPM é gratuito e nele é possível fazer download de pacotes de software sem necessidade de autenticação. Dito isso, e considerando também que o npm é utilizado em projetos relacionados a web, considera-se que o projeto desenvolvido neste TCC ao ser

adicionado neste gerenciador de pacotes estará em um espaço em que poderá ser bem aproveitado pela comunidade de desenvolvedores

3.6 Considerações sobre o contexto tecnológico

As tecnologias listadas são parte essencial para o desenvolvimento do framework proposto. Nesse sentido, a linguagem typescript além de fornecer ferramentas para desenvolvimento também vai prover estrutura na forma de tipagem forte para o projeto, mesmo que ele seja importado em um projeto javascript puro. Os eventos assíncronos do DOM serão utilizados para que o framework internamente interaja com IndexedDB de forma que abstraia esses eventos do usuário desenvolvedor. O NPM irá oferecer fácil acesso a milhões de pessoas e usuários para o pacote com framework desenvolvido, enquanto Rollup gerará de fato esse pacote disponibilizado, de forma que todas as suas dependências internas estão auto-contidas e o projeto esteja minificado, ocupando um tamanho de memória mínimo. Vale notar que o bundle gerado pelo Rollup não é de uso exclusivo do NPM, de forma que o bundle pode ser compartilhado de diversas outras formas que não dependam desse serviço. Além disso, o framework deve se encaixar naturalmente no desenvolvimento de PWAs, no sentido de que solucione problemas para o desenvolvimento desse tipo de aplicação, embora existam outros casos para uso do framework.

4 ARMAZENAMENTO DE DADOS NA WEB

Esta seção faz um levantamento tanto histórico quanto atual de métodos utilizados para armazenar dados na web, dentro do navegador, por aplicações, passando por maneiras já descontinuadas como o padrão websql chegando até ferramentas atuais como Cache API, web storage e IndexedDB.

4.1 COOKIES

Cookies web foram a primeira forma de armazenar dados cliente-side no navegador. Embora aceitável originalmente na história da web, por falta de alternativas, essa técnica é hoje ultrapassada. Cookies armazenados são enviados entre o lado do navegador e do servidor em todas as requisições podendo causar atrasos nas conexões de rede. Dessa forma, para o armazenamento de dados de forma geral existem soluções mais modernas dentro do navegador como web storage e o IndexedDB. Ainda assim, os cookies continuam e continuarão a ser suportados nos navegadores, pois apesar de ineficazes para armazenamento geral de dados, os cookies possuem outras áreas que são eficientes e muito úteis para aplicações web. Nesse sentido, uma dessas áreas é o gerenciamento de sessão, pois cookies lidam bem com dados que devem ser compartilhados entre cliente e servidor como informações de login ou um carrinho de compras. Além disso, cookies são usados para rastreamento de usuários, de forma que é possível registrar e analisar o comportamento de um cliente.

4.2 WEBSQL

Com nome bastante auto explicativo, o websql foi a primeira grande tentativa de trazer um padrão para armazenamento de volumes de dados ao browser. A ideia básica foi fazer o uso do SQLite, uma biblioteca pequena, leve, rápida e escalável desenvolvida em C e engine de SQL (HICKSON 2010) com funcionalidade completa, e a partir disso desenvolver padrões para os navegadores e desenvolvedores web. A ideia teve certa popularidade sendo

ainda em 2022 suportada pelas versões mais recentes do Chrome e Opera, ainda assim o web sql nunca teve suporte completo em navegadores importantes no mercado como Firefox e Safari e hoje não é mais mantido, foi descontinuado. Desse modo, as primeiras preocupações com o uso do web sql no navegador, que levaram a seu descontinuamente, estão documentadas no site da W3C (VUKICEVIC, 2009), e foram feitas por Vladimir Vukicevic em 2009, na época desenvolvedor no grupo Mozila. O principal argumento de Vladimir para que a web deixasse o web sql foi que não existe um padrão de SQL, embora o SQL-92 seja geralmente utilizado como base, cada implementação faz alterações a ele criando diversos dialetos de SQL, dessa forma conteúdo para web criando com SQL Lite como base dificilmente funcionaria com qualquer outro SQL. Dessa forma, ao adotar o websql o padrão da web seria "fazer o que o SQL Lite" o que não é padrão suficiente e causo que futuras mudanças de sintaxe em versões do SQL Lite poderiam acarretar em incompatibilidades fazendo com que todos os navegadores precisam sempre se atualizar conforme. Assim sendo, o web sql foi descontinuado em novembro de 2010, seu uso não é recomendado podendo perder suporte em futuras versões de navegadores em que ainda é suportado em 2022 como o Chrome, Opera e Edge. Todavia apesar de estar há mais de 10 anos descontinuado, por possuir suporte no navegador da Google dominante na web e com cerca de 70% de market share, o web sql continua em 2022 utilizado por diversas aplicações e ferramentas de gerenciamento de dados dentro da web.

4.3 WEB STORAGE

O web storage é uma API da web que possibilita o armazenamento de dados no cliente através de pares chave/valor. Ao contrário dos cookies, os dados no web storage não são trocados entre cliente e servidor a toda requisição preservando a performance e também garantindo mais segurança na rede. Além disso, outra vantagem se comparado a cookies é a possibilidade de armazenar mais dados sendo o limite mínimo em navegadores 5 megabytes. Dentro do navegador o webstorage é dividido em duas partes com propósitos diferentes: a *f* e o *local storage*. Nesse sentido, a principal diferença entre os dois tipos de webstorage é que no *local storage* os dados são mantidos mesmo após o fim da sessão, ou seja, o navegador pode ser fechado e os dados são preservados para uso futuro enquanto no session storage assim que o sessão encerra os dados serão automaticamente apagados do navegador. Dessa

forma, apesar dessa diferença de funcionamento ambos local e session storage funcionam da mesma forma bastando mudar o prefixo para indicar qual deles está sendo utilizado, assim a imagem abaixo ilustra o uso do web storage através do *local storage.*

```
localStorage.username = 'Fábio';
localStorage.getItem('usename');
localStorage.setItem('username', 'Fábio');
localStorage.removeItem('username');
```

Figura 4.0 Exemplificando o localStorage
(FONTE: Imagem feita pelo autor)

Como pode ser visto, a api do webstorage é bem simples para guardar pares chaves/valor, sendo muito útil para aplicações que precisem guardar poucos dados no navegador. Contudo o web storage possui duas grandes desvantagens (Ali Alabbas, 2018) a primeira é a incapacidade de salvar estruturas de dados complexas como objetos e listas, sendo isso somente possível caso objetos sejam decodificados e codificados ao inserir e remover itens, como observado na imagem abaixo, causando piora significativa na performance.

```
localStorage.setItem('user', JSON.stringfy({name: 'Fábio'}));
JSON.parse(localStorage.getItem('user'));
```

Figura 4.1 Salvando objetos no localStorage
(FONTE: Imagem feita pelo autor)

Outrossim, ao contrário do indexedDB ou cache API o web storage possui funciona de maneira síncrona, dessa forma quando o mesmo realiza operações bloqueia a thread principal da página no navegador, podendo causar travamentos inconvenientes ao usuário final. Em suma, para aplicações que necessitem de maiores dimensões de armazenamento ou até mesmo que tratem de poucos dados porém realizem acesso com alta frequência o web storage não é satisfatório (Ali Alabbas, 2018).

4.4 CACHE API

O Cache API é a mais recente ferramenta para armazenamento implementada em navegadores modernos. Podendo ser usado quando não há conexão à internet ou também para desafogar a rede em caso de conexões instáveis, a API guarda requisições e suas respostas no navegador diminuindo a necessidade por novas transações na rede. Além disso, embora também envolva requisições assíncronas, para garantir maior performance, sua usabilidade é muito mais trivial e de alto nível se comparado ao IndexedDB sendo composta por operações simples como adicionar uma entrada na cache, adicionar múltiplas entradas na cache, buscar uma resposta na cache e deletar uma entrada na cache como é exemplificado na imagem abaixo.

```
cache.add(new Request('/alguns-dados.json'));
cache.addAll(['/alguns-dados.json', '/mais-dados.json']);
const response = await cache.match(request);
cache.delete(request);
```

Figura 4.2 Exemplificando o cache API
(FONTE: Imagem feita pelo autor)

Entretanto o Cache API com sua maior simplicidade não substitui o Indexeddb. longe disso a simplicidade do Cache API se deve a especialização dessa funcionalidade, sendo muito bom para o que se propõem ser uma cache para armazenar recursos estáticos como o HTML ou Javascript de uma página até o resultado de alguns endpoints que trazem dados do servidor. Porém para o armazenamento e manipulação de volumes gerais de dados o indexedDB continua necessário para aplicações sendo muito mais flexível com sua API de baixo nível.

4.5 INDEXEDDB

A API mais moderna da web para armazenamento de dados IndexedDB foi pela primeira vez disponibilizada nos navegadores em julho de 2010 no lançamento do Firefox versão 4 pela Fundação Mozilla (RANGANATHAN, 2010). Sendo assim, com o tempo, adotado pelos demais navegadores do mercado, até que em 2015 o “Indexed Database API” foi recomendado oficialmente pela W3C em sua primeira versão e desde 2018 tem sua versão atual 2.0 também recomendada pelo consórcio (Ali Alabbas, 2018). Também segundo a W3C,

enquanto o webstorage (**4.3**) também é considerado um padrão moderno ele não dá suporte para recuperação de chaves de forma ordenada, busca de valores de forma eficiente, salvamento de valores com chave compartilhada e indexação de dados (Ali Alabbas, 2018), justificando assim o uso do IndexedDB,. Outro fator importante além dos citados pela W3C é a característica assíncrona não bloqueante da thread principal que não é presente no webstorage.

A construção do indexedDB como uma API de baixo nível faz com que seu uso na forma crua esteja longe do ideal para o desenvolvedor de aplicações, de forma que o uso de ferramentas como as listadas no capítulo **5.0** sejam essenciais. Nesse sentido, para entender essa API e como é possível melhorar o uso da ferramenta serão analisados alguns trechos de código, adaptados de documento da W3C, para gerenciamento de dados com IndexedDB versão 2.0.

```
const request = indexedDB.open("library-database");
let db;

request.onupgradeneeded = function() {
  // Inicialmente o banco não existe, são criados objectStores e índices
  const db = request.result;
  const store = db.createObjectStore("books", {keyPath: "isbn"});
  const titleIndex = store.createIndex("by_title", "title", {unique: true});
  const authorIndex = store.createIndex("by_author", "author");

  // Dados do banco são populados.
  store.put({title: "Quarry Memories", author: "Fred", isbn: 123456});
  store.put({title: "Water Buffaloes", author: "Fred", isbn: 234567});
  store.put({title: "Bedrock Nights", author: "Barney", isbn: 345678});
};

request.onsuccess = function() {
  db = request.result;
};
```

Figura 4.3 Iniciando e populando banco com IndexedDB
(FONTE: Adaptado de Indexed Database API 2.0 por W3C , 2018)

Nesse primeiro exemplo da Figura **4.3**, vemos o primeiro exemplo de código utilizando o IndexedDB, no qual é aberto um banco “library-database”. O acesso a esse é banco acontece pela variável “db” definida na segunda linha e inicializada dentro de callbacks **2.3.1**, de modo que para existir acesso ao banco na linha principal do programa é preciso fazer uma quebra de escopo na função “request.onsucess” que modifica o valor de uma variável declarada fora dela.

Ainda dentro de "request.onupgradeneeded", função chamada sempre quando um banco de dados com número de versão maior que o existente é carregado, é criada uma objectStore para guardar dados no banco. Também são criados dois índices para título e autor para otimizar buscas nesta objectStore. Além disso, por padrão o indexedDB cria um terceiro índice automaticamente para o keyPath, configurado como “isbn”, que serve como chave principal da objectStore, tendo como característica ser um propriedade do tipo read-only, somente leitura. Por fim, dentro dessa função, são adicionados alguns itens, no caso do exemplo livros, na objectStore. Uma observação importante, dentro do exemplo, é que não existe validação ou estruturação dos dados que são inseridos na objectStore, de forma em que é possível adicionar um livro sem autor ou título e também criar propriedades novas em qualquer objeto salvo.

Vale notar que tanto “onsuccess” e “onupgradeneeded” são chamadas assincronamente de forma não bloqueante.

```
const tx = db.transaction("books", "readonly");
const store = tx.objectStore("books");
const index = store.index("by_author");

const request = index.openCursor(IDBKeyRange.only("Fred"));
request.onsuccess = function() {
  const cursor = request.result;
  if (cursor) {
    // Função é chamada para cada item com match author = ‘Fred’
    report(cursor.value.isbn, cursor.value.title, cursor.value.author);
    cursor.continue();
  } else {
    // Não existem mais itens com match.
    report(null);
  }
};
```

Figura 4.4 Exemplificando uma transação no indexedDB
(FONTE: Adaptado de Indexed Database API 2.0 por W3C , 2018)

Nessa segunda parte de código, visualizada na Figura 4,4, é criado uma transação em modo leitura sobre a objectStore “books” do exemplo anterior. Essa transação utiliza um “cursor” para aproveitar a otimização proveniente da indexação por autor, definida também no exemplo anterior. Nesse sentido, a requisição para o IndexedDB busca autores com nome “Fred” e a função passada como callback em “request.onsucess” será chamada para cada autor encontrado com esse nome, logo se existem 20 autores esse nome a função é chamada 20 vezes sempre de forma não bloqueante. Pode-se notar nesse exemplo que embora robusta e eficiente a implementação do IndexedDB não resulta em código simples.

Esse capítulo descreveu as alternativas utilizadas para armazenamento de dados na web, mostrando soluções atuais como webstorage e indexedDB e também depreciadas, para fundamentação histórica, como websql e cookies (utilizado ainda para autenticação). Nesse sentido, ao desenvolver uma nova tecnologia é importante entender não só o estado da arte atual mas também o contexto que levou ao desenvolvimento do mesmo. Ademais, o capítulo também se aprofundou no indexedDB e razões para utilizá-lo com auxílio de ferramentas, este tópico ainda será expandido nas seções de projetos similares e desenvolvimento.

5 PROJETOS SIMILARES

O quinto capítulo faz uma análise de projetos similares ao proposto neste trabalho, esses projetos serão primeiramente descritos além de serem levantadas possíveis vantagens e desvantagens entre eles e o proposto. Nesse sentido, foi utilizado a simples premissa de "o projeto em questão busca facilitar o uso do Indexeddb abstraindo parte de sua lógica?". Assim, em caso de resposta positiva para a pergunta anterior o projeto é considerado similar pois possui um objetivo final muito próximo ao do framework desenvolvido. Dessa forma, serão analisadas três bibliotecas open source disponíveis no github e npm, Dexie.js, PouchDB e IDBWrapper em suas seções específicas e por fim são destacadas as diferenças entre o projeto desenvolvido e os trabalhos descritos na seção análise dos trabalhos.

5.1 DEXIE.JS

Segundo a própria documentação o Dexie.js é um wrapper minimalista para IndexedDB, o banco de dados padrão do browser. A ferramenta tem como objetivo melhorar 3 itens principais do IndexedDB, sendo eles evitar a maneira ambígua de tratamento de erros, estender e melhorar o sistema de consultas ao banco de dados, além de diminuir a complexidade do código. Na Figura **5.0** vemos um exemplo da documentação do Dexie em que é possível confirmar esses 3 objetivos, no qual primeiro é criado uma instância de banco "amigos" que guarda valores como nome, idade e identificador. Em seguida é adicionado um item ao banco e realizada uma consulta que deve retornar esse mesmo item, também ao fim vemos que todos os erros serão tratados na função *catch*.

```
var db = new Dexie("BancoDeAmigos");
 db.version(1).stores({
   amigos: "++id,name,age"
 });
 db.amigos.add({name: "Fábio", age: 21}).then(() => {
     return db.amigos.where("age").below(38).toArray();
 }).catch((e) => {
     alert ("Error: " + (e.stack || e));
 });
```

Figura 5.0 Exemplificando do uso do Dexie.JS
(FONTE: Adaptado de Dexie.org, 2022)

Embora em sua versão atual (3.2.1) o Dexie.js seja voltado ao IndexedDB, a ferramenta inicialmente suportava por padrão o websql. Nesse sentido, ainda que hoje esse suporte não exista por padrão, o histórico e escolhas na arquitetura do Dexie deixam um legado que não pode ser deixado de lado e por conseguinte seria possível obter resultados finais diferentes em uma ferramenta 100% voltada ao IndexedDB.

5.2 POUCH DB

Retirado de sua página oficial, o PouchDB é um banco de dados open-source inspirado pelo Apache CouchDB desenhado para rodar eficientemente no browser. Dentro do navegador utiliza o indexedDB como padrão, assim fornecendo uma API simples para uso do mesmo de maneira similar ao Dexie.js observado no tópico anterior. Entretanto, o PouchDB possui alguns diferenciais interessantes e pode-se dizer que é um projeto mais ambicioso que o Dexie dentro de um nicho de necessidade mais específico. Nesse sentido, esses diferenciais são a sincronização com o CouchDB e a possibilidade de utilizar a ferramenta também no backend com Node.js e LevelDB.

Sobre a sincronização com CouchDB, a ferramenta Pouch possibilita o armazenamento de dados de forma local enquanto offline e então sincroniza com CouchDB e servidores compatíveis quando a aplicação está novamente online, mantendo assim os dados de usuário sempre sincronizados. Essa funcionalidade porém não é de uso obrigatório, sendo assim possível utilizar a ferramenta apenas como um wrapper de indexeddb, de maneira similar ao Dexie, considerando que o Pouch possui uma interface tão simples quanto a do primeiro. O segundo diferencial está na possibilidade de usar a mesma interface de armazenamento no navegador e no servidor, pois a API do Pouch oferece suporte para guardar dados no servidor utilizando o LevelDB, um sistema de armazenamento do tipo chave-valor desenvolvido pela Google. Essa ideia é muito interessante pois oferece uma interface idêntica para gerenciamento de dados no servidor e navegador, porém o gerenciamento de dados no servidor foge do escopo desse trabalho que se limita ao IndexedDB em que o uso só é possível dentro do navegador.

```
var db = new PouchDB('amigos');
db.put({
  _id: '901',
  name: Fábio,
  age: 21
});
```

Figura 5.1 Exemplificando o uso do PouchDB adaptado da documentação do próprio (FONTE: Adaptado de pouchdb.com, 2022)

Diferentemente do Dexie.js o PouchDB ainda oferece suporte ao websql (versão 7.2.1), basicamente caso o navegador utilizado não ofereça suporte ao IndexedDB. A ferramenta tenta utilizar o websql no cenário em que o IndexedDb não possui suporte pelo navegador. Embora possa parecer interessante, essa funcionalidade afeta pouquíssimos navegadores, visto o vasto suporte ao IndexedDB discutido na seção **4.5**. Ademais, também é possível configurar o websql como opção principal. Por conseguinte, assim como o Dexie e de maneira até maior, em virtude do sistema de armamento legado websql ainda ser suportado, o Pouch também deixa a possibilidade de que uma ferramenta única para o IndexedDB ofereça vantagens pela especificidade maior.

5.3 IDBWrapper

Com cerca de 66.000 downloads semanais no NPM, o ID Wrapper assim como o Dexie.js tem como objetivo ser um wrapper minimalista para o IndexedDB, nesse sentido é também um projeto com escopo mais limitado que o PouchDB. A principal vantagem dessa biblioteca sobre as demais previamente descritas é que o IDBWrapper foi desde o começo uma ferramenta para o IndexedDB e logo nunca deu suporte a projetos legados para armazenamento de dados na web. Contudo esse projeto se propõe a não mudar extensivamente a interface de uso do IndexedDB, dessa forma para lidar com a natureza assíncrona desse banco de dados o wrapper utiliza callbacks, que tiveram suas desvantagens e alternativas discutidos em **2.3**. Outra desvantagem do mesmo é não possuir possibilidade de estruturação prévia de documentos, ou seja qualquer formato de dado pode ser incluído no banco de forma a possibilitar inconsistências no BD e problemas na organização de código. É importante notar que de certa forma, essas desvantagens podem ser também consideradas

vantagens para aplicações que procuram uma ferramenta realmente minimalista e afinal é essa justamente a proposta do wrapper.

5.4 Análise dos trabalhos

As soluções analisadas sem dúvidas obtiveram sucesso em abstrair complexidade e facilitar o uso e desenvolvimento com o IndexedDB. Dessa forma, os resultados obtidos por esses projetos servem como mais uma inspiração para o projeto de framework. Todavia, existem importantes diferenças que fazem com que o trabalho desenvolvido possa ser uma escolha melhor em determinados contextos. Dentro dessas diferenças está o suporte ao depreciado websql (descrito em **4.2**) pelos dois primeiros projetos mencionados: Dexie e PouchDB. Além disso, nenhum dos três projetos funciona como ODM (caracterizado em **2.4**), provendo mapeamento e validação através de esquemas para organização bem definida de dados. Nesse sentido, os projetos mencionados se limitam a fornecer uma série de funções abstraindo lógica enquanto o trabalho proposto deve funcionar como framework (mais detalhes em **2.2)**, e dessa forma, oferecer estrutura para organização do código de aplicação na parte de gerenciamento de dados, além de possuir extensibilidade através de classes e orientação a objetos.

6 DESENVOLVIMENTO

Nesta seção será abordado o desenvolvimento do projeto proposto em conjunto com o processo criativo para resultar no framework final. Isso envolve uma série de etapas, como arquitetura do framework e seu interação com componentes dentro do navegador, visão geral do diagrama de classes, exposição de funcionalidades de forma simples para o usuário final do framework, geração de erros customizados até finalmente a integração do framework com o gerenciador de pacotes e sua disponibilização ao público final.

6.1 VISÃO GERAL DO FRAMEWORK

O “Typed Indexeddb" nome proposto para o framework projetado, foi desenvolvido para funcionar dentro do navegador. Ele funciona como uma camada entre uma aplicação final do frontend desenvolvida e a API do IndexedDB. Dessa forma, como um ODM a ferramenta desenvolvida mapeia e valida dados obtidos do código da aplicação executando no navegador preservando a integridade do banco de dados proveniente do browser que é acessado pelo framework através do IndexedDB. Sendo assim, a interação entre Indexeddb e navegador ocorre através de eventos assíncronos no DOM como “success”, “error” e “oncreatedb”. de forma que o programador só precise interagir com o Typed Indexeddb.

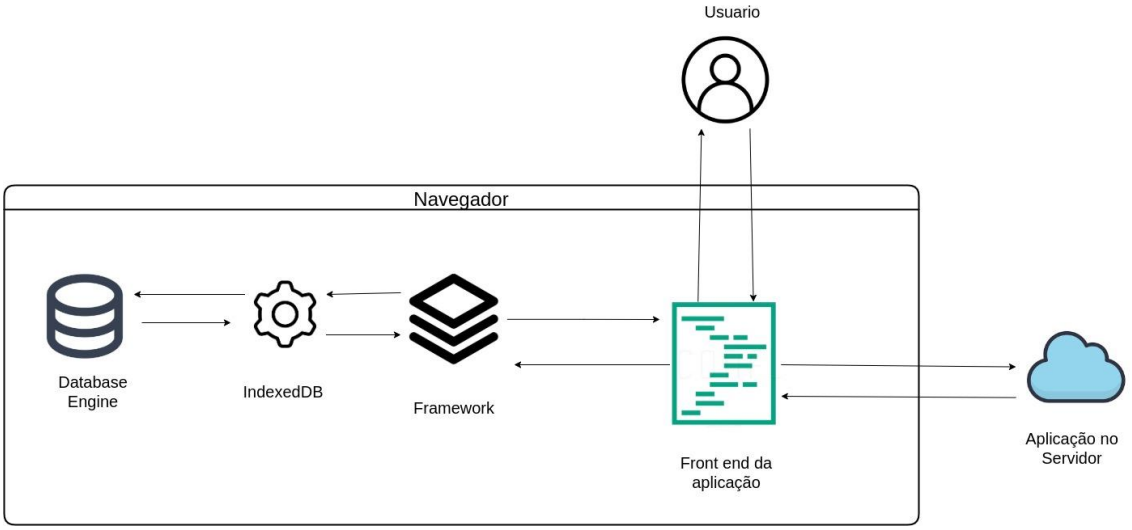

Figura 6.0 O Typed Indexeddb dentro do navegador
(FONTE: Imagem feita pelo autor)

6.2 Esquemas e Modelos

No ponto de vista de quem utiliza o framework existem duas classes principais, sendo que três são expostas de fato ao programador e as demais dentro da modelagem, serão privadas para o funcionamento interno do framework ou então devem servir para configurar o funcionamento geral do próprio banco de dados. Essas duas classes principais são Schema e Model que têm como objetivo modelar e representar os dados do banco. Nesse sentido, a classe Schema modela documentos de forma a refletir a estrutura na qual os mesmos devem ser guardados no banco. Na imagem abaixo vemos como um utilizador do framework pode criar um Schema simples que representa uma pessoa.

```
const schema = new Schema('person', SchemaAttribute.object.shape({
    name: SchemaAttribute.string,
    age: SchemaAttribute.number,
    is_brazilian: SchemaAttribute.boolean.isRequired(),

    date: SchemaAttribute.object.shape({
        default: SchemaAttribute.number,
        time: SchemaAttribute.string,

        limit: SchemaAttribute.object.shape({
            upperBound: SchemaAttribute.number.isRequired(),
            lowerBound: SchemaAttribute.number,
        })

    })
}))
```

Figura 6.1 O Exemplo de criação de um Schema

(FONTE: Imagem feita pelo autor)

Primeiramente pode-se notar que um Schema é composto por SchemaAttributes que definem a tipagem dos itens dentro de um documento, inicialmente como argumento o Schema deve receber um SchemaAttribute do tipo object ou array indicando ou documento ou uma coleção de documentos. A validação dos tipos de um Schema ocorre de forma que um atributo pode ou não ser requerido, o que é definido pela função isRequired de SchemaAttribute. Dessa forma, no exemplo acima atributos não requeridos como "name" e "age" podem não ser passados para um documento sem problemas, porém caso passados devem possuir tipo string e number respectivamente. Por outro lado, atributos requeridos como "is_brazilian" devem obrigatoriamente estar presentes em todo documento criado com esse Schema. Também é primordial que cada Schema receba nome único, passado no primeiro parâmetro, todavia isso não é necessário caso os Schema não compartilhem o mesmo espaço de dados, representado na classe TypedIndexedDbDatabase.

A segunda classe essencial no uso do framework chama-se Model. Um Model representa um item que pode ou não estar no banco de dados e que tem sua estrutura definida por um Schema. Na Figura 6.2 pode ser visto um exemplo de criação de um Model partir do Schema de pessoa da imagem anterior.

```
const personModel = new Model(personSchema).build({
    name: 'Fábio',
    age: 18,
    too: true,
    date: {
        default: 9021,
        limit: {
            upperBound: 9021,
        }
    }
})
```

Figura 6.2 O Exemplo da criação de um Model

(FONTE: Imagem feita pelo autor)

Enquanto um Schema representa uma possível estrutura para um documento que guarda dados de uma pessoa, o Model representa a pessoa em si de acordo com o Schema passado em sua criação. Nesse sentido, a validação dos dados Model é feita a partir do

Schema de forma que cada Schema possa ter inúmeros modelos, assim esse relacionamento entre as duas entidades busca garantir a estrutura e integridade dos dados. Quanto ao Model da imagem pode-se observar que baseado no seu Schema esse seria um modelo válido, todas as suas propriedades conferem com os tipos do Schema e embora dados como “date.time” e “date.limit.lowerBound” não existem nesse modelo ambos não são marcados como required no Schema..

Vale notar que um modelo criado não é salvo automaticamente na base de dados do navegador. Primeiramente, os dados ficam apenas na heap do programa onde podem ser manipulados até que, quando o usuário decide de fato guardar os dados, podem ser utilizadas funções como save. Dessa forma, a necessidade de uso de assincronismo só acontece no momento de salvamento em si.

Além de funcionar como ODM e prover validação aos dados, a classe Model também é aproveitada para trazer novas funcionalidades importantes no contexto de gerenciamento de dados mas que não são suportadas pela API do Indexeddb nativamente, como por exemplo busca por valores similares análogo ao operador “LIKE” do SQL e também buscas com agrupamento.

6.3 EXTENSIBILIDADE

Como explicado em **2.2** uma das características de um framework orientado a objetos é a extensibilidade entre funcionalidades e organização através de classes e objetos que se relacionam entre framework e aplicação. Dessa forma, o typed IndexedDB oferece extensibilidade na modelagem de Schemas a partir da classe SchemaAttribute, nesse sentido além de oferecer a montagem de schemas a partir de tipos primitivos é possível que o desenvolvedor da aplicação crie tipos mais complexos utilizados por diferentes documentos no banco de dados.

```
const dateSchemaAttribute = SchemaAttribute.object.shape({
    default: SchemaAttribute.number,
    time: SchemaAttribute.string.isRequired(),
```

```
    limit: SchemaAttribute.object.shape({
        upperBound: SchemaAttribute.number,
        lowerBound: SchemaAttribute.number,
    })
})
const personSchema = new Schema(SchemaAttribute.object.shape({
    name: SchemaAttribute.string,
    age: SchemaAttribute.number,
    birthday: dateSchemaAttribute,
}))
```

Figura 6.3 Extensão de tipos através de SchemaAttribute

(FONTE: Imagem feita pelo autor)

A Figura **6.3** exemplifica o fenômeno descrito. A classe SchemaAttribute é estendida de forma a criar um atributo para Data que pode então ser utilizado por diversos documentos ao longo da aplicação e assim a modelagem de Data só precisa ser feita uma vez no código fonte, evitando repetições que geram mais linhas de código e possíveis erros.

6.4 MODELAGEM

Um modelo é uma simplificação da realidade de forma que seja facilitado o entendimento de um sistema robusto. Nesse sentido, na Figura **6.4** encontra-se o diagrama de classes do framework desenvolvido. O diagrama de classes tem como função representar a estrutura do projeto desenvolvido e como suas partes interagem. Esse tópico também abordará essas classes e suas interações. Vale notar que Models e Schemas já foram apresentados na seção 6.2, essas duas classes são o core do framework e essenciais para as demais partes do projeto. Assim, pode-se dizer que todo o projeto desenvolvido existe para suportar o uso dessas duas de forma a provisionar uma interface de fácil acesso para programadores ao typed indexedDb.

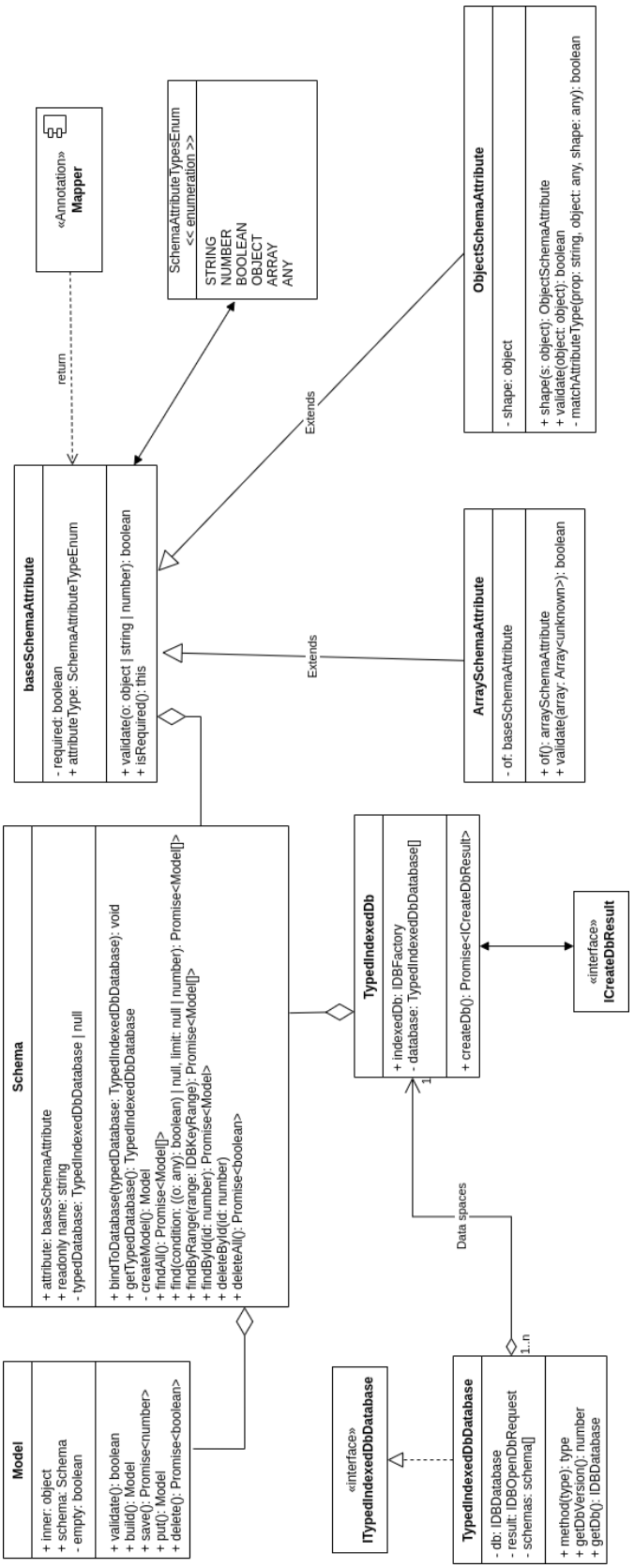

Figura 6.4 Diagrama de classes do framework desenvolvido

(FONTE:Imagem feita pelo autor)

6.4.1 SchemaAttribute e suas classes

Como já visto na seção **6.2** SchemaAttributes são fundamentais para criação de Schemas. Em uma visão geral um SchemaAttribute possui três funções principais: representar a tipagem dos valores guardados no banco, indicar se um atributo é ou não nullable, sendo que um atributo não nullable deve obrigatoriamente possuir um valor, e além disso a terceira função que do ponto de vista do usuário não fica 100% evidente é a validação dos valores do próprio atributo.

Vejamos primeiramente a última funcionalidade citada, ao contrário do que pode-se pensar não existe uma grande função no Schema para validar um modelo. Pelo contrário, tudo que faz um Schema em termos de validação é chamar a função validate no seu primeiro SchemaAttribute. Assim, cada SchemaAttribute fica responsável por se auto validar. Dessa forma, a validação final é verdadeira caso todos os atributos validem com sucesso. Portanto, pode-se pensar que um Schema é composto por uma árvore em que nós são SchemaAttributes com a função de retornar verdadeiro ou falso caso sejam ou não válidos. Para que um SchemaAttribute possa de fato fazer essa validação é preciso que exista uma especialização, visto que podem existir diferentes tipos de atributos, dessa forma existem três tipos de SchemaAttribute, baseSchemaAttribute, ArraySchemaAttribute e ObjectSchemaAttribute.

Na base dos SchemaAttributes está o baseSchemaAttribute. Essa classe é inicializada através de um valor proveniente do Enum SchemaAttributeTypes que indica o tipo do atributo. Nesse sentido, a própria classe baseSchemaAttribute é capaz de ser utilizada para atributos do tipo string, booleanos ou numéricos que pela simplicidade podem ser resolvidos em único método validate dessa classe. Contudo, o mesmo não acontece em arrays ou objetos que necessitam de validações mais completas. Dessa forma, para validação de objetos e arrays foram criadas duas subclasses de baseSchemaAttribute: ObjectSchemaAttribute e ArraySchemaAttribute. Assim sendo, para fins de implementação considera-se que um objeto é válido quando quando todos os seus atributos são válidos, ou seja, cada chave possui um valor do tipo especificado por essa chave, e o formato do objeto com chaves e tipos é dado pela propriedade “shape”. Enquanto isso, uma array é válida se todas as suas entradas são do tipo especificado e o tipo da array é dado pela propriedade “of”. Vale notar que na implementação do framework é completamente possível que um objeto possua outros objetos

dentro dele, ou que um array possua múltiplos arrays, ou mesmo que exista um array de objetos em que cada objeto possui um outro array. Todo esse processo de validação é facilitado por estar contido dentro de um SchemaAttribute, logo ao validar um array, por exemplo, não importa se o array está dentro de um objeto ou outro objeto, tudo que importa para validação é se todos os itens estão de acordo com tipo esperado passado denotado pela variável “of”.

Em suma, um SchemaAttribute complexo, dos tipos array ou objeto é válido se e somente se todos os atributos que ele possui são válidos. Essa validação ocorre de forma individual e como ilustra a Figura 6.5 feita de baixo para cima e da esquerda para direita.

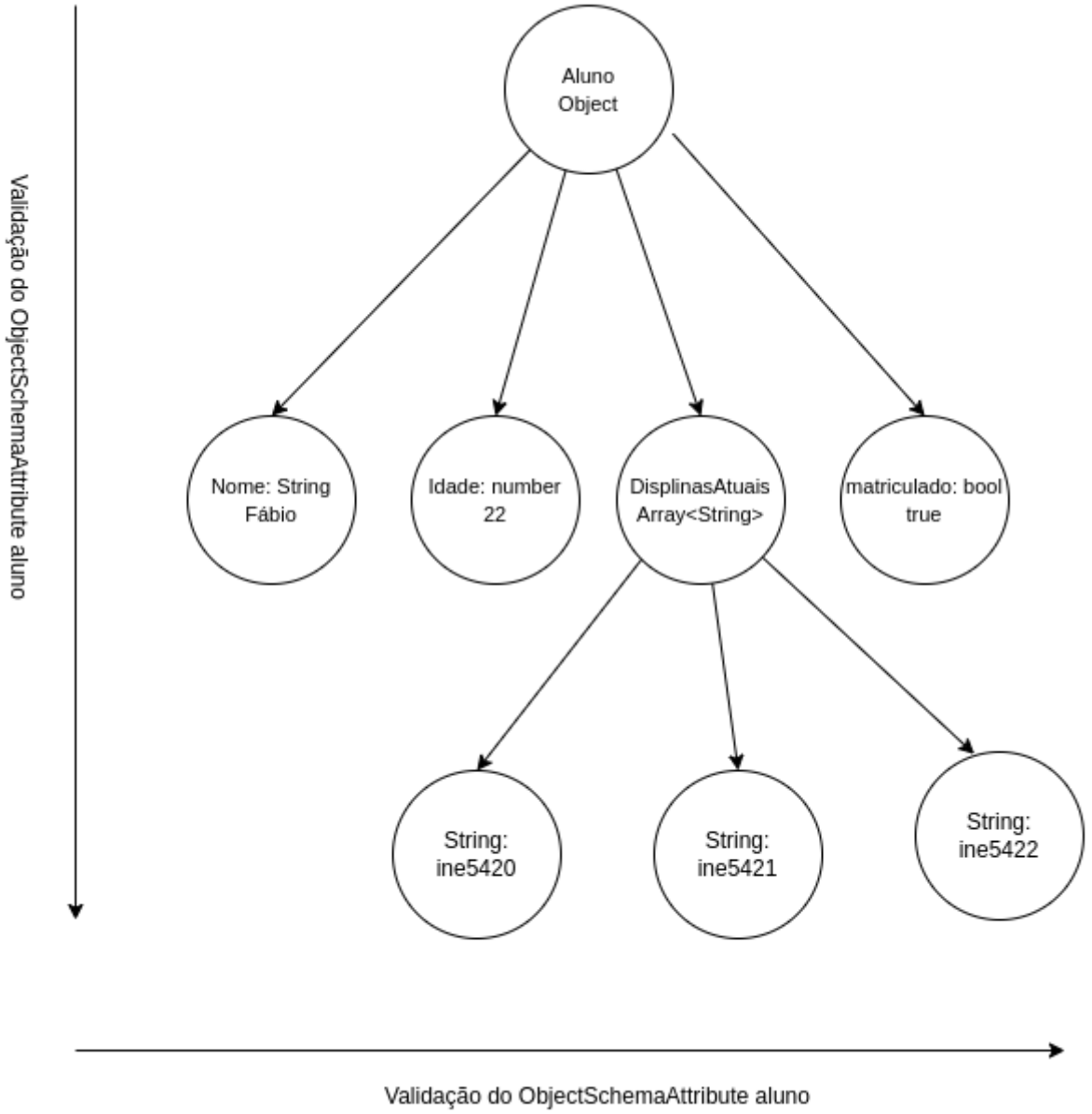

Figura 6.5 Ilustração da validação de um ObjectSchemaAttribute

(FONTE: Imagem feita pelo autor)

A Figura **6.5** representa a validação do SchemaAttribute montado na Figura **6.6**, que como demonstrado na seção de extensibilidade **6.3** poderia ser um Schema em si ou apenas um parte de um Schema maior. Dessa forma a validação dos dados nessa imagem ocorre da seguinte forma { Nome => Idade => DisciplinasAtuais[0] => DisciplinasAtuais[1] => DisciplinasAtuais[2] => matriculado }, assim quando todos os índices de "displinasAtuais" são válidos o atributo disciplinas atuais é considerado válido e quando todos os atributos de Aluno são válido o SchemaAtribute aluno é também válido.

```
SchemaAttribute.object.shape({
  nome: SchemaAttribute.string.isRequired(),
  idade: SchemaAttribute.number,
  disciplinasAtuais:
SchemaAttribute.array.isRequired().of(SchemaAttribute.string),
  matriculado: SchemaAttribute.boolean,
})
```

Figura 6.6 Código para SchemaAttribute ilustrado em 6.5

(FONTE: Imagem feita pelo autor)

6.4.2 Classe Mapper

A classe mapper tem como função simplificar a interface para que o usuário final crie Schemas. Na verdade, do ponto de vista do usuário não existe uma classe mapper; o que acontece é que quando o usuário utiliza um SchemaAttribute na verdade ele está utilizando uma instância de Mapper. A implementação de Mapper é pequena e feita somente em um arquivo demonstrado na imagem posterior.

```
class Mapper {
  get number() {
    return new baseSchemaAttribute(schemaAttributeTypes.NUMBER);
  }

  get string() {
    return new baseSchemaAttribute(schemaAttributeTypes.STRING);
  }

  get boolean() {
    return new baseSchemaAttribute(schemaAttributeTypes.BOOLEAN);
  }

  get any() {
    return new baseSchemaAttribute(schemaAttributeTypes.ANY);
  }

  get array() {
    return new arraySchemaAttribute();
  }

  get object() {
    return new ObjectSchemaAttribute();
  }
}

export default new Mapper();
```

Figura 6.7 Código para classe mapper

(FONTE: Imagem feita pelo autor)

Como mencionado anteriormente, o desenvolvedor usuário do framework utiliza um Mapper pensando que está a utilizar um SchemaAttribute, afinal de contas tudo que um Mapper faz é retornar SchemaAttributes. A vantagem de utilizar Schema Attributes é um código mais limpo tornando a declaração de Schemas muito mais legível e por consequência fáceis de manter. Um dos artifícios da classe Mapper é utilizar *getters*. No javascript funções do tipo *getter* não precisam ser chamadas. Logo, ao utilizar um SchemaAttribute, o resultado é "SchemaAttribute.string" e não SchemaAttribute.string(). Além disso, o Mapper esconde a criação de objetos das diferentes classes de SchemaAttribute (object, array, base) e os parâmetros necessários para inicialização. Em suma, a Figura **6.8** ilustra bem o objetivo dessa classe comparando a declaração de um Schema sem e depois com um Mapper.

```
// Sem mapper
const personSchema2 = new Schema(
  "named",
  new ObjectSchemaAttribute().shape({
    name: new baseSchemaAttribute(schemaAttributeTypes.STRING).isRequired();
    age: new baseSchemaAttribute(schemaAttributeTypes.NUMBER);
  })
)

// Com Mapper
const personSchema = new Schema(
  "named",
  SchemaAttribute.object.shape({
    name: SchemaAttribute.string.isRequired(),
    age: SchemaAttribute.number,
  })
)
```

Figura 6.8 Código para classe mapper

(FONTE: Imagem feita pelo autor)

Nessa imagem pode-se perceber que os objetivos da classe Mapper são realmente atingidos. A segunda implementação do Schema é muito mais enxuta e tem intenções de modelagem muito mais claras e tudo isso ainda ocorre sem que o usuário sequer precise saber da existência de Mapper.

6.4.3 Classes TypedIndexedDb e TypedIndexedDbDatabase

As classes TypedIndexedDb e TypedIndexedDbDatabase se complementam, seu uso essencial para configuração inicial do framework dentro de aplicações. A primeira classe tem como objetivo secundário verificar se o browser utilizado é compatível com IndexedDb, dado que navegadores muito antigos não são suportados. Já o objetivo principal é criar um espaço de dados através da função createDb, observada na imagem abaixo.

```
public createDb(dbName: string, schemas: Schema[]): Promise<ICreateDbResult> {
  const request: IDBOpenDBRequest = indexedDB.open(dbName, 1);

  return new Promise((resolve, reject) => {
    request.onerror = () => reject({ created: false });

    request.onupgradeneeded = () => {
      const db = request.result;

      const stores = db.objectStoreNames

      for (let i = 0; i < stores.length; i++) {
        db.deleteObjectStore(stores.item(i))
      }

      schemas.forEach((s) => {
        const store = db.createObjectStore(s.name, {
          keyPath: "id",
          autoIncrement: true,
        });
      });
    };

    request.onsuccess = () => {
      const db = new TypedIndexedDbDatabase(request.result, request, schemas);
      this.databases.push(db);

      return resolve({
        created: true,
        database: db,
      });
    };
  });
}
```

Figura 6.9 Código para função createDb

(FONTE: Imagem feita pelo autor)

Essa função recebe um nome para esse espaço de dados e uma lista de schemas que serão vinculados a esse espaço, criando objectStores para cada Schema e abstraindo a complexidade do indexeddb e suas callbacks (**2.3.1**). Por fim essa função retorna uma Promise (**2.3.2**) para um objeto da classe TypedIndexedDbDatabase. A partir desse momento a primeira classe não precisa mais ser utilizada e a segunda fica encarregada de gerenciar e também representar esse espaço de dados e para isso tem acesso aos Schemas vinculados.

Vale notar que alguns projetos de maior complexidade, podem desejar utilizar mais de um espaço de dados. Nesse sentido, a função TypedIndexedDb.createDb pode ser chamada outras vezes passando diferentes nomes de identificação para os espaço de dados que devem ser criados além de novas listas de Schemas vinculados a esse espaço. Com isso, cada projeto que utilize o framework deve possuir apenas um instância de TypedIndexedDb, podendo haver multiplas de TypedIndexedDbDatabase, uma para cada espaço de dados desejado. Na seção **6.6.2** (Uso do Typed IndexedDb) será abordado como essas duas classes são utilizadas dentro de uma aplicação.

6.5 Integração com o gerenciador de pacotes

Um dos objetivos iniciais desse projeto foi disponibilizar a aplicação dentro do NPM - Node package manager (seção **3.5**) para que possa ser facilmente utilizado por diversos projetos de software ao redor do mundo. Nesta seção será refletido como foi esse processo.

A primeira questão importante é saber o que de fato está sendo exportado para o usuário da biblioteca. Isso é definido no arquivo main.ts, observado na Figura **6.10**.

```
import TypedIndexedDb from "./TypedIndexedDb";
import Schema from "./Schema/index";
import SchemaAttribute from "./SchemaAttribute/index";
import Model from "./Model/index";

export default {
  TypedIndexedDb,
  Schema,
  SchemaAttribute,
  Model
}

export {
  TypedIndexedDb,
  Schema,
  SchemaAttribute,
  Model
}
```

Figura 6.10 main.ts código exportado pelo pacote

(FONTE: Imagem feita pelo autor)

Como pode-se observar na Figura acima somente três classes são exportadas de fato para usuário Model, Schema, SchemaAttribute e TypedIndexedDb. A ideia para isso é esconder a complexidade interna e fornecer somente aquilo que é de fato necessário. Vale notar que TypedIndexedDb é exportado em formato singleton (6.4.3), logo por mais que esse módulo seja exportado diversas vezes só existirá um instância do mesmo em cada projeto. Em contrapartida o número de instâncias de Schema, Model e SchemaAttribute é ilimitado.

Dentro do arquivo package.json existem metadados que organizam o projeto para o npm, desde informações como nome do autor e do projeto, como instruções para o deploy do pacote.

```
{
  "name": "typed-indexed-db",
  "author": "Fábio de Abreu",
  "version": "0.1.7",
  "files": [
    "dist"
  ],
  "main": "dist/typed-indexed-db.umd.js",
  "module": "dist/typed-indexed-db.es.js",
  "types": "dist/index.d.ts",
  "scripts": {
    "prepack": "json -f package.json -I -e \"delete
this.devDependencies; delete this.dependencies\""
  }
}
```

Figura 6.11 Arquivo package.json

(FONTE: Imagem feita pelo autor)

Na imagem acima destaque-se algumas propriedades importantes relativas ao arquivo package.json do framework desenvolvido, enquanto algumas como “author”, “name” e “version” são auto explicativas, outras valem a pena ser aprofundadas. Primeiramente na entrada files indicamos quais arquivos serão publicados no pacote, nesse caso passamos apenas a pasta dist contendo o bundle gerado pela Rollup, dessa forma o código fonte em si não é exportado. Nas entradas “main” e “module” são indicados o arquivo raiz do bundle gerado, ou seja o primeiro arquivo que é de fato importado pelas aplicações. Para isso, são utilizadas duas entradas, pois "module" dá suporte a projetos modernos javascript que utilizam ES modules (Irianto, 2019) enquanto "main" dá suporte que utilizam UMD servindo como fallback para aplicações menos modernas. Nesse sentido, a principal distinção entre ambos é que o ES modules reduz o bundle final gerado pelo Rollup por sua sintaxe ser suportada pelos navegadores, diferente do UMD que precisa ser transpilado. Enquanto isso, na entrada types indicamos a raiz do arquivo de tipagem gerado pelo Rollup é importante

exportar a tipagem no pacote para garantir suporte também para projetos javascript. Por fim em "scripts.prepack" indicamos um script para ser executado antes do pacote ser realmente publicado. Nesse caso, o script utilizado remove dependências de desenvolvimento do projeto que não devem ser exportadas na versão de uso final.

A partir dessa organização foi feito a publicação do pacote executando o comando “npm publish” no terminal dentro da raiz do projeto. Antes disso, é preciso ter uma conta registrada no npm, pois o mesmo exige autenticação para publicação de pacotes. Dessa forma, a primeira versão do projeto “0.1.0” foi publicada em 12/06/2022, estando disponível para visualização no website do npm https://www.npmjs.com/package/typed-indexed-db. Além disso, a partir do momento em que foi publicada, qualquer desenvolvedor tem o poder de utilizar o Typed IndexedDB em seus projetos. Nesse sentido, a instalação pode ser feita dentro de projetos com o comando “npm install typed-indexed-db”. A seção posterior **6.6** irá se aprofundar em como a ferramenta pode ser utilizada a partir desse ponto.

6.6 Ponto de vista do usuário

Para ilustrar o ponto de vista do usuário desenvolvedor, que utiliza o framework projetado para gerenciamento de dados locais em seus projetos, foi criada uma aplicação web que testa funcionalidades do framework e que também pode servir de exemplo e documentação para uso do mesmo. Nesse sentido, o objetivo dessa aplicação não foi criar uma excelente experiência visual e sim demonstrar de maneira simples o uso da ferramenta Typed IndexedDb e como essa pode ser utilizada em projetos web. Sendo assim, a aplicação foi disponibilizada em https://typed-indexed-db-test-app.netlify.app/.

6.6.1 Aplicação de exemplo

Nessa seção, é apresentada a aplicação de teste desenvolvida suas funcionalidades e interface de usuário, sem adentrar muito em detalhes de como a mesma foi codificada internamente ou como o Typed IndexedDb foi utilizado por ela. Todavia, esses detalhes internos aparecem na seção **6.6.2**.

A aplicação de exemplo foi desenvolvida na linguagem javascript, utilizando a ferramenta React.js para simplificar a criação de interfaces de usuário. Vale notar que o framework desenvolvido pode funcionar com javascript ou typescript (no qual foi desenvolvido) e também com qualquer ferramenta para criação de UIs ou mesmo sem nenhuma.

Em uma visão de alto nível, esse app consiste em um instrumento para gerenciar usuários e tarefas. Cada usuário tem suas informações básicas de cadastro (nome, idade), e além disso metadados (data de criação e data da última edição) gerenciados automaticamente. Esses usuários, podem possuir tarefas vinculadas, que possuem nome, prioridade e usuário vinculado, além de uma data de criação. Em termos de ações, é possível criar usuários e tarefas relacionadas. Usuários também podem ser editados ou removidos, enquanto tarefas somente podem ser removidas. É importante mencionar que ao remover um usuário, suas tarefas vinculadas são também removidas. Nesse sentido, todos esses dados são armazenados no navegador, ficando salvos no espaço de armazenamento destinado ao indexedDB e como será visto na próxima seção gerenciados pelo framework desenvolvido.

Apesar do foco da aplicação estar na simplicidade, visto que sua função é mostrar o uso do projeto deste trabalho, os componentes da aplicação foram projetados seguindo

padrões de responsividade de forma que devam se adaptar a diferentes tamanhos telas, sejam laptops, tablets, celulares. Dessa forma, seguem figuras referente às telas desenvolvidas para esse app.

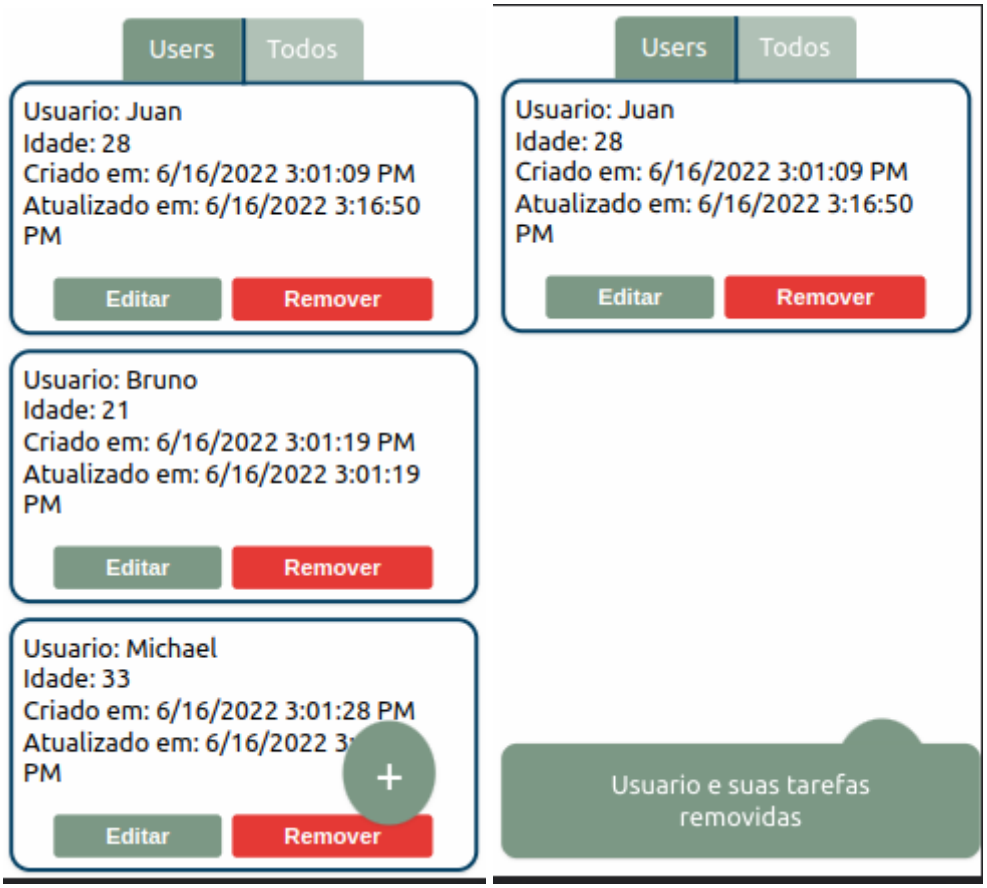

Figura 6.12 Aplicação de testes lista de usuários

(FONTE: Imagem feita pelo autor)

Na Figura **6.12**, é possível observar a tela em que usuários são listados com suas informações e possíveis ações sobre os mesmos. Além disso, na porção inferior, existe na primeira imagem um botão de adicionar usuário e também na segunda imagem uma mensagem exibida temporariamente após excluir um usuário com tarefas associadas.

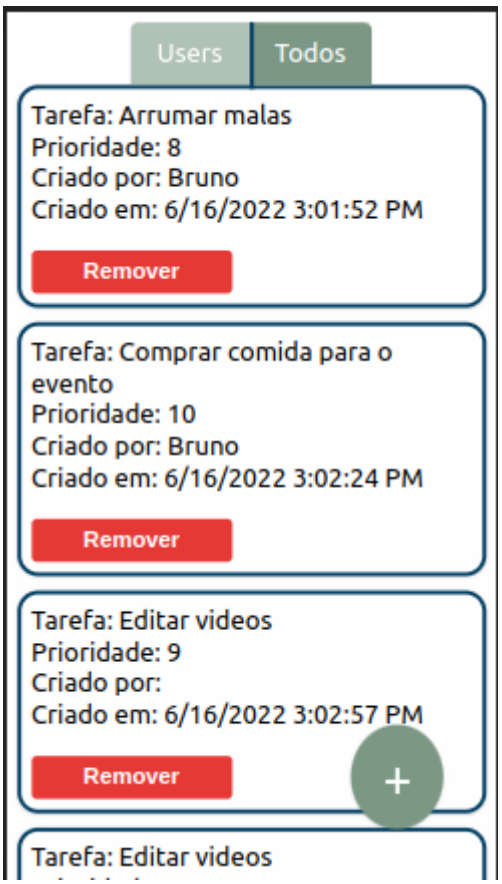

Figura 6.12 Aplicação de testes lista de tarefas

(FONTE:Imagem feita pelo autor)

Nessa Figura **6.12** é mostrada uma Figura similar à anterior, porém o que é listado dessa vez são as tarefas cadastradas. Utilizar o botão de adicionar nesta tela, diferentemente da anterior, abre um modal para criação de tarefas. Vale notar que as visões de usuários e tarefas são alternadas através de duas tabs na parte superior com valores “Users” para usuários e “Todos” para tarefas.

Figura 6.13 Aplicação de testes Modais para criação de usuários e tarefas

(FONTE: Imagem feita pelo autor)

Por fim, nessa Figura **6.13** temos os modais para criação de usuários e tarefas, considerando que para usuários também é possível realizar edições, o mesmo modal é utilizado ao criar e editar usuários, com a diferença que nesse caso já abre pré-preenchido com os dados atualmente salvos no banco. Dessa forma passada uma visão de alto nível sobre aplicação na próxima seção será abordado o uso do Typed IndexedDb nesse app teste.

6.6.2 Uso do Typed IndexedDB

Nessa seção será visto como é possível utilizar o Typed IndexedDb, para gerenciamento de dados, dentro de uma aplicação utilizando o app de testes descrito em **6.6.1**. É importante ressaltar que o app foi desenvolvido em javascript e utiliza a ferramenta React para gerenciamento da interface visual. Nesse sentido, a instalação do Typed

IndexedDB foi feita através do NPM (mais detalhes sobre esse tema em **6.5)**. Além disso, essa seção não se aprofunda em todas as características da aplicação e sim na parte de gerenciamento de dados e como isso se relaciona com as demais.

Ao elaborar um framework para IndexedDB, um dos principais objetivos foi poder isolar a organização dos dados do desenvolvimento da interface e da lógica de negócios das aplicações. Nesse sentido, dentro do app de testes foi criado um diretório schemas que contém os schemas que servem como molde para dados do tipo usuário e tarefa usados ao longo da aplicação, vejamos na Figura **6.14** a implementação do esquema tarefa ou todo.

```
import { Schema, SchemaAttribute } from "typed-indexed-db";

const todoSchema = new Schema(
  "todo",
  SchemaAttribute.object.shape({
    task: SchemaAttribute.string.isRequired(),
    priority: SchemaAttribute.number,
    created_by: SchemaAttribute.number,
    created_at: SchemaAttribute.number,
  })
);

export default todoSchema;
```

Figura 6.14 Código do arquivo schema/todo.js

(FONTE: Imagem feita pelo autor)

Primeiramente vale notar que Schema e SchemaAttribute são importados de “typed-indexed-db“ e utilizados para criar o esquema da mesma maneira que foi visto na seção **6.2**. Dessa forma, uma instância de todoSchema é por fim exportada por esse arquivo. É importante saber que essa instância é criada somente uma vez dentro da aplicação. Logo, cada vez que o arquivo em questão é importado, o mesmo objeto todoSchema é retornado, assim garantindo consistência na aplicação de forma que em diversos lugares é possível utilizar o mesmo Schema para um *todo* e também evita preencher a memória com diversas versões desse Schema.

Como mencionado em **6.4.3,** para utilizar o Typed Indexed e seus schemas e modelos, é preciso também criar uma instância de banco de dados com a classe TypedIndexedDb e passar os schemas que fazem parte desse espaço de armazenamento. Nesse sentido, vejamos parte do código de App.jsx onde isso acontece.

```
import { TypedIndexedDb, Model } from "typed-indexed-db";

import todoSchema from "./schemas/todo";
import userSchema from "./schemas/user";

const createDatabase = async () => {
    const res = await TypedIndexedDb.createDb("dados", [todoSchema,
userSchema]);
    if (!res.created) return;

    updateUsers();
    updateTodos();
};

const updateUsers = async () => {
    const storedUsers = await userSchema.findAll();
    setUsers(storedUsers);
};


const updateTodos = async () => {
    const storedTodos = await todoSchema.findAll();
    setTodos(storedTodos);
};
```

Figura 6.15 Configuração do framework trechos de código seleccionados de App.jsx

(FONTE: Imagem feita pelo autor)

Em createDatabase, é feita a configuração a partir da classe TypedIndexedDb. No exemplo, é criado um espaço chamado "dados" para que modelos com schemas do tipo "userSchema" ou "todoSchema" fiquem contidos. Nesse sentido, é importante lembrar que a função TypedIndexedDb.createDb só cria um banco de dados novo caso não exista nenhum banco no navegador com o nome fornecido. Nesse cenário, é somente feita a conexão entre os schemas do código fonte e as colunas já existentes no banco de dados.

Assim que o banco é iniciado, são chamadas as funções updateTodos e updateUsers, que têm como objetivo sincronizar os dados da aplicação na memória principal e os dados do banco de dados. Ambas as funções são bem similares: a primeira linha carrega todos os dados vinculados a um schema no banco utilizando a função schema.findAll e salva seu resultado no estado da aplicação usando setTodos e setUsers, que são funções de gerenciamento de estado de aplicação fornecidas pelo React. Além disso, nesse exemplo de código também é possível notar a simplicidade em que a leitura de dados e conexão de dados acontece devido ao uso do Typed IndexedDb em conjunto com o async await (**2.3.3**), fornecendo funcionalidades de assincronismo modernas para projetos web.

Dentro de App.jsx também acontece a criação e edição de usuários a partir dos dados preenchidos no Modal (visto na Figura **6.13**) essa inserção e atualização de dados é feita a partir de objetos do tipo Model, a imagem seguinte demonstra como isso a inserção de um *todo* é feita.

```
import { Model } from "typed-indexed-db";
import todoSchema from "./schemas/todo";

const handleSubmitTodo = async (e) => {
    e.preventDefault();

    const now = (Date.now() / 1000) | 0;
    if (editingTodo === 0) {
      const todo = new Model(todoSchema).build({
        task: taskName,
        priority: Number(priority),
        created_at: now,
        created_by: Number(selectTodoUser),
      });

      todo.save();
      setEditingTodo(-1);
      return updateTodos();
    }
}
```

Figura 6.16 Inserção de um todo trechos de código seleccionados de App.jsx

(FONTE: Imagem feita pelo autor)

A Figura **6.16** acima mostra como acontece a criação de um *todo*, Observa-se que é criado um Modelo com schema do tipo todoSchema e nesse *todo* é então chamada a função *build* que preenche os dados do *todo*. Em seguida, é chamada a função *save* que de fato salva o modelo e seus dados no banco de dados. Vale notar que nesse caso não é utilizado a palavra chave *await*. Embora a inserção de um modelo no banco a partir da chamada *model.save*() seja sim assíncrona, não é preciso utilizar essa palavra-chave se não existir necessidade de ler os dados da inserção. Dessa forma, a chamada por *save* é simplesmente delegada ao loop de eventos (descrito em **2.3**) para ser executada, porém não existe espera pelo seu resultado. Após o salvamento no banco, é chamada também a função *updateTodos*, mesma da imagem **6.15** que atualiza os dados do banco no estado da aplicação e consequentemente na interface de usuário. Dessa forma, a atualização de dados ocorre de forma análoga a inserção, com a diferença que para modelos pré-existentes deve ser utilizada a Função *Model.put* no lugar de *Model.build* como no exemplo anterior. O salvamento em si feito por *Model.save* é feito da mesma forma.

Outras funcionalidades do framework desenvolvido também utilizadas nesta aplicação de testes foram *schema.deleteById*, que deleta um item de um schema pelo id, *Model.delete*, que deleta o item do banco vinculado ao modelo, *schema.findById*, que retorna um modelo do schema associado ao id informado e *schema.find*, que retorna uma lista de modelos que se encaixam nas condições passadas por parâmetro. Todas essas podem ser utilizadas de forma simples utilizando o async await, de forma que considera-se o uso das mesmas intuitivo após o entendimento dos demais exemplos nesta seção.

Com o desenvolvimento da aplicação de testes foi possível verificar a utilidade e facilidade de uso do framework desenvolvido. Além disso, o código desenvolvido ficará como exemplo, em forma de documentação auxiliar, para projetos futuros mais complexos que desejam utilizar o Typed IndexedDb como sua ferramenta.

7 CONCLUSÕES

Em termos gerais o projeto do framework alcançou seu objetivo principal, fornecer uma maneira simples para utilizar o indexedDB dentro dos navegadores oferecendo funcionalidades e soluções de como estruturar os dados através de Model e Schemas, além da possibilidade de criar diferentes espaços de dados, para projetos que necessitem dessa funcionalidade mais avançada. Em termos de simplificação para chamadas assíncronas (**2.3.3**) ao banco de dados, não é possível para usuários utilizar callbacks (seu uso é abstraído no código fonte), promises são opcionais, enquanto o async await, mais moderno, é recomendado, na maioria dos casos, pelo fato de utilizar menos linhas de código. Nesse sentido, todos os exemplos utilizam essa sintaxe (async await) para os métodos de interação com o banco. Logo, esse objetivo também é considerado completo.

Ademais, o framework foi também disponibilizado no gerenciador de pacotes npm, como era esperado, estando disponível para projetos com javascript ou typescript. Nessa via, foram registrados 349 downloads do framework em sua primeira semana de npm, número interessante mas certamente não é possível tirar conclusões no que tange a isso tão cedo.

Apesar disso, assim como Bjarne Stroustrup afirma que existem dois tipos de linguagem de programação, as que as pessoas reclamam e as que ninguém usa, o mesmo vale para software. Dessa forma, é desejado que à medida que o framework possa ganhar popularidade dentro do universo do desenvolvimento, novas demandas e sugestões de usuários surjam, trazendo possibilidades de melhorias para o projeto, afinal software que não é atualizado perde seu sentido. Outrossim, é esperado que após a entrega do projeto final o código fonte do framework seja disponibilizado no github, ajudando o entendimento do framework para usuários que queiram se aprofundar ou mesmo sugerir mudanças tenham essas possibilidades. Por hora, a única versão do framework disponibilizada no npm é o bundle gerado pelo Rollup e o código fonte não pode ser encontrado em nenhum lugar, embora possa ser utilizado. No ponto de vista do criador, três principais ideias de melhorias seriam, a criação de uma documentação altamente simplificada com exemplos de configuração, instalação e uso para que o básico da ferramenta possa ser rapidamente assimilado por outros desenvolvedores que possam a partir disso fazer seus próprios testes utilizando o Typed Indexeddb em suas próprias aplicações. Enquanto que a segunda melhoria

seria trazer a opção de utilizar ids gerados aleatoriamente para os Models inseridos no banco, função atualmente não suportada pelo projeto e que também não é suportada pelo IndexedDb. Todavia, até o momento é possível utilizar ids numéricos com incremento automático. Por fim, a terceira e a última ideia seria criar uma versão da aplicação de testes utilizando o IndexedDb sem o framework desenvolvido, para que sirva de comparação em termos de linhas de código utilizadas e tempo para desenvolvimento.

Por conseguinte, pode-se dizer que o projeto de framework que passou a se chamar Typed IndexedDb atingiu seus objetivos, possui de fato possibilidades para extensões e melhorias, o que deve ser visto como um ponto positivo pois software deve ser sempre aperfeiçoado. Considera-se que a ferramenta já pode ser utilizada e beneficiar projetos de software do tipo aplicação com escopo maior que precisem de soluções para gerenciamento de dados client-side.

8 REFERÊNCIAS

RANGANATHAN, Arun. Beyond HTML5: Database APIs and the Road to IndexedDB. [S. l.]: Mozilla Hacks, 1 jun. 2010. Disponível em: https://hacks.mozilla.org/2010/06/beyond-html5-database-apis-and-the-road-to-indexeddb/. Acesso em: 12 dez. 2021.

MDN WEB DOCS. Progressive web apps (PWAs). [S. l.], 28 dez. 2021. Disponível em: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps. Acesso em: 6 jan. 2022.

HICKSON, Ian; GOOGLE INC (ed.). Web SQL Database. [S. l.]: W3C - World Wide Web Consortium, 18 nov. 2010. Disponível em: https://www.w3.org/TR/webdatabase/. Acesso em: 8 jan. 2022

MICROSOFT DOCS. Visão geral dos Aplicativos Web Progressivos (PWAs. [S. l.]: Microsoft Docs, 15 set. 2021. Disponível em: https://docs.microsoft.com/pt-br/microsoft-edge/progressive-web-apps-chromium/. Acesso em: 26 dez. 2021.

FOWLER, Martin; J. SADALAGE, Pramod. Nosql Essencial: Um Guia Conciso Para o Mundo Emergente da Persistência Poliglota. [S. l.: s. n.], 2012.

GYŐRÖDI, Cornelia; GYŐRÖDI, Robert; SOTOC, Roxana. A Comparative Study of Relational and NonRelational Database Models in a Web- Based Application. [S. l.]: (IJACSA) International Journal of Advanced Computer Science and Applications, 6 nov. 2015. Disponível em:

https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.736.5333&rep=rep1&type=pdf. Acesso em: 1 fev. 2022

BHAT, Uma; JADHAV, Shraddha. Moving Towards Non-Relational Databases. [S. l.]: International Journal of Computer Applications, 22 jan. 2010. Disponível em: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.447.8328&rep=rep1&type=pdf. Acesso em: 1 fev. 2022

LEPAGE, Pete. Storage for the web. [S. l.]: Web.Dev, 27 abr. 2020. Disponível em: https://web.dev/storage-for-the-web/. Acesso em: 19 dez. 2021.

Alex. Storage for the web. IndexedDB, Cache API. [S. l.]: Tech Planet, 19 jul. 2020. Disponível em: https://techplanet.today/post/storage-for-the-web. Acesso em: 27 jan. 2022.

CROCKFORD, Douglas. O Melhor do JavaScript. 1. ed. [S. l.]: Yahoo Press, 2008. Ebook.

MICROSOFT (Typescriptlang.org). TypeScript for JavaScript Programmers. [S. l.], 18 nov. 2021. Disponível em: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html. Acesso em: 22 dez. 2021.

HTTPS://WWW.TYPESCRIPTLANG.ORG/ASSETS/TYPESCRIPT-HANDBOOK.PDF. In: MICROSOFT. The TypeScript Handbook. [S. l.], 13 out. 2020. Disponível em: https://www.typescriptlang.org/assets/typescript-handbook.pdf. Acesso em: 4 jan. 2022.

SILVA, Ricardo; PRICE, Roberto. A busca de generalidade, flexibilidade e extensibilidade no processo de desenvolvimento de frameworks orientados a objetos. [S. l.], 1998. Disponível em: https://www.inf.ufsc.br/~ricardo.silva/publications/Ideas98.PDF. Acesso em: 5 jan. 2022.

WAZLAWICK, Raul. Engenharia de Software para Sistemas de Informações: Conceitos e práticas que fazem sentido. Florianópolis: [s. n.], 2012.

VUKICEVIC, Vladimir. Web Storage & SQL. [S. l.]: W3C, 8 abr. 2009. Disponível em: https://lists.w3.org/Archives/Public/public-webapps/2009AprJun/0106.html. Acesso em: 2 fev. 2022.

FOWLER, Martin. Padrões de arquitetura de aplicações corporativas. [S. l.]: Addison-Wesley, 2003. 559 p.

THE DATABASE that Syncs. [S. l.], 2016. Disponível em: https://pouchdb.com/. Acesso em: 6 mar. 2022.

A MINIMALISTIC Wrapper for IndexedDB. [S. l.], 2015. Disponível em: https://dexie.org/. Acesso em: 6 mar. 2022.

IDB-WRAPPER. [S. l.], 2018. Disponível em: https://www.npmjs.com/package/idb-wrapper. Acesso em: 6 mar. 2022.

INDEXED Database API 3.0: W3C Recommendation, 30 January 2018. W3C: Ali Alabbas e Joshua Bell, 30 jan. 2018. Disponível em: https://www.w3.org/TR/IndexedDB-2. Acesso em: 7 mar. 2022

RANGANATHAN, Arun; WILSHER, Shawn. Firefox 4: An early walk-through of IndexedDB. Hacks Mozilla, 1 jun. 2010. Disponível em: https://hacks.mozilla.org/2010/06/comparing-indexeddb-and-webdatabase/. Acesso em: 7 mar. 2022.

VETTOR, Rob. Relacional versus Dados NoSQL. [S. l.], 2022. Disponível em: https://docs.microsoft.com/pt-br/dotnet/architecture/cloud-native/relational-vs-nosql-data. Acesso em: 3 fev. 2022.

IRIANTO, Igor. What Are CJS, AMD, UMD, and ESM in Javascript?. [S. l.], 2019. Disponível em: https://irian.to/blogs/what-are-cjs-amd-umd-and-esm-in-javascript/. Acesso em: 4 nov. 2021.

MAKING PWAs work offline with Service workers. MDN Web Docs, 2022. Disponível em: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Offline_Service_workers. Acesso em: 11 mar. 2022.

CACHESTORAGE: The CacheStorage interface represents the storage for Cache objects.. MDN Web Docs, 2022. Disponível em: https://developer.mozilla.org/en-US/docs/Web/API/CacheStorage. Acesso em: 11 mar. 2022.

ROLLUP. Rollup: Rollup is a module bundler for JavaScript which compiles small pieces of code into something larger and more complex, such as a library or application.. [S. l.], 2022. Disponível em: https://rollupjs.org/guide/en/. Acesso em: 26 mar. 2022.

TEELING, Michael. What is Data Integrity? Learn How to Ensure Database Data Integrity via Checks, Tests, & Best Practices. [S. l.], 14 maio 2012. Disponível em: Michael Teeling. Acesso em: 14 jul. 2022.

Apêndices

Apêndice A - Código fonte

O código fonte, tanto do framework TypedIndexedDb quanto da aplicação de testes, se encontra disponível para acesso em um repositório do Bitbucket link: https://bitbucket.org/fabio0abreu/typed-indexed-db/src/master/

Apêndice B - Artigo

Neste apêndice será apresentado o artigo no formato SBC, referente ao presente projeto.

Framework para gerenciamento de dados client-side com IndexedDB

Fábio de Abreu[1]

[1]Departamento de informática e estatística
Universidade Federal de Santa Catarina (UFSC) - Florianópolis SC Brasil

abreu.fabio@grad.ufsc.br

Abstract. *With the evolution of web applications and the popularization of PWAs - Progressive Web Apps, management of large amounts of data inside browsers has become increasingly essential. In that sense, the IndexedDB web API shines as a modern browser native, low level and highly performant solution. However the manipulation of data through this tool is complex, requiring high developer time, and therefore cost, to handle unusual concepts from IndexedDB API, in addition to the asynchronous nature of the instrument, making it efficient for development in its root state. In that manner, the referred project aims to create an alternative form, so that developers can manage data with IndexedDb*

Resumo. *Com a evolução das aplicações web e principalmente a popularização dos PWAs - Progressive Web Apps, é cada vez mais necessário o gerenciamento de grandes quantidades de dados dentro dos navegadores. Nesse sentido, o IndexedDB se destaca como uma API, nativa dos navegadores modernos, de baixo nível e alta performance. Entretanto, a manipulação dos dados por meio dela é complexa, exigindo que o desenvolvedor gaste grande quantidade de tempo com conceitos pouco usuais da API do IndexedDB e também de programação assíncrona. Nesse sentido, o projeto referenciado busca criar uma forma alternativa para que desenvolvedores possam gerenciar esses dados usando o IndexedDb, criando um framework para gerenciamento de dados client-side*

1. Introdução

Em projetos de software é ideal que a atenção dos desenvolvedores esteja voltada da maior maneira possível aos requisitos funcionais, visto que é dessa forma que o cliente final percebe o valor no sistema. Nesse sentido, para o gerenciamento de dados em uma aplicação é esperado pelos desenvolvedores, ferramentas de uso simples de forma que o fluxo de seu recurso mais valioso, o tempo, seja direcionado aos problemas a qual esse software se propõe a solucionar. Considerando que integridade de dados, garantia que dados são salvos exatamente como intencionado, é essencial para qualquer aplicação e também é um componente fundamental na segurança da informação (Teeling, 2012). O projeto analisado busca criar um framework que abstraia o uso do IndexedDb oferecendo, não somente abstraindo sua complexidade mas também oferecendo novas funcionadlides para estrturar e garantir a integridade do banco de dados.

2. O projeto

O projeto em questão, se baseou em um metodo de pesquisa com natureza aplicada e uma abordagem qualitativa utilizando procedimentos técnicos bibliográficos e estudo de caso. Nesse sentido, o objetivo principal proposto foi Criar um framework que abstraia o gerenciamento de dados cliente-side, utilizando o indexedDB, garantindo facil integração a projetos de software e disponibizar o resultado final ao NPM - Node Package Manager. Em termos de codigo, o projeto se centrou em oferecer uma estrutura baseada em ODMs - Object Document Mapper, de forma a abstrair os dados na memoria principal e banco de dados. Esses dados interagem dentro do framework por meio Modelos e Esquemas que organizam o formato esperado do banco de dados fornecendo funcionalides de atualização, remoção e inserção que garantem tipagem, estrura de dados mas também de codigo garantindo a integridade dos dados. Vale notar que o framework em questão foi chamado Typed IndexedDb.

3. Aplicação de teste

Para validar os resultados do projeto, foi desenvolvida na linguagem javascript, utilizando a ferramenta React.js para simplificar a criação de interfaces de usuário. De forma simplificada, esse app teste consiste em um instrumento para gerenciar usuários e tarefas. Embora, seja uma interface simples, visto que estetica de UI não era o objetivo principal por trás da aplicação, o app de testes cumpre seu objetivo de validar o framework desenvolvido. Nesse sentido, não só o app importa o framework com o uso do NPM, toda a estrtura de dados é montada através do Typed IndexedDb, que por sua via cordena todas as ações de salvamento, atualização inserção e deleção de dados da aplicação. De certa forma, é nitido que o framework facilitou muito a criação da aplicação como o mesmo interage com o IndexedDb, ainda assim em suas conclusões é mencionado que para mostrar de maneira ainda mais clara seria possivel re-criar o mesmo app sem o uso do framework. Dessa forma, poderia ser comparado o tempo de desenvolvimento e numero de linhas de codigo utilizadas e assim demonstrando ainda mais o poder do projeto de framework.

4. Analise dos resultados

termos gerais o projeto do framework alcançou seu objetivo principal, fornecer uma maneira simples para utilizar o indexedDB dentro dos navegadores oferecendo funcionalidades e soluções de como estruturar os dados através de Model e Schemas, além da possibilidade de criar diferentes espaços de dados, para projetos que necessitem dessa funcionalidade mais avançada. Em termos de simplificação para chamadas assíncronas (2.3.3) ao banco de dados, não é possível para usuários utilizar callbacks (seu uso é abstraído no código fonte), promises são opcionais, enquanto o async await, mais moderno, é recomendado, na maioria dos casos, pelo fato de utilizar menos linhas de código. Nesse sentido, todos os exemplos utilizam essa sintaxe (async await) para os métodos de interação com o banco. Logo, esse objetivo também é considerado completo. Ademais, o framework foi também disponibilizado no gerenciador de pacotes npm, como era esperado, estando disponível para projetos com javascript ou typescript. Nessa via, foram registrados 349 downloads do framework em sua primeira semana de npm, número interessante mas certamente não é possível tirar conclusões no que tange a isso tão cedo. Ainda assim, é esperado que o projeto ainda passe por uma evolução agora que será de fato disponibilizado para uso e feedback por parte da comunidade open source. Como

destaque, três funcionalides interessantes e que poderiam aparecer no roadmap para o futuro da ferramenta são: a criação de uma documentação altamente simplificada com exemplos de configuração, instalação e uso para que o básico da ferramenta possa ser rapidamente assimilado por outros desenvolvedores que possam a partir disso fazer seus próprios testes utilizando o Typed Indexeddb em suas próprias aplicações. Enquanto que a segunda melhoria seria trazer a opção de utilizar ids gerados aleatoriamente para os Models inseridos no banco, função atualmente não suportada pelo projeto e que também não é suportada pelo IndexedDb. Todavia, até o momento é possível utilizar ids numéricos com incremento automático. Por fim, a terceira e a última ideia seria criar uma versão da aplicação de testes utilizando o IndexedDb sem o framework desenvolvido, para que sirva de comparação em termos de linhas de código utilizadas e tempo para desenvolvimento. Pode-se dizer que o projeto de framework que passou a se chamar Typed IndexedDb atingiu seus objetivos, possui de fato possibilidades para extensões e melhorias, o que deve ser visto como um ponto positivo pois software deve ser sempre aperfeiçoado. Nsse sentido, considera-se que a ferramenta já pode ser utilizada e beneficiar diversos tipos de aplicação, sendo que como previsto encontra-se disponivel e com fácil instalação pelo gerenciado de pacotes NPM.

5. References

TEELING, Michael. What is Data Integrity? Learn How to Ensure Database Data Integrity via Checks, Tests, Best Practices. [S. l.], 14 maio 2012. Disponível em: Michael Teeling. Acesso em: 14 jul. 2022.