Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

André Luiz Abdalla Silveira

DELPo
– Continuando a refatoração –

São Paulo
15 de Novembro de 2.019

DELPo
– Continuando a refatoração –

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Marco Dimas Gubitoso

São Paulo
15 de Novembro de 2.019

Agradecimentos

Meus agradecimentos pricipais são a Deus, meus adorados pais que me proporcianram (e continuam a fazê-lo) as melhores oportunidades que eu poderia ter. Meu agradecimento também dirige-se ao meu orientador que me apoiou em todos os momentos do meu trabalho. Agradeço também por ter me introduzido ao pessoal do NEHiLP que me proporcionou a graça de trabalhar com o DELPo. Tem sido uma experiencia e tanto.

Outro agradecimento importante dirige-se ao pessoal da Codeminer 42, empresa em que trabalhei por algum tempo. Em especial a Elias e Alessandro. O primeiro trabalhava a meu lado e sempre me ajudava quando tinha disponibilidade. O último também deu uma grande ajuda, e além de tirar algumas dúvidas, forçava-me a escrever um código que seguisse, da melhor forma possível, os princípios do bom código

Resumo

O projeto do Dicionário Etimológico da Língua Portuguesa (DELPo) nasceu da necessidade dos pesquisadores que se dedicam ao estudo da etimologia do idioma contarem com informações a respeito da origem e do significado dos verbetes. O dicionário estará disponível online, mas de forma incompleta, dado que muitas coisas foram refatoradas como o banco de dados (agora em PostgreSQL), módulos foram refatorados de forma que as dependências fiquem no seu devido lugar. A qualidade do código foi peça chave nessa refatoração, bem como a corbertura completa de testes e a implementação de uma interface agradável ao cliente (pesquisadores do NEHiLP)
Palavras-chave: dicionário, etimologia, banco de dados, desenvolvimento web, testes, UI

Abstract

The project 'Dicionário Etimológico da Língua Portuguesa (DELPo)' came from the need of researchers dedicated to etymology studies to count on information regarding the origin and meaning of entries. The dictionary will be available online, in spite of being incomplete, given that there are many things that have been refactored, like the database structure and some dependencies put in their rightful place. Other important things to this project was to bring an intensive test coverage to the Rails API and provide an enjoyable user interface to the clients (NEHiLP researchers)
Keywords: dictionary, etymology, database, web development, tests, UI

Sumário

Lista de Figuras **vi**

1 Da Introdução **1**
1.1 Uma observação Importante . . . 2
1.2 Objetivos Específicos . . . 2
1.3 Sobre a proposta . . . 2
1.4 Metodologia . . . 3
1.5 Descrição dos capítulos . . . 4

2 Dos conceitos principais **5**
2.1 O projeto e os conceitos associados . . . 6
2.1.1 Sobre o NEHiLP e o DELPo . . . 6
2.1.2 Como funciona o processo de inserçã ode palavras através da moagem? 6
2.1.3 Sobre a análise etimológica do DELPo . . . 7
2.2 O Back-end . . . 8
2.2.1 Ruby . . . 9
2.2.2 Rails . . . 10
2.2.3 RSpec . . . 11
2.3 O Front-end . . . 12
2.3.1 Vue JS . . . 12
2.3.2 Nuxt JS . . . 12
2.3.3 Jest . . . 14
2.4 Boas práticas em programação . . . 15
2.4.1 SOLID . . . 15
2.4.2 Outras recomendações . . . 16

3 Sobre o Desenvolvimento **17**
3.1 Arquitetura de projeto . . . 18
3.1.1 Por que as ferramentas descritas foram utilizadas ao invés de outras similares? . . . 18
3.1.2 Por que mudar a implementação do front? Por que usar o modo API? 18
3.1.3 Por que refatorar o banco de dados e outros setores da aplicação? . . 19

3.1.4 Como será o desenvolvimento do front end? . . . 19
3.2 Decisões de código . . . 19
3.2.1 RR – Na sessão anterior foi mencionado que o banco de dados foi alterado. Que alterações são essas, e por que foram feitas? . . . 19
3.2.2 N – Por que a implementação do design pattern Builder foi abandonada na criação de formulários no front-end? . . . 21
3.2.3 N – Existe uma quantidade considerável de código repetido no front. Isso não é algo ruim? . . . 22

4 Dos Resultados Obtidos 23
4.1 O que foi feito? . . . 24
4.1.1 API Rails . . . 24
4.1.2 Interface . . . 24
4.1.3 O moedor . . . 24
4.1.4 Vendo as coisas funcionar . . . 24
4.1.5 Testes e cobertura . . . 24
4.2 O que será feito? . . . 24
4.2.1 Refatorações . . . 24

5 Das Conclusões 25
5.1 A história deste trabalho . . . 25
5.2 O que aprendi com esse trabalho . . . 25
5.3 Legado Esperado . . . 25

Referências Bibliográficas 26

Lista de Figuras

2.1 Hierarquia de dados do DELPo 7
2.2 Consegue entender o que está sendo feito? 9
2.3 Como funciona uma aplicação MVC (fonte) 10
2.4 Exemplo de programa e teste respectivo 11
2.5 Exemplo de um componente Vue 13
2.6 Organização de pastas e arquivos 14
2.7 Exemplo de um conjunto de testes usando Jest 15

3.1 Como estavam implementadas as acepções 20

Capítulo 1

Da Introdução

Neste capítulo

1.1 Uma observação Importante 2
1.2 Objetivos Específicos 2
1.3 Sobre a proposta 2
1.4 Metodologia 3
1.5 Descrição dos capítulos 4

— Introdução —

Segundo o manual do NEHiLP, p.15, o idioma português é aquele com menor quantidade de informação etimológica disponível se comparado com outros idiomas românicos. Com vistas a minorar esse problema, alguns projetos de pesquisa visam métodos de otimizar a obtenção de registros. O projeto DELPo é um ótimo exemplo. Através de um algoritmo *modeor* que recebe um arquivo, registram-se ocorrências, lemas e flexões que auxiliam a monitorar que elementos são mais recentes. Este fluxo mimetiza o algoritmo que se aplica a cartões físicos.

O projeto já conta com um portal web em produção. Este trabalho, entretanto, era feito em PHP, e havia alguns gargalos na aplicação. Alguns foram removidos ano passado, outros estão sendo estirpados agora. Além disso, cada parte da implemetação agora está sendo feita com as ferramentas que o aluno julga mais apropriadas. Por exemplo, a parte do projeto implementada em Rails (Ruby on Rails) é o "backend" enquanto as visualizações são feitas em Nuxt.js

1.1 Uma observação Importante

Por se tratar da continuação do trabalho de Santin, Galati, e Cardoso (2018), muitas coisas foram extraídas do trabalho deles. Muitas citações e paráfrases estarão presentes neste trabalho, principalmente quando se trata dos conceitos relacionados à parte do cliente. Isso se deve pelo fato óbvio do trabalho ser quase o mesmo, com poucas diferenças (a proposta é muito semelhante, mas o cliente e a aplicação são os mesmos). Desta forma, não há motivo razoável para que não se mantenha muito do que foi feito.

1.2 Objetivos Específicos

Aqui tratar-se-á de todos os objetivos de forma específica e como pretendo cumpri-los.

Gerar um sistema eficiente. Trata-se de uma parte muito importante deste trabalho. Dado que no sistema legado (em PHP) observavam-se alguns gargalos de desempenho, e que estes comprometiam a utilização por parte de pesquisadores, este trabalho propõe-se a eliminar, ou reduzí-los de maneira a facilitar a vida dos entusiastas e estudiosos da área.

Além da motivação nobre de ajudar, não se pode ignorar o impacto de um grande trabalho como este terá no currículum do aluno em questão. Um trabalho bem feito abrirá portas para grandes oportunidades que aparecerão dado que quem faz um excelente serviço poderá fazer muitos outros. Do contrário, um serviço mal feito, um software de qualidade duvidosa não vai abrir portas ao aluno, que nem se daria ao trabalho de mostrar algo mal-feito.

Verificar a importância real de um código limpo. Ao início da graduação, o aluno incauto pode fazer uma vaga ideia, mas jamais entenderá a importância de fazer um código legível [1]. Isso muda quando o aluno têm contato com um código ilegível. A dificuldade em refatorar o código, entender o que se deseja, ou até mesmo a necessidade de refazer grandes partes daquele programa será traumática o bastante para que o aluno passe a se preocupar com o que ele faz.

Mas o que é um código de limpo? Segundo Lukaszuk, é um código que pode ser lido, compreendido, melhorado, extendido e mantido por qualquer outro desenvolvedor.

Ele fez um trabalho importantíssimo ao juntar vários preceitos em um só documento. Um grande punhado de convenções (ou conselhos, se preferir) para quem deseja fazer um código excelente. Por se tratar de mais de 50 "leis", é um tanto trabalhoso que sigam-se todos eles, mas com certeza se trata de um bom ponto de partida.

Na sessão 2.4 econtram-se algumas diretrizes da produção de bom código.

1.3 Sobre a proposta

Vale saber, além dos objetivos supra citados, qual é a proposta geral desse trabalho.

- continuação do trabalho de criar uma aplicação em Rails[2] da API usada no novo portal do DELPO. Inclui cobertura extensiva de testes de unidade.
- retoque no banco de dados da aplição e articulação das entidades usando PostgreSQL
- modificação do modelo de autenticação de usários e implementação de níveis de permissão compatível com o que o manual do NEHiLP determina

[1] Salvo o caso desse aluno já tiver alguma experiência com programação

[2] vide p.10

- melhoramento do sistema moedor [3]
- Permitir que a plataforma esteja disponível para uso da melhor forma possível. Ainda que isso implique que algumas refatorações e melhorias ficarem de lado.

O moedor que há na aplicação atual é enorme e exige grande refatoração. Entretanto, como a bagunça é grande, (diferente do prazo) o que será feito são alguma melhorias em alguns métodos, remoção dos desnecessários, e por aí vai. A ideia é que, na medida do possível, as funções fiquem mais fáceis de entender, e o código ganhe em inteligibilidade. O moedor tem dois módulos: o de leitura e o de análise.

O primeiro trata de quebrar o texto em contextos (orações) e palavras. Cada quais destas é analisada com o objetivo de encontrar sua forma básica[4], procurar se ela existe no banco, bem como datar e retrodatar de acordo com o que for necessário. Os resultados obtidos são armazenados em uma tabela de mesmo nome.

O segundo começa a partir do ponto em que a leitura termina. Este módulo depende da interação do pesquisador que faz as partes da tarefas que a máquina não é capaz de fazer. Este módulo toma os resultados obtidos no módulo acima, e os apresenta ao pesquisador da forma como será mencionada na sub-seção 2.1.1 na descrição do programa moedor. Resumidamente, uma palavra contida na obra pode ser marcada em azul (se ainda não estiver no banco de dados) ou em vermelho (caso já exista no banco de dados e esteja associada a uma data *maior*[5] que a do texto analizado)

1.4 Metodologia

A metodologia seguida nesse trabalho consiste de alguns passos:

1. Conhecer o projeto a fundo.
2. Compreender o código.
3. Entender os relacionamentos entre as entidades do banco de dados
4. Realizar as mudanças necessárias (enquanto faz os testes)
5. Fazer a interface visual do sistema
6. Se possível, criar uma documentação suficiente para a compreensão e utilização dessa ferramenta

O passo 1 é o mais básico de todos e consistiu de uma conversa inicial com o professor Mário, «dizer o que ele é do projeto» do projeto DELPo. O professor Gubi também me ajudou a entender alguns aspectos importantes do assunto. Entretanto, o manual do NEHiLP e o artigo de Viaro (2017) foram de grande ajuda para entender a diferença entre os lemas, e tornaram-se poderosas fontes de consulta.

O passo seguinte foi o maior desafio, dado que eu só tenho contato regular com um dos membros do antigo grupo de trabalho. Isso tornou-se uma dificuldade tamanha e me gerou um grande bloqueio que me levou a reorganizar as entidades do banco de dados. Ainda que seja notável que a equipe anterior tem um bom domínio de ferramentas SQL, percebe-se

[3] vide p.6
[4] por exemplo, abrindo → abrir
[5] vide sub-seção 2.1.2 para mais detalhes

que não se utilizaram plenamente da interface provida pelo Rails para poder implementar melhor os modelos. Além disso, o algoritmo do moedor ainda é um pesadelo para quem se aventura a refatorá-lo.

O passo 3 também exigiu um bom tempo. Inicialmente tentou-se fazer menores alterações e manter o código. Isso seria muito menos trabalhoso, e demandaria menos tempo também. O que se observou, entretanto é que fazer mudanças específicas não era tão simples quanto parecia, ou quanto de fato seria se estivesse com um dos membros da antiga equipe consigo. Dessa forma, decidiu-se refazer as tabelas com bas nas que já tinham sido feitas. Enquanto os modelos ficavam prontos e se conectavam, os testes cobriam se as determinações e os relacionamentos foram realmente estabelecidos.

Além das mudanças supracitadas, o passo 4 engloba as mudanças nos controladores e a criação de métodos auxiliares (*helpers*). Esses métodos, inseridos em módulos independentes, garantem um código bem compartimentado, conciso e não repetitivo.

O maior desafio foi refatorar o moedor. Os principais problemas com o código são nomes de métodos que não descrevem o que ele faz, métodos desnecessários (como o que declara um hash que poderia ser uma constante dentro do módulo), e a funação principal do módulo com mais de mil linhas e com muitas atribuições desconhecidas.

A parte 5 está em desenvolvimento neste exato momento e conta com o auxílio da professora (?) Vanessa, também integrante do NEHiLP. Nos capítulos que se seguem, falar-se-á sobre as ferramentas usadas. Aqui basta dizer que o objetivo principal é proporcionar uma boa experiência ao usuário.

1.5 Descrição dos capítulos

O capítulo 2, **dos conceitos principais** funciona como um cabeçalho de um programa. Isso pois, nessa parte do código é onde ficam os comandos de importação de bibliotecas.

Analogamente, no próximo capítulo, tratar-se-ão de conceitos relativos ao material a ser implementado, e do jargão a ser utilizado. Detalhes a serem esclarecidos para que o cérebro do leitor "compile"(em outras palavras, que ele compreenda) o conteúdo das páginas que se seguirão.

No capítulo 3, **sobre o desenvolvimento**, tratar-se-ão de detalhes técnicos sobre a implementação. Será uma descrição a dois níveis, sendo a primeira, uma descrição verbal, e o seguinte, a nível de implementação.

No capítulo 4, **dos resultados obtidos**, encontra-se uma reflexão sobre o que se passou além do código, o que está pronto e o que pode ser feito no futuro.

O último capítulo, **das conclusões finais**, faz-se um resumo do que está documentado, destacando o que for mais conveniente.

Capítulo 2

Dos conceitos principais

Neste capítulo

2.1 O projeto e os conceitos associados 6
2.1.1 Sobre o NEHiLP e o DELPo 6
2.1.2 Como funciona o processo de inserçã ode palavras através da moagem? 6
2.1.3 Sobre a análise etimológica do DELPo 7
2.2 O Back-end 8
2.2.1 Ruby 9
2.2.2 Rails 10
2.2.3 RSpec 11
2.3 O Front-end 12
2.3.1 Vue JS 12
2.3.2 Nuxt JS 12
2.3.3 Jest 14
2.4 Boas práticas em programação 15
2.4.1 SOLID 15
2.4.2 Outras recomendações 16

— Introdução —

Nesse capítulo, o objetivo é explicitar os conceitos que serão cruciais para a compreensão do que é dito nos demais capítulos. Como já dito, funciona como o cabeçalho de um arquivo de código em que se inclui as bibliotecas e demais arquivos com definições e módulos e classes e tudo o mais. Tratando desse trabalho, é o capítulo que garante que o texto estará auto contido na medida do possível, ou seja, qualquer um que lê-lo, compreenderá o que estiver escrito sem ter que pesquisar muito fora daqui.

2.1 O projeto e os conceitos associados

2.1.1 Sobre o NEHiLP e o DELPo

Uma das finalidades principais do NEHiLP é pesquisar sobre Linguística Histórica, Filologia e Etimologia. Para a pesquisa, requere-se a análise de documentos das mais variadas épocas, a fim de recolher informações linguísticas. Gerar e organiar dados para o estudo da história da língua portuguesa é uma das principais metas deste trabalho.

Nos dias de hoje, o núcleo de pesquisa conta com projetos associados a diferentes universidades. Estes projetos são:

- DBB, DEPARC [1]
- LIBOLO [2]
- PHPP [3]
- MAP (Mulheres na América Portuguesa) [4]
- DELPo – centro da atenção nesse trabalho

A finalidade do DELPo é ser um dicionário etimológico online. A ideia é que cada acepção esteja acompanhada de sua ocorrência mais antiga, bem como o contexto nela inserida. O projeto consiste de um portal que, equipado com um banco de dados, pretende entregar algumas funcionalidades citadas abaixo. [5]

Moedor de textos: recebe um texto e uma data, e analisa suas palavras. Cada uma pode ser marcada em azul (caso ainda não esteja no banco de dados), ou em vermelho (caso já exista no banco de dados e esteja associada a uma data cronologicamente posterior à data do texto analisado)

Metaplasmador: recebe uma palavra em latim vulgar, e reproduz como ela se desenvolveu do latim ao português

N-GRAM: programa que avalia a quantidade de ocorrências de uma palavra ao longo dos anos.

2.1.2 Como funciona o processo de inserçã ode palavras através da moagem?

Para adicionar palavras no dicionário, é necessário que uma obra seja submetida por um pesquisador juntamente à sua data, ao Moedor do DELPo. Cada palavra presente no texto inserido é analisada, e então uma das seguintes ações é executada:

a se a palavra não estiver no banco de dados, o moedor marca-a como *inexistente*. Associa a ela a data da obra submetida e apresenta-a ao pesquisador, que decidirá se a ocorrência deve ser processada.

[1] vinculados a UFBA
[2] vinculado a Macau University e USP
[3] vinculado a UNESP e USP
[4] vinculados a USP
[5] Estas funcionalidade já existem. O objetivo deste trabalho, portanto, é melhorar e atualizá-las

b se a palavra já existir no banco de dados e estiver vinculada a uma data maior[6] do que a da obra submetida, ela (a palavra) é marcada como *retrodatou* e é associada à data do texto inserido. Também nesse caso, o vocábulo é apresentado ao pesquisador (nível 5), que decidirá se deve autorizar a atualização da data vinculada a ele no banco.

Assim, ao final da inserção de uma obra, cada verbete estará associado à data de sua primeira ocorrência conhecida. Quanto mais obras forem submetidas, mais detalhes sobre uma determinada palavra serão conhecidas, e mais próxima de verdadeira primeira ocorrência a data associada estará.

2.1.3 Sobre a análise etimológica do DELPo

A lexicografia e a definição de lema

Pelo que aponta (Salviano, 2014, p. 20), a lexicografia é o estudo que permitiu o desenvolvimento de métodos e técnicas de produção de dicionário seja ele de que tipo for. Percebe-se que o trabalho de elaboração de dicionários dependende por completo dos estudos desta área.

Para este estudo, o *lema* é uma unidade básica provida de significado. Dessa forma, num determinado dicionário, todo signo que seja objeto de definição ou explicação é um lema. Pode-se dizer que os verbetes descritos num dicionário são lemas. "Algoritmo" pode ser considerado um lema, por exemplo.

Interesante pontuar o que disse Viaro (2017, p. 146) sobre duas palavras escritas da mesma forma poderem ser um lema ou dois. No primeiro caso, o fenômeno seria fruto de homonímia (timilogias distintas). No segundo caso, a polissemia seria responsável por isso.

Definição de novos termos para o tratamento computacional dos dados

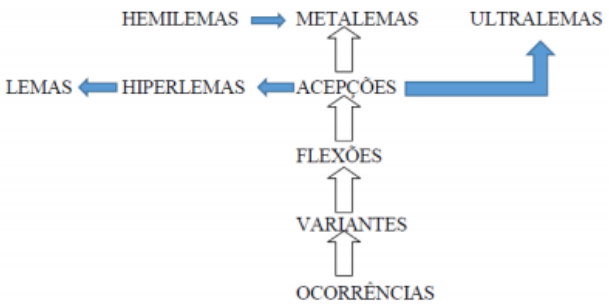


Figura 2.1: *Hierarquia de dados do DELPo*

Acima enconrtam-se termos criados com o único intuito de facilitar o tratamento computacional dos dados a serem manipulados.

O **Metalema** se trata de um conjunto de letras único e dotado de significado, independete da quantidade de acepções a que corresponda. Não há metalemas iguais. Isso o diferencia de um lema lexicográrifico tradicional pois, como já mencionado, uma palavra pode estar associado a um ou dois lemas.

Veja um exemplo: a palavra "manga" pode estar associada a dois lemas ($manga^1$– fruta, $manga^2$–parte do vestuário). No entanto, não é possível para o moedor do DELPo separar esses caracteres em dois lemas distintos, dado que o programa deveria associar os devidos significados a cada palavra automaticamente. Para o sistema, a fruta e a peça do vestuário

[6] Dada duas datas d_1 e d_2, diz-se que $d_1 > d_2$ se, e somente se, d_1 é posterior a d_2

são classificadas como um único metalema. A atribuição de sentido aos verbetes, por outro lado, deve ser feita posteriormente por um pesquisador.

Para acharmos o metalema de uma palavra adota-se a mesma metodologia que a lexicografia tradicional adota para o lema. Ou seja, um metalema

- de um verbo é sua forma no infinitivo (cantava → cantar)
- de um substantivo é sua forma no singular (mesas → mesa)
- de um adjetivo é sua forma no masculino singular (bonitas → bonito)
- é sempre adotado em sua ortografia atual (heróico → heroico)

Hemilema é uma palavra que precisa da interpretação da acepção de um metalema para ser compreendida. É a denominação usada ao tratar de abreviaturas e palavras truncadas devido a fatores tais quais o mau estado de conservação das obras.

Tomemos, como exemplo, `mtos` e `mta`. Ambas são interpretadas como uma das acepções do metalema `muito` e portanto não remetem a `muitos` e `muita`, suas respectivas flexões.

Convém observar que uma sigla, ainda que se trate de um tipo singular de abreviatura, será tratada como metalema ao invés de um metalema. Isso acontece pois há siglas que se assemalham a nomes comuns[7] e só um pesquisador conseguiria classificá-las da forma apropriada.

O conceito de **Hiperlema** foi criado com a finalidade de recuperar as condições de homonímia e polissemia, ausentes na constituição do metalema. Para que entendamos melhor como o conceito de hiperlema é aplicado, é necessário definir alguns termos referentes às acepções.

Uma acepção é subordinada se remete a uma outra acepção, chamada subordinante. Se uma acepção subordinante não for ao mesmo tempo subordinada, ela é chamada de acepção principal. Toda acepção principal gera um hiperlema. Um metalema, por sua vez, pode ter duas ou mais acepções principais e, nesse caso, gerará hiperlemas homônimos, o que nos possibilita fazer a distinção entre homonímia e polissemia de um único metalema.[8]

Ultralema é um conceito pensado para tratar radicais, raízes e elementos de formação de uma palavra (como prefixos e sufixos). A palavra `caligrafia`, por exemplo, possui dois radicais (*cali* e **grafia**) e, portanto, dois ultralemas – um para cada radical. Já a palavra **bigamia** tem um prefixo `bi-`, e um radical `gamia`, e portanto, também possui dois ultralemas – um para o prefixo e outro para o radical. Eis as formas de um ultralema:

`-x` quando é um sufixo final

`-x-` quando é um sufixo não final (requer uma vogal temática, por exemplo)

`x-` quando é um prefixo

`x` sem os hifens, nos demais casos (quando é raiz, por exemplo)

2.2 O Back-end

Antes de entrar no mérito da definição de back e front-end, é conveniente explicar o porquê da separação dos tópicos. Além da obvia separação entre os dois repositórios, deve-se pelo fato de constituirem diferentes camadas do mesmo sistemas. Camadas distintas, desenvolvidas de outras maneiras e com ferramentas distintas.

[7] Crea, por exemplo

[8] Porque precisamos de um hiperlema, então? – Notas de Estudante

```ruby
module AutoriasHelper

    def eh_autor?(obra, autor)
        Autoria.exists?(autor_id: autor, obra_id: obra)
    end

    def pega_autoria(obra, autor)
        Autoria.where(autor_id: autor, obra_id: obra).first
    end

end
```

Figura 2.2: *Consegue entender o que está sendo feito?*

O back-end, a camada mais funda do portal, é onde os dados são inseridos, consultados e removidos. Esta camada, ao contrário da outra, não fica acessível ao cliente, que dificilmente compreenderia o que está se passando.

2.2.1 Ruby

Ruby é uma linguagem de programação orientada a objetos criada por Yukihiro Matsumoto em 1993. Ruby pode ser considerada como que um encontro entre Python, Perl e Smalltalk. Por ser uma linguagem tão versátil e permitir que o desenvolvedor concentre melhor suas energias em fazer o algoritmo[9] como disseram Carlson e Richardson (2009), e pelos motivos citados abaixo, escolheu-se Ruby para o desenvolvimento do back-end.

Ruby apresenta uma notável facilidade em lidar com expressões regulares e com texto, fator de grave importância para a produção do moedor (funcionalidade principal do sistema). Além disso, programar em Ruby é um exercício menos cansativo do que com boa parte das outras linguagens, muito em razão de alguns de seus métodos ou palavras chaves tornarem o texto do arquivo fonte mais compreensível para humanos e máquinas exercitando a sugestão de D. E. Knuth que recomenda que a programação deva ser feita de modo que humanos entendam o que se pediu da máquina.

Um arquivo bem formatado em Ruby permite que tal recomendação seja posta em prática. Veja, na figura 2.2 um exemplo tirado do módulo auxiliar de `autorias` (Entidade intermediária entre autor e obra[10]). Além da obra de Carlson e Richardson (2009), o site **Tutorials Point** também é uma boa sugestão quando se precisa de alguma ideia, ou em momento de dúvida.

O livro, tal como o título e os próprios autores (Carlson e Richardson, 2009, xvii) sugerem, é um livro de receitas. Apresenta soluções para prblemas comuns e pequenos tutoriais. A página, como o nome diz, é um compilado de tutoriais e é perfeito para quando há dúvidas pontuais. A figura na sessão sobre RSpec é um print tirado da página.

Quando a pesquisa visa compreender os limites do Ruby e do Rails, olhar a documentação oficial é a melhor solução. Por se tratar de múltiplas páginas, não achei conveniente colocar todas na bibliografia, senão o guia do Rails. No entanto, uma pesquisa com as palavras chave "ruby", o nome da classe a explorar e "docs" ajudam muito, também. Pesquisar por gems pra fazer algo por você também gera ótimos resultados (foi assim que encontrei uma gem que transforma núeros romanos em decimais e vice-versa.)

[9] ao invés de ficar lutando contra a programação

[10] muitos para muitos

2.2.2 Rails

Segundo (Carlson e Richardson, 2009, p. 613), Ruby on Rails (ou só Rails) é o que há de melhor no universo Ruby, e ainda que ele não reinvente a roda em algum aspecto (utilizando-se do que já foi verificado e validado como o próprio Active Record[11] e o paradigma MVC – Model, View, Controller).

Sobre o primeiro, convém saber que é o maior, senão o único responsável por "proteger" o desenvolvedor de dores de cabeça com scripts de migrações e pesquisas e views nos bancos de dados. Não que o desenvolvedor não possa colocá-las ali, mas dificilmente haja uma pesquisa que não se possa fazer sem a sua interface de busca. É um grande exemplo do que foi dito no início da sub-seção 2.2.1 sobre preocupar-se mais com o algoritmo a ser desenvolvido que com a linguagem em si.

O paradigma MVC é o conceito principal ao se tratar de Rails, ainda que não seja exclusivo de ferramenta alguma, podendo mesmo ser aplicado a sistemas heterogêneos (leia-se que utilizam mais de um framework). A sigla junta as iniciais das palavras *model*, *view* e *controller*.

model realaciona os dados, cuida das validações a nível de back-end, transações, e por aí vai. O sistema conta com o *ActiveRecord*, que serve de ponte entre o programador, além de gerar métodos de recepção de dados (*getters*) e de manipulação (*setters*) de acordo com os campos da tabela (figura 2.3).

view é a camada de visualização do Rails. Implementado com a biblioteca *ActionView*, um sistema baseado em ERb[12] para apresentar as views.

controller fica no meio dos dois anteriores, e também é responsável por lidar com as requisições. De um lado, pega os dados necessários, de outro, organiza-os e manda pras views. É responsabilidade do *ActiveController*

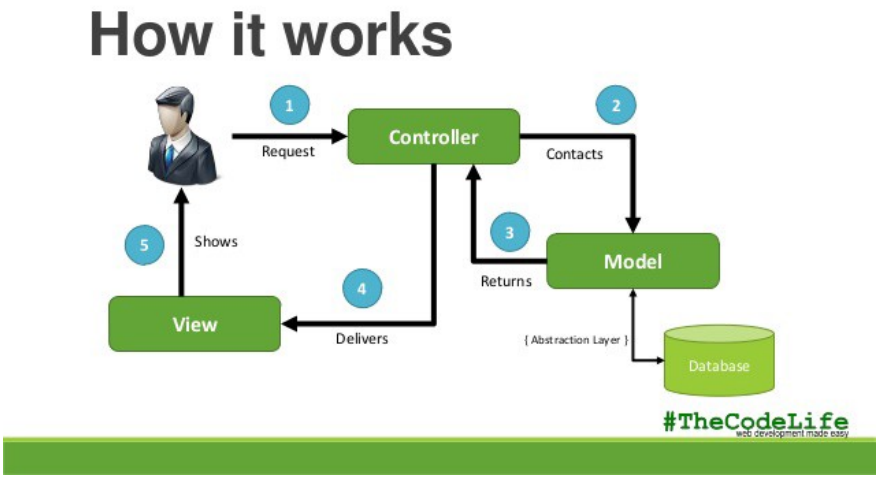


Figura 2.3: *Como funciona uma aplicação MVC (fonte)*

Outro ponto muito forte (e ao mesmo tempo, um calcanhar de aquiles[13]) da ferramenta são as convenções que, para o desenvolvedor tradicional de Rails, importam mais que um arquivo de configurações gigante. Seguindo os padrões que a ferramenta estipula também

[11] responsável por mapear os objetos criados no código em modelos e relacionamentos
[12] Embedded Ruby
[13] chegarei aí ao falar do front-end

salva ao desenvolvedor muito tempo que pode ser usado para a criação e planejamento de novas funcionalidades.

O "ponto de máximo" do Rails, seu grande destaque, vai para o desenvolvimento de aplicações que façam CRUD (create, read, update e delete) de modelos em geral. Começa-se com um modelo conceitual em mente, e pouquíssimo tempo depois há um produto mínimo viável. Ideal quando se deseja fazer determinada coisa a toque de caixa.

Atualmente, vem sendo usado bastante no modo API (modo escolhido para esta aplicação). Usar esse modo significa que sua aplicação serve JSON (Javascript Object Notation) ao invés de HTML. Além disso, a aplicação em modo API (não só em Rails, como qualquer outra linguagem e framework) permite que as respostas sejam coletadas por várias outras aplicações, e essa é só uma das vantagens de usar o Rails desta forma. O guia do modo API fornece algumas outras como a proteção a ateques como o de IP Spoofing.

2.2.3 RSpec

Provar que um método funciona 100% das vezes requer, por vezes, uma prova formal. Testes, de qualquer natureza (seja de integração ou unitário), têm como finalidade garantir que o método vai dar a resposta coreeta nos cenários apontados.

Vale dizer também, que em um sistema robusto e cheio de funcionalidades, provar formalmente que tudo está em perfeito estado é um trabalho hercúleo e cansativo sem contar que é um grande redutor de produtividade. É por isso que que se faz necessária a construção de testes. Praticamente todo aluno de computação passou por dificulades ao caçar problemas em trabalhos no início de sua graduação. Estes provavelmente saberão melhor o quanto convém ter um conjunto de testes para as funções criadas.

Para produzir os testes que garantem, na medida do possível, a consistência e o funcionamento completo do portal, esse projeto conta com RSpec. Diferentemente de outras ferramentas de teste, o seu foco é no comportamento das funções[14] como podemos observar na figura2.4. Trata-se da ferramenta mais adequada para garantir que os métodos e requisições a serem desenvolvidos seguirão o comportamento esperado.

```
class ClassRoom
   def initialize(students)
      @students = students
   end

   def list_student_names
      @students.map(&:name).join(',')
   end
end

describe ClassRoom do
   it 'the list_student_names method should work correctly' do
      student1 = double('student')
      student2 = double('student')

      allow(student1).to receive(:name) { 'John Smith'}
      allow(student2).to receive(:name) { 'Jill Smith'}

      cr = ClassRoom.new [student1,student2]
      expect(cr.list_student_names).to eq('John Smith,Jill Smith')
   end
end
```

Figura 2.4: *Exemplo de programa e teste respectivo*

[14] BDD – Behavior Driven Development

2.3 O Front-end

Como havia dito na seção 2.2.2, há um ponto negativo no Rails relativo à "filosofia" do Rails em adotar muitas convenções. Se por um lado isso realmente facilita a vida do programador, por outro lado, pode impossibilitá-lo de fazer determinadas coisas. Em outras palavras, pode-se dizer que Rails é maravilhoso quando se seguem os padrões, mas pode virar uma dor de cabeça se quiser fazer as coisas da forma como o desenvolvedor prefere.

Por exemplo: não seria impossível usar algum framework Javascript para as views do Rails, mas o trabalho necessário para produzir uma solução adequada é muito maior que usar o Rails como API e fazer uma aplicação em Nuxt.JS a parte.

2.3.1 Vue JS

Segundo o guia oficial, Vue (prununciado como $/vju/ \rightarrow$ **view**), é uma ferramenta voltada à produção de interfaces de usuário, sendo este o seu foco. No vídeo exibido na página principal do Vue.js, podemos ver alguns de seus traços básicos.

O principal forte é versátil. Mas em que sentido? A depender do que você precisa do Vue, há uma receita do que pode ser feito. Quer fazer um site estático? É possível. Quer uma ferramenta que complemete um back-end (produzindo uma produção fullstack)? Dá pra fazer, também. Vue é uma "massinha de modelar" em questão de ferramenta javascript.

Vue também é uma ferramenta acessível. Além de se tratar de um projeto de software livre, qualquer um que tiver um bom domínio de HTML, CSS e JS podem utilizá-lo. Além disso, o programador que está entrando em uma equipe utilizando Vue provavelmente terá pouco trabalho em se interar do que foi feito. A curva de aprendizado pode ser incrivelmente suave, também.

Além disso, conta com uma grande quantidade de ferramentas afins construidas a partir dessa. Um exemplo ótimo é o **VuePress**, uma ferramenta voltada a produção de documentação. Ao que parece, compila arquivos em *Markdown* e o transformam em uma SPA (Single Page Application)

A figura 2.5 mostra um componente Vue. Observe a separação clara do que é HTML e o que é Javascript. Abaixo da tag template (HTML) e script (JS), há uma tag style (CSS) que pode ser aplicada somente àquele componente ou não.

2.3.2 Nuxt JS – Vue on Steroids

Baseado em seu guia oficial (está na bibliografia), Nuxt é um framework progressivo (ou seja, que pode servir como uma biblioteca javascript ou framework) baseado em Vue.js. Seu foco é o mesmo que o anterior, mas conta, por padrão, com muitas das bibliotecas que você deveria inserir manualmente no Vue. Esse é o motivo de dizer que Nuxt.js é um Vue com esteroides.

Longe de ser apenas uma versão fortificada do framework acima, Nuxt adota soluções para alguns dos seus problemas. Veja alguns desses.

Construir uma aplicação do zero não é trivial, e por isso talvez existam frameworks a facilitar tal tarefa. Vue.js é um bom exemplo, e facilita muito a vida do desenvolvedor. Entretanto, não é tão prático dado que a instalação inicial não inclui peças importantes como o **Vuex** e **Vue Router**. Não partes difíceis de incluir, mas é bem melhor quando elas estão pré-instaladas e configuradas, como vê-se implementado no Nuxt. Vale saber que muitas, senão todas as suas configurações padrão são baseadas no que os desenvolvedores consideram boas práticas.

```
<template>
    <div class="hero is-primary">
      <div class="hero-body">
        <div class="container">
            <h1 class="title">
              {{ phrase }}
            </h1>
            <h2 class="subtitle">
              A line on the myth Chuck Norris
            </h2>
        </div>
      </div>
    </div>
</template>

<script>
export default {
    data() {
        return {
            phrase: ''
        }
    },
    beforeMount: function () {
        this.$axios.get('api/chuck_fact').then(r =>{
          this.phrase = r.data.fact
        }).catch(e =>{
          console.log(e)
        });
    }
}
</script>
```

Figura 2.5: *Exemplo de um componente Vue*

Outra mudança notável é a organização das pastas e arquivos (veja a figura 2.6)

Em projetos Vue (a), temos, por padrão, somente o diretório `src/` concentrando a maior parte dos arquivos relevantes do projeto. Dentro deste, há uma separação maior, mas ainda não dá pra saber que componentes gerarão páginas e quais serão simples componentes que as formarão. No Nuxt (b), não há uma pasta que concentre quase todo o cógigo da aplicação. Ao invés disso, há muitas pastas que abrigam cógigos cuja natureza está descrita pelo nome do diretório.

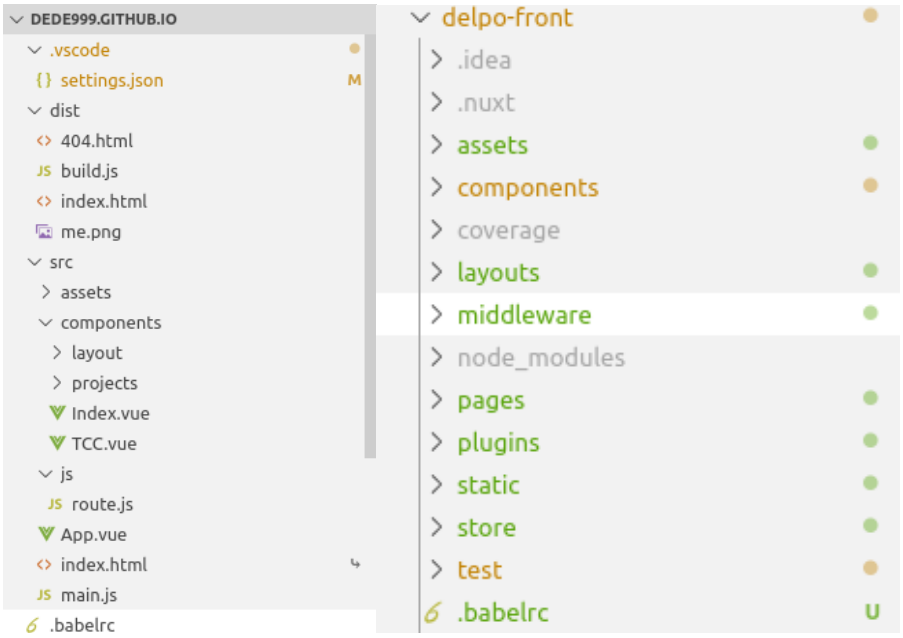
Search Engine Optimization(SEO) é a propriedade de uma aplicação que facilita a indexação e pesquisa de elementos nela presente. Isso pode ser feito em Vue, por exmplo. Mas a solução não é difícil, mas envolve carregar página por página no servidor. Em uma plataforma menor talvez seja viável. Entretanto, quando o sistema ganha em complexidade e capilaridade, torna-se cada vez menos produtivo. O Nuxt é pré configurado para fazer essa ação por debaixo do pano, simplificando o trabalho do desenvolvedor.

Quando uma aplicação Vue.js carrega no servidor, o processo leva um pouco mais de tempo na primeira vez se comparado aos posteriores recarregamentos que sobem somente as mudanças feitas. Isso acontece, porque toda a aplicação é carregada ao mesmo tempo nesta ocasião. No caso do Nuxt, só o HTML e o CSS são pré carregados por padrão. O javascript é carregado por demanda, e isso permite que o carregamento das páginas ocorra de maneira mais rápida.

Para acabar, vale falar um pouco de duas características deste framework:

- Ao contrário de seu predecessor, Nuxt é mais fácil de configurar, e permite que as configurações padrão sejam sobrescritas com menos trabalho.

- Configura rotas de acordo com a organização do arquivos no diretório `pages/`.



(a) *Vue* **(b)** *Nuxt*

Figura 2.6: *Organização de pastas e arquivos*

2.3.3 Jest

Jest, de modo similar a RSpec, é uma ferramenta de testes para Javascript criada pela equipe do Fecaebook, motivo pelo qual alguns acreditam que seja uma biblioteca exclusiva do React (framework javascript também desenvolvida pelo Facebook), o que não é verdade.

É uma ferramenta performática, e parte deste bom desempenho se deve ao fato de que cada teste ser rodado num processo distinto rodando em paralelo com os demais. Isso é possível, pois cada teste tem o seu "contexto global" garantindo independêcia entre cada um. Para aumentar ainda mais a performance, testes que falharam são executados primeiro. Além disso, os testes são organizados baseado no tempo necessário a executá-los.

A ferramenta verifica a cobertura de testes na aplicação. Em outras palavras, iserindo um argumento de linha de comando[15] **`-- --coverage`**, é possível que se veja o quanto de cada parte do código é verificado através dos testes. Pode-se questionar a importância dessa funcionalidade. Entretanto, saber o que já foi coberto é importantíssimo para dirigir os esforços na direção certa. Importante saber que a cobertura de testes será exibida na sessão 4.1.5 do capítulo dos resultados obtidos.

Outras características bem interessantes do Jest:

boas falhas Jest provém o desenvolvedor com mensagens significativas a cerca do motivo daquele teste ter falhado, o que é crucial para eliminar o erro.

configurações Jest pode rodar com pouca ou nenhuma configuração.

mocking com Jest, é possível simular qualquer objeto fora do escopo do teste.

[15] Em aplicações Nuxt, essa verificação é feita sem a necessidade de argumentos adicionais

```
import { shallowMount } from "@vue/test-utils";
import App from "../src/App";

describe("App.test.js", () => {
  let cmp;

  beforeEach(() => {
    cmp = shallowMount(App, {
      // Create a shallow instance of the component
      data: {
        messages: ["Cat"]
      }
    });
  });

  it('equals messages to ["Cat"]', () => {
    // Within cmp.vm, we can access all Vue instance methods
    expect(cmp.vm.messages).toEqual(["Cat"]);
  });

  it("has the expected html structure", () => {
    expect(cmp.element).toMatchSnapshot();
  });
});
```

Figura 2.7: *Exemplo de um conjunto de testes usando Jest*

2.4 Boas práticas em programação

Na sesão 1.2, falou-se da importância de um código limpo. Aqui tratar-se-á de como produzir código de qualidade. Para isso foi trazido uma publicação de Schmitz (2019) que cita práticas que enriquecem a qualidade do código. Aqui citarei os pontos mais importantes.

2.4.1 SOLID

SOLID é uma sigla formada pelo título de cada um dos 5 princípios componentes.

S Single Responsibility Principle → Cada classe ou função deve ter uma responsabilidade única no sistema – O importante é que classes e funções tenham uma tarefa

O Open-closed Principle → Objetos e entidades devem ser abertos a extensão, e fechados a modificação – Em outras palavras, sempre que algo novo precisar ser criado, o correto é criar uma função nova, e não alterar o que já foi feito.

L Liskov Substitution Principle → Se a classe B herda da classe A, então você deve ser capaz de usar B no lugar de A sem quebrar a funcionalidade – Dado que haja uma classe **`Animais`** que possua um método **`andar`** que retorne as novas coordenadas do bicho. Se houver uma outra classe **`Felinos`** que herde da primeira, e que esta também possua o método **`andar`**. Neste caso, o método deve retornar a mesma coisa, ou seja, as novas coordenadas do felino.

I Interface Segregation Principle → Desenvolvedores não devem implementar métodos de interface que não vá usar – Se há métodos que uma interface não utiliza, ela tem que ser quebrada em mais.

D Dependency Inversion → Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações – Uma classe deve ser a mais abstrata possível, e não deve depender de outras classes

2.4.2 Outras recomendações

Além desses preceitos, há outros pilares a levar em conta, e quando seguindos, melhoram e muito a qualidade do código. Eis alguns desses presentes no texto de Schmitz (2019)

- Use nome de variáveis que tenham significado e que sejam fáceis de pronunciar. Nomes como **`i`**, **`x`** e **`item`** são contra exemplos fortíssimos.
- Ainda sobre variáveis, é aconselhado que os nomes auto explicativos, de forma que comentários não sejam necessários. Ex: **`SaveRecord`** → **`SavePerson`**
- Faça uso extensivo de constantes. Ex: **`if age >= 18`** → **`if age >= LEGAL-AGE`**
- Evite funções com mais de um `if` encadeado.
- Evite código duplicado.

Há outros mais, e vale dizer que muitos desses pilares estão mencionados no texto de Lukaszuk (2018), como o que se refere a nomes de variáveis. Como já foi dito antes, dada a multiplicidade de diretrizes, não é trivial que todas sejam seguidas. O importante é ter uma ampla noção o que é um código de qualidade

Capítulo 3

Sobre o Desenvolvimento

Neste capítulo

3.1 Arquitetura de projeto 18

3.1.1 Por que as ferramentas descritas foram utilizadas ao invés de outras similares? 18

3.1.2 Por que mudar a implementação do front? Por que usar o modo API? 18

3.1.3 Por que refatorar o banco de dados e outros setores da aplicação? . 19

3.1.4 Como será o desenvolvimento do front end? 19

3.2 Decisões de código 19

3.2.1 RR – Na sessão anterior foi mencionado que o banco de dados foi alterado. Que alterações são essas, e por que foram feitas? 19

3.2.2 N – Por que a implementação do design pattern Builder foi abandonada na criação de formulários no front-end? 21

3.2.3 N – Existe uma quantidade considerável de código repetido no front. Isso não é algo ruim? 22

— Introdução —

Neste capítulo encontram-se reflexões spbre decisões de projeto que foram tomadas e uma reflexão sobre cada uma. A ideia é que, ao terminar este capítulo, que se possa agrugar as reflexões em temas, de forma que o conteúdo fique bem organizado. Pode ser que o capítulo ganhe uma feição de página de perguntas e respostas, e isso é proposital.

3.1 Arquitetura de projeto

3.1.1 Por que as ferramentas descritas foram utilizadas ao invés de outras similares?

No caso do back-end, o Ruby on Rails fora utilizado por alguns motivos. O primeiro, e mais importante é a continuidade. Como é de conhecimento entre aqueles com maior participação no projeto, o DELPo já foi desenvolvido em PHP, algumas partes em Pearl, depois Django, e agora em Rails. Muita migração já aconteceu, e outra estaria praticamente fora de cogitação.

Outro motivo é o fato de que a fluência do aluno em Rails é infinitamente maior em Ruby que Python. Além do mais, não é só o desenvolvimento que importa: os testes são cruciais, e não há uma ferramenta de testes com que o aluno esteja perfeitamente familiarizado. Eu precisaria passar por outra curva de aprendizado que, pequena ou grande, representaria um tempo perdido para essa aplicação em específico.

A fluência e experiência no uso de RSpec também foram fatores determinantes na escolha desta ferramenta. A escolha do Vue e Nuxt.js também se deve a esses fatores.

Não há muito que se possa falar sobre Angular.js, dado que pouco conheço da ferramenta. Aprender sobre a ferramenta enquanto a aplicação é feita parece uma ideia pouco sábia a nível estratégico. O aluno responsável por este trabalho conhece um pouco mais sobre React.js, e isso já é suficiente para gerar um desconforto e rejeição em relação ao framework. Tal rejeição se deve ao fato de que ainda que Vue e React sejam parecidos em alguns aspectos, o código em um componente do último não possui a organização estrutural que o Vue tem. Essa organização, chamada de *Code Splitting*, além de tornar o código mais legível, torna possível que algumas partes do componente sejam pré carregadas e outras sejam carregadas por demanda.

Entre as bibliotecas de testes baseadas em Javascript, escolheu-se Jest. O motivo dessa esolha é simples: no site onde o aluno busca tutoriais, a ferramenta de testes abordada é esta. A acessibilidade da ferramenta foi o motivo principal da escolha.

3.1.2 Por que mudar a implementação do front? Por que usar o modo API?

Muita coisa que estava pronta teve que ser alterada. Outras tiveram que ser abandonadas. Que motivação levaria a um aumento de trabalho a ser feito?

Primeiramente, quando o Rails foi criado em 2005, não havia tantas ferramentas de Javascript como há hoje (junto com as mencionadas acima, ainda há Ember.js, Backbone, e outras tantas ...). É uma ferramente pra entregar produtos minimamente viáveis em pouco tempo. Com o surgimento de frameworks voltados à preparação da camada de apresentação, usar o Active View tournou-se cada vez menos interessante.

Outro motivo é a possibilidade de construir uma interface mais agradável e dinâmica para a aplicação. Pode não parecer o melhor dos motivos, mas justamente pelo fato dessas ferramentas "js" serem voltadas à construção da camada de apresentação é que o resultado esperado é bem melhor.

Além disso, desenvolvendo a aplicação Rails em modo API, permite-se separar quem atende o cliente (o back-end), de quem o serve (front-end), delimitando tarefas e papeis. Facilitando aos próximos desenvolvedores se concentrarem em apenas um lado da aplicação

De fato, a quantidade de trabalho aumentou. Nem tanto pela adoção do modo API que não dá muito trabalho, mas por razões que envolvem a próxima pergunta.

3.1.3 Por que refatorar o banco de dados e outros setores da aplicação?

Além da mudança de modos, a necessidade de reescrever controladores e modelos foi o maior motivo. O fato de não compreender completamente algumas decisões de projetos dos alunos que tomaram o projeto no ano seguinte me fizeram tomar a decisão de criar uma segunda branch na qual eu deletei controladores, modelos e as tabelas para fazer tudo de novo. Parece um esforço tolo, mas era necessário dado o travamento advindo da incompreensão de partes da aplicação.

Mas por que não consultar os alunos do ano passado? Isso foi feito, mas de forma limitada porque o responsável atual só tem contato recorrente com um deles. Este me ajudou no que pode. Mas haviam outras dúvidas, e com elas eu não poderia garantir a qualidade do código, e não conseguiria fazer os testes (não é possível testar o que não se conhece)

Os controladores, pensados para fornecer dados para construir as páginas ERB tinham, por vezes, mais de uma responsabilidade, o que não é uma coisa boa, mas naquele contexto dava pra entender. Ao mudar para o modo API, foi possível limpar os controladores com a ajuda de uma gema *Active Model Serializer*, que permitiu buscar as informações de uma forma mais organizada.

Outra coisa que precisou ser acertada foi o link entre as entidades. Ainda que a conexão a nível de SQL estivesse correta, as funcionalidades do framework não estavam sendo plenamente aproveitadas, gerando necessidade de código que não precisaria estar lá. Era necessário entender os modelos, o que fazia cada um. Dessa forma, não só os encaixes ficariam melhor, como os testes seriam mais fáceis de fazer.

3.1.4 Como será o desenvolvimento do front end?

O desenvolvimento do front será feito em Nuxt.js. O foco principal é na construção do moedor e do sistema de login. Como nem todos os controladores foram construídos, trabalhar-se-á nas duas pontas (Front e Back). Nesta fase, o TDD será deixado um pouco de lado, para que o desenvolvimento seja mais rápido. Recomendações de estilo de dódigo, como as vistas neste link serão seguidas na medida do possível.

Outra coisa que será observada no desenvolvimento será as regras de bom código, como a responsabilidade única, por exemplo. Para aplicar este princípio, a ideia é criar componentes que exibam um tipo de dado ou que tenha uma função específica. Trata-se, é claro, da situação ideal, e do desejo do aluno em questão.

3.2 Decisões de código

Antes de começar a falar sobre isto, fica a conhecimento do leitor a seguinte convenção:

RR Decisões sobre o projeto Ruby on Rails

N Decisões sobre o projeto Nuxt

3.2.1 RR – Na sessão anterior foi mencionado que o banco de dados foi alterado. Que alterações são essas, e por que foram feitas?

Sim, algumas mudanças foram feitas no banco de dados. Não foram miutas, mas foram essenciais. A cada entidade recriada, o modelo era construido, as validações eram construidas

de acordo com os modelos originais. Além das validações, as ligações entre modelo eram reforçadas por meio de palavras chave que auxiliam o mapeador objeto- relacionamento. Tudo isso quase sempre vinha depois de construir os testes relativos àquele modelo. Tendo dito isso, pode-se mencionar as mudanças principais no banco de dado.

Acepções – Histórico Acepções

Antes de começar a falar sobre essas entidades, vale saber que à dupla **Flexão** e **Histórico de flexões**, aplica-se o que será dito aqui.

No caso dessa entidade, não era trivial o que ela fazia. Conversando com o professor, percebi que a segunda entidade funcionava tal qual uma lista ligada simples onde o mais novo ligava-se ao mais velho, e assim vai. O lado bom de fazer isso é que manter o registro de que pesquisador colocou uma determinada acepção. Além de ser fácil remover, caso necessário for, pode ser fácil de averiguar a cronologia das acepções. A cabeça da lista era o mais recente, e a instãncia na tabela acepção tinha uma chave estrangeira para essa cabeça. Veja como estava implementado.

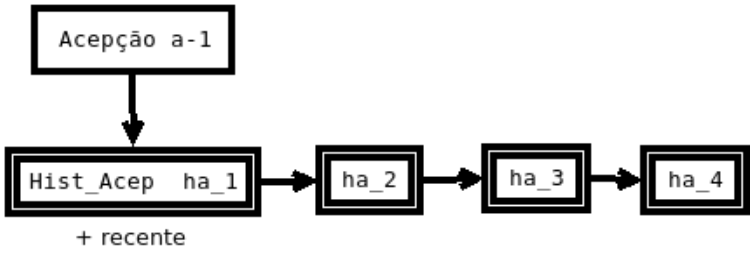


Figura 3.1: *Como estavam implementadas as acepções*

Mas havia um problema nessa implementação. O campo de ligação entre **Acepção** e **Histórico Acepção** só ajuda a verificar o mais atual. Para descobrir o meis antigo (para achar o *Terminus a quo* – taq), é necessário fazer de 0 a n operações `JOIN` aumentando muito o custo da pesquisa por esta informação. Trata-se de um potencial garagalo.

Antes de pensar nas mudanças, bom pensar a razão de se precisar das instâncias mais antiga e mais recente. A importância da mais antiga se deve ao fato de que, através dela, pode-se conhecer a data aproximada ou real da sua primeira aparição. A mais nova é importante pois através dela é que se determina a grafia atua da acepção.

Para refatorar a entidade, retirou-se o campo `:histacepatual_id` da classe **Acepção** e `:anterior_id` de **Hist_Acepção**. Além disso, criou-se uma referêcia da primeira na segunda classe, e uma chave na classe de acepções para a mais antiga e mais recente. As mudanças na prática podem ser vistas na tabela abaixo.

Justificativas:

- Acessos

  1, 2 e 4 Há uma chave que permite o acesso direto a essa instância

  3 É necessário percorrer a lista inteira para encontrar a instâcia

- Inserção

  5, 6 e 8 Há uma chave que permite o acesso direto à instância a ser comparada

  7 É necessário percorrer a lista para saber onde será colocado o próximo emelemnto

- Remoção

		Desempenho	
operação	frequência	antes	depois
acessar a mais recente	alta	$O(1)$[1]	$O(1)$[2]
acessar a mais antiga	alta	$O(n)$[3]	$O(1)$[4]
Adicionando uma entrada em **Hist_Acepção**			
atualizar mais recente	média-alta	$O(1)$[5]	$O(1)$[6]
atualizar a mais antiga	média-alta	$O(n)$[7]	$O(1)$[8]
Removendo uma entrada em **Hist_Acepção**			
atualizar mais recente	baixa	$O(1)$[9]	$O(n * lg\ n)$[10]
atualizar a mais antiga	baixa	$O(n)$[11]	$O(n * lg\ n)$[12]

Tabela 3.1: *Comparativo do desempenho entre as versões antes e depois da refatoração Desempenho e frequêcia são estimados com base no fluxo conhecido de operação*

9 Há uma chave que permite o acesso direto à instância a ser comparada, após a remoção, conecta-se o anterior à acepção

11 É necessário percorrer a lista inteira para chegar no último elemento, deletá-lo, e remover o seu registro do novo registro mais antigo

10 e 12 Após a remoção, pesquisa-se o elemento mais antigo ou recente. Isso depende da ordenação do histórico de acepções. Não se sabe em que algoritmo se baseia a ordenação do Ruby. Supondo que seja baseado em comapações, com certeza roda em $O(n * lg\ n)$

Porque tirar o nome e o taq da tabela de acepções? No `README.md` do projeto em Rails, sessão de perguntas, foi feita uma reflexão similar.[1] Nesta ocasião, a resposta foi "Depois de reflexões, ficou claro que a definição pode ser alterada a cada inserção ou remoção de lemas. Gerar demanda de mais uma ou duas pesquisas, ainda que simples, constitui um gargalo tolo, e por isso a definição de flexões ou acepções deve ficar nas tabelas principais". O erro principal desse raciocínio é o que uma ou duas pesquisas a mais provavelmente não serão responsáveis por um gargalo tão grande assim. Além do mais, a implementação pode diminuir esse impacto.

Um outro fator a considerar é Lei de Demeter. Segundo ela, é importante que uma classe saiba muito pouco ou quase nada das outras classes. Utilizar Esse princípio não impediria que se colocasse esses campos na tabela de acepções. Entretanto, ao implementar dessa forma, o projeto fica menos acoplado, e portanto, mais flexível e menos difícil de escalar.

Nas operações de remoção de históricos de acepções o desempenho médio é pior considerando-se a solução nova. Isso não pode ser um problema? Não, porque são operações que tiveram um desempenho estimado pior são as mesmas operações apontadas como menos frequentes pelo cliente, enquanto as operações mais repetidas tiveram um melhoramento.

3.2.2 N – Por que a implementação do design pattern Builder foi abandonada na criação de formulários no front-end?

A ideia inicial no que diz respeito aos formulários, é que haveria uma classe Javascript que geraria os formulários. Mas por que fazer assim e não um formulário por vez?

[1] A pergunta foi sobre a definição da acepção e da flexão

O motivo é que fazer vários formulários pode resultar em repetição de código. Fazer isso não é pecado, mas reduz a qualidade do código. Além do mais, código repetido pode requerer testes com mais código duplicado, reduzindo a ainda mais a qualidade do código, sem mencionar a necessidade de se replicar determinadas mudanças.

Exite uma troca inerente no processo. Buscar um sistema mais enxuto e esclável implica, na maioria das ocasiões, em um aumento (considerável ou não) de complexidade. Entretanto, uma vez implementado um construtor de formulários, pode-se simplesmente chamar o componente com o tipo específico de formulário a se criar. Essa ideia veio de uma palestra sobre como aplicar alguns design patterns no desenvolvimento com Vue.js. O vídeo desta palestra pode ser encontrado no YouTube [2], no site VueMastery, aba Conference Videos → VueConf US 2019, e neste link se precisar de legendas para o vídeo.

Por que então os formulários não foram implementados desta forma então? Ao tentar implememntar uma solução similar, tive dificuldade, pois não sabia como passar os dados para a requisição HTTP para o backend. Para evitar maiores atrasos, decidi criar múltiplos componentes. Passado o prazo do TCC, é uma coisa que adoraria fazer com mais calma.

3.2.3 N – Existe uma quantidade considerável de código repetido no front. Isso não é algo ruim?

De fato. Em diversos momentos do desenvolvimento foi dito que repetir código era uma má prática, e não deixou de ser. O problema, entretanto, foi conciliar a vontade de fazer a melhor aplicação, com o melhor cógigo, o mais limpo possível, com a necessidade do DELPo de estar de pé. Não posso faltar com o respeito com os integrantes do NEHiLP, e entregar algo que não funcione, mas também não posso escrever um código porco que a curto prazo nenhuma viva alma seria capaz de compreender. Provavelmente é uma das coisas em que mais preciso amadurecer, e pude aprender um pouco mais com esse projeto.

[2] O título do vídeo é “Phenomenal Design Patterns In Vue with Jacob Schatz”

Capítulo 4

Dos Resultados Obtidos

Neste capítulo

4.1 O que foi feito? 24

4.1.1 API Rails 24

4.1.2 Interface 24

4.1.3 O moedor 24

4.1.4 Vendo as coisas funcionar 24

4.1.5 Testes e cobertura 24

4.2 O que será feito? 24

4.2.1 Refatorações 24

— Introdução —

Neste capítulo serão mostrados os resultados deste trabalho. Mas como fazer isso se capturas de tela não são capazes de mostrar a aplicação funcionando ao vivo? A ideia é ilustrar as funcionalidades principais enquanto o link da aplicação em produção ficará diponível aqui.

Outro ponto importante a mostrar aqui é a cobertura de testes. A cobertura do frot não será tão extensiva quanto a do back, mas ao comparar com o que se tinha no começo do ano, é infinitamente maior.

4.1 O que foi feito?

4.1.1 API Rails

4.1.2 Interface

4.1.3 O moedor

Como foi dito na sessão 1.3, o moedor é um punhado gigantesco de código com uma necessidade enorme de ser fefatorado. Não porque é ruim ou está incorreto, mas por estar extremamente confuso, com as responsabilidades misturadas, funções enormes[1]. A função `initialize` também precisam de modificações, mas fica para a list de refatorações a fazer.

A ideia é mover para os *helpers* tudo exceto pela função de inicialização e de leitura. O que for para os helpers, será testado.

4.1.4 Vendo as coisas funcionar

Link da aplicação em produção

4.1.5 Testes e cobertura

4.2 O que será feito?

4.2.1 Refatorações

[1] Sendo a maior a `processa_contextos` com exatas 962 linhas à época da contagem

Capítulo 5

Das Conclusões

Neste capítulo

5.1 A história deste trabalho 25
5.2 O que aprendi com esse trabalho 25
5.3 Legado Esperado 25

5.1 A história deste trabalho

Professor, devo começar desse ano? Ou posso falar de toda a minha saga desde aquele plano com a FGV?

5.2 O que aprendi com esse trabalho

5.3 Legado Esperado

Referências Bibliográficas

Jes() *Jest Homepage*. Citado na pág. 14

MN2(2015) *Manual do NEHiLP*. 2.1 edição. Citado na pág. 1, 2, 3

Nux() *Nuxt.JS Guide*. Citado na pág. 1

RoR() *Ruby on Rails Guides*. versão 6.0.0. Citado na pág. 10

Vue() *Vue.JS*. Citado na pág. 12

Carlson e Richardson(2009) Lucas Carlson e Leonard Richardson. *Ruby Cookbook*. O'Reilly Media, 1ª edição. Citado na pág. 9, 10

Lukaszuk(2018) Wojtek Lukaszuk. Clean code. https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29, 2018. Citado na pág. 2, 16

Salviano(2014) Barbara Neves Salviano. O uso do dicionário de língua como instrumento didático no ensino de língua portuguesa para alunos surdos: em busca de um bilinguismo funcional. *http://hdl.handle.net/1843/MGSS-9LZNZ9*. Citado na pág. 7

Santin *et al.***(2018)** Adriano Tetsuaki Ogawa Santin, Luiz Fernando Antonelli Galati e Mauricio Luiz Abreu Cardoso. *Refatoração do Projeto Delpo*. IME-USP. Citado na pág. 2

Schmitz(2019) Daniel Schmitz. Boas práticas. https://gist.github.com/danielschmitz/95c6eb40a3845f89498a3c748e932f44, 2019. Citado na pág. 15, 16

Viaro(2017) Mário Eduardo Viaro. O dicionário etimológico da língua portuguesa (delpo): conceitos de metalema, hemilema, hiperlema e ultralema. http://siba-ese.unisalento.it/index.php/dvaf/article/view/17775/15134, 2017. Visitado a 9/10/2019. Citado na pág. 3, 7