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

Fabio Brzostek Muller

Desenvolvimento de Interface em Python/Django para o NEHiLP

São Paulo
Dezembro de 2017

Desenvolvimento de Interface em Python/Django para o NEHiLP

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Marco Dimas Gubitoso

São Paulo
Dezembro de 2017

Resumo

O Núcleo de Estudos Históricos da Língua Portuguesa (NEHiLP) é um grupo vinculado à Faculdade de Filosofia, Letras e Ciências Humanas da Universidade de São Paulo (FFLCH-USP) que tem como um dos principais projetos e objetivos a criação de um Dicionário Etimológico da Língua Portuguesa (DELPo). Para facilitar o trabalho dos pesquisadores envolvidos no projeto, foi desenvolvido a partir de 2013 um sistema web em que é possível inserir, visualizar e analisar dados relacionados à pesquisa. O sistema, com código em PHP e Perl, se beneficiaria muito de uma reformulação, já que diferentes partes dele foram feitas por pessoas diversas e em vários períodos de tempo, misturando muitos estilos e ideias distintas quanto à organização do sistema. Assim, este trabalho de conclusão de curso tem como objetivo iniciar um processo de reescrita do sistema do NEHiLP. Para a nova versão, escolheu-se usar o *framework* Django, em Python, já que é uma tecnologia bastante moderna e popular, com boas funcionalidades e muitos recursos disponíveis na internet. Para fazer a adaptação, foi decidido primeiro criar os modelos a partir das tabelas do banco de dados atual, depois adaptar as páginas administrativas e de conteúdo. Esses objetivos iniciais foram cumpridos e, como ainda havia tempo, mais alguns dos programas restantes foram adaptados também. Com os modelos já feitos e uma nova organização de arquivos e programas criada e já utilizada, considera-se que foi preparada uma base muito importante para a adaptação futura do resto do sistema. Com a nova versão, a manutenção e o desenvolvimento de novas funções devem ficar consideravelmente mais simples, ajudando bastante os desenvolvedores e contribuindo para a evolução do projeto.

Palavras-chave: sistema web, banco de dados, PHP, Python, Django, etimologia.

Sumário

Capítulo 1

Introdução

O Núcleo de Apoio à Pesquisa em Etimologia e História da Língua Portuguesa (NEHiLP), vinculado à Faculdade de Filosofia, Letras e Ciências Humanas da Universidade de São Paulo (FFLCH-USP), é um grupo que visa à divulgação das pesquisas acadêmicas brasileiras sobre Linguística Histórica, Filologia e Etimologia [1].

Proposto em 2012 e ativo desde 2013, o núcleo tem como um dos principais projetos a criação do primeiro Dicionário Etimológico da Língua Portuguesa (DELPo) [2]. Outro projeto diretamente relacionado é o de Retrodatação, que consiste, basicamente, no estabelecimento das datas das primeiras ocorrências das palavras da língua portuguesa [3].

1.1 Motivação

Os esforços necessários para a realização desses projetos são, além de complexos, multidisciplinares. Assim, em parceria com o Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP), organizada pelo Prof. Dr. Marco Dimas Gubitoso, do Departamento de Ciência da Computação (DCC/IME-USP), foi desenvolvido um sistema de auxílio aos pesquisadores dos projetos de Retrodatação e do DELPo.

O sistema permite que os pesquisadores insiram textos e informações sobre eles e as palavras que os compõem. Além disso, ferramentas permitem a consulta e modificação dessas informações, que são armazenadas em um banco de dados. O banco guarda, por exemplo, detalhes sobre obras inseridas, palavras, classificações delas e variações e relações entre elas. Para permitir o acesso e a modificação dessas informações, a interface faz consultas complexas ao banco.

O projeto está em uma fase estável, com as funcionalidades principais implementadas e funcionando; assim, o trabalho feito no momento é principalmente de correção de *bugs* ou adição de funções mais simples. No entanto, as funcionalidades presentes nessa interface estão implementadas em diversos programas em PHP e Perl, que foram desenvolvidos por várias pessoas e em diferentes períodos. Considerando esses pontos, o projeto se beneficiaria bastante de uma reformulação que deixasse os programas mais simples e mais robustos, facilitando a manutenção do sistema e a adição de novas funcionalidades.

1.2 Objetivos

O objetivo principal do trabalho foi iniciar um esforço de reescrita do código usando Python e o *framework* Django [4], fazendo ao mesmo tempo uma refatoração que, combinada com os benefícios da linguagem e do *framework*, deixará o sistema mais organizado

e moderno. O objetivo não incluiu a conversão do sistema inteiro; a prioridade foi a implementação das funcionalidades mais básicas, principalmente acesso ao banco de dados já existente, seguida da adaptação das páginas de administração e conteúdo e, por fim, de algumas das de busca e edição/inserção, com o entendimento de que, mesmo incompleto, o trabalho será de bastante ajuda ao projeto.

1.3 Métodos

No planejamento inicial, o trabalho foi dividido em três etapas principais. A primeira, de estudo do sistema já existente, se deu por meio da leitura do código fonte e do manual, da conversa com os desenvolvedores e da atuação na manutenção do sistema atual, além do estudo do próprio Django.

A segunda etapa teve como foco a modelagem da estrutura do banco de dados já existente para o esquema de classes utilizado pelo Django, além da implementação de uma interface mínima de inserção e acesso de dados no banco, o que compôs a base da implementação.

Por fim, a terceira etapa tratou do restante da implementação, com o desenvolvimento de páginas e funcionalidades presentes no sistema atual e a definição de passos necessários para se poder de fato usar a nova versão.

1.4 Estrutura do Texto

O texto está organizado da seguinte forma: em primeiro lugar, esta Introdução, contextualizando e explicando a motivação do trabalho, seus objetivos e os métodos usados para atingi-los.

Em seguida, no Capítulo 2, são apresentados conceitos fundamentais para a compreensão do trabalho. Parte deles são conceitos do Django, que explicam resumidamente seu funcionamento e introduzem a terminologia relacionada. A outra parte é relacionada ao NEHiLP, explicando os conceitos e termos usados nos projetos e no sistema.

No Capítulo 3, o sistema atual é apresentado em detalhes. A estrutura do banco e dos programas, as conexões entre diferentes arquivos e as funcionalidades são apresentadas, e os problemas são examinados mais a fundo.

O Capítulo 4 descreve a parte inicial do desenvolvimento do sistema em Django: a criação dos modelos que representam as tabelas do banco de dados e a criação das primeiras páginas como testes para verificar o funcionamento da integração com o banco.

No Capítulo 5, o desenvolvimento das páginas e funcionalidades do sistema é descrito, com explicações sobre o que mudou em relação à versão atual e detalhes relevantes de implementação.

No Capítulo 6, são descritas as funcionalidades e páginas que não foram desenvolvidas e as observações sobre os passos necessários para realizar a transição do sistema atual para o novo. Além disso, são apresentadas ideias sobre como o desenvolvimento do projeto pode prosseguir. Por fim, a conclusão recapitula os pontos principais do trabalho, dando uma visão geral sobre o que estava planejado, o que foi de fato feito e o que faltou e quais são os pontos positivos e negativos.

Capítulo 2

Conceitos

Para compreender o trabalho, é preciso primeiro entender alguns conceitos fundamentais intimamente relacionados a ele. Em particular, é necessário explicar em mais detalhes os projetos e ideias do NEHiLP, e o que é e como funciona o Django, o *framework* usado no desenvolvimento da nova versão do sistema.

2.1 NEHiLP

Desde sua criação em 2012, as atividades do NEHiLP tiveram um grande foco na execução dos projetos de Retrodatação e do DELPo. A pesquisa, feita tanto a partir de textos antigos quanto de atuais, permite o registro das primeiras ocorrências de vocábulos e suas respectivas acepções e de outras informações como a frequência de uso e características sociolinguísticas e estilísticas. Assim, é importante explicar os objetivos dos projetos e os conceitos relacionados a eles e ao sistema do NEHiLP. Informações mais detalhadas estão disponíveis no Manual do NEHiLP [5].

2.1.1 Retrodatação

Para estabelecer etimologias de palavras, um método que pode ser usado por pesquisadores é o de analisar textos, procurar trechos em que determinadas palavras aparecem, e associar essas ocorrências às datas das respectivas obras. No entanto, esta abordagem está sujeita a vários tipos de erros. O pesquisador costuma notar apenas palavras que ele não esperava que fossem aparecer em um texto de data tão antiga como o que está sendo analisado. Mas inúmeras outras palavras podem escapar essa "intuição". Além disso, um pesquisador sozinho não conseguirá analisar todas as palavras do texto, o que torna necessário várias pessoas e muitas análises do mesmo texto para cobri-lo exaustivamente.

Dessa forma, um programa capaz de ler um texto e analisar automaticamente cada palavra poderia ajudar muito nas pesquisas, resolvendo problemas como estes. O ideal seria um programa que soubesse as ocorrências mais antigas registradas de todas as palavras do português, o que infelizmente não é possível. Assim, surgiu a ideia de um programa que analisasse um texto e avisasse os pesquisadores quando o texto contivesse ocorrências mais antigas do que as já registradas. A ideia de ter um programa analisando os textos surgiu em 2012 junto com o próprio NEHiLP, e esse programa foi batizado Moedor de Textos. Sua implementação, que mudou o nome para apenas Moedor, aconteceu nos anos seguintes e seguiu a ideia de manter um registro das ocorrências mais antigas das palavras analisadas pelo programa, para contornar a falta de uma fonte com as datas das ocorrências mais antigas de todas as palavras da língua.

Com isso, o projeto Retrodatação tem seus objetivos divididos em duas frentes: por um lado, o desenvolvimento do Moedor, capaz de receber textos, analisar as palavras, avisar se ocorrerem retrodatações e salvar as datas das ocorrências mais antigas de cada palavra; por outro, a inserção de textos pelos pesquisadores, de modo a construir uma base de dados o mais ampla possível contendo essas ocorrências, suas datas, obras, contextos etc. Na frente de desenvolvimento do programa, o projeto está finalizado, com o Moedor pronto e funcionando e o resto do sistema do NEHiLP complementando e facilitando o funcionamento; os programas estão estáveis e o foco está na correção de bugs e criação de funcionalidades novas mais simples. Já na frente da pesquisa, textos continuam sendo moídos e analisados, de modo a melhorar e aumentar a base de dados, o que permite o avanço do outro projeto principal, a criação do DELPo.

2.1.2 DELPo

Ao contrário de outras línguas europeias, como inglês, francês, espanhol e italiano, o português não conta com informações etimólogicas de boa qualidade em seus dicionários. Alguns dos problemas com os dados disponíveis são confusão sobre o que é de fato etimologia e o que são derivação por sufixo/prefixo ou origem remota da palavra, além de falhas no tratamento de termos influenciados por outras línguas ou com origem nelas. Assim, a pesquisa na área é constantemente prejudicada pela falta de recursos com nível similar aos disponíveis nessas outras línguas europeias.

Considerando isso, o Projeto DELPo do NEHiLP tem como objetivo a criação de um dicionário etimológico da língua portuguesa, que consiga se adequar aos requisitos da linguística moderna e preencher as lacunas deixadas por materiais existentes. Os dados coletados no Projeto Retrodatação são extremamente importantes para isso, e os programas desenvolvidos para o NEHiLP são essenciais para a busca e categorização deles e a obtenção de ainda mais informação.

Ao mesmo tempo em que depende dos resultados do Projeto Retrodatação, o Projeto DELPo é mais ambicioso e focado na pesquisa. Considerando essa relação entre os dois e o caráter do segundo, é possível dizer que a criação do DELPo é um dos objetivos finais do trabalho do NEHiLP, o que significa que o projeto está constantemente em processo de expansão e melhora. Dessa forma, o sistema do NEHiLP também fica, apesar da fase estável, em desenvolvimento contínuo, sendo alterado e tendo funcionalidades adicionadas de acordo com as demandas do Projeto DELPo.

2.1.3 Conceitos Relacionados

No trabalho de pesquisa dos projetos e no sistema feito para auxiliá-lo, alguns termos são recorrentes e, para facilitar a compreensão quando aparecerem em outras partes deste trabalho, eles são explicados de forma concisa a seguir. Enquanto alguns destes termos são conceitos da etimologia já existentes, outros foram propostos durante o desenvolvimento do sistema e representam estruturas no banco de dados ou, possivelmente, até novos conceitos etimológicos.

Variantes

Uma variante representa uma variante ortográfica de uma palavra, ou seja, a palavra exatamente como ocorre no texto, sem passar por lematização (ver Lemas) nem ter a ortografia atualizada.

Ocorrências

Sempre que uma variante aparece num texto, é registrada uma ocorrência, especificando-se o texto e o contexto no qual ela ocorreu. Assim, para cada variante podem existir n ocorrências, mas para cada ocorrência há uma única variante. Além disso, as variantes no banco de dados contêm a informação sobre qual é sua ocorrência mais antiga.

Contextos

O contexto é um pedaço de um texto. Cada ocorrência tem um único contexto, mas um contexto pode ser o mesmo para n ocorrências. O Moedor, ao receber um texto, separa ele em contextos e os salva no banco de dados.

Lemas

Um lema é uma forma básica de uma palavra, uma versão com tempo verbal, número, gênero pré-definidos, geralmente a mesma que aparece nos dicionários. Exemplo: o lema de "aconteceu" é "acontecer", e o de "gatas" é "gato". O processo de normalizar uma palavra para obter o lema é chamado de lematização. No banco de dados do NEHiLP não há uma tabela para lemas, mas sim para Metalemas.

Flexões

No contexto do banco de dados, uma flexão é uma versão não lematizada da palavra, mas com a ortografia atual. Assim, se, por exemplo, a variante é "offereceu", a flexão é "ofereceu". Cada variante tem uma única flexão, enquanto uma flexão pode ser a mesma para n variantes.

Acepções

Uma acepção é um registro sobre o significado de uma flexão, contendo a forma lematizada da flexão e informações como definição, classificação semântica e classificação morfológica. Cada flexão tem uma única acepção, mas cada acepção pode estar relacionada a n flexões. Se várias flexões com lemas de grafia idêntica têm significados distintos, elas geram várias acepções, todas com a mesma grafia mas cada uma com seu significado.

Metalemas

O metalema foi criado para resolver o problema de que o sistema não tinha como saber se dois lemas com a mesma grafia são um caso de homonímia (origens distintas) ou de polissemia (origem comum). Assim, o metalema não tem informações sobre significado e agrega acepções homônimas. Cada acepção tem um único metalema, e cada metalema pode estar associado a n acepções, cada uma com significado distinto.

Terminus a quo e *Terminus ad quem*

O *terminus a quo* é definido como a data da ocorrência mais antiga de uma acepção, flexão ou variante.

O *terminus ad quem*, por sua vez, é definido como a data da ocorrência mais recente de uma acepção, flexão ou variante.

Hiperlemas

Um hiperlema é a acepção mais antiga dentre um conjunto de acepções que têm o mesmo metalema. Ele é útil para separar homônimos com o mesmo metalema, já que, ao contrário dos metalemas, hiperlemas homônimos podem existir. Cada hiperlema tem uma acepção principal (a mais antiga) e cada acepção tem seu hiperlema, mas um hiperlema pode ser o mesmo para n acepções.

Ultralemas

Um ultralema é um componente morfológico (geralmente uma raiz de palavra, mas não sempre) relacionado à origem de diversas palavras. Cada ultralema pode remeter a diversas acepções, e cada acepção também pode ser relacionada a diversos ultralemas. Uma relação análoga existe também entre ultralemas e hiperlemas.

Hemilemas

Um hemilema é uma abreviatura ou uma palavra truncada. Para compreender o(s) significado(s) do hemilema, ele pode ser relacionado a uma ou mais acepções. Uma acepção também pode ser relacionada a vários hemilemas. O conceito (e a tabela no banco) de hemilema não está em uso no sistema hoje em dia, havendo apenas algumas entradas de teste inseridas no início do desenvolvimento.

Obras, Autores, Cidades, Editoras

O banco de dados do NEHiLP também conta com tabelas para armazenar informações sobre obras, autores, cidades e editoras. Um detalhe importante é que as datas associadas às entidades descritas são definidas por meio de uma hierarquia que tem sua base nas obras: uma ocorrência tem um contexto, vinculado a uma obra, que tem uma data; uma variante tem uma ocorrência mais antiga, cuja data vem da obra associada ao contexto da ocorrência; uma acepção tem uma flexão, que tem variantes, que têm suas datas, e assim por diante.

Contextos têm uma única obra cada, mas cada obra pode ter dado origem a n contextos. Além disso, uma obra pode ter n autores, de diferentes tipos (intelectual, material, editor etc.), e cada autor pode ter também m obras. O mesmo tipo de relação existe entre obras e editoras. Cada editora, por sua vez, tem uma cidade (e uma cidade pode estar associada a n editoras). Várias entradas no banco representam filiais de uma mesma editora em diferentes cidades.

2.2 Django

Django é um *framework* web em Python, grátis e com código aberto. Ele se propõe a cuidar sozinho da maioria das inconveniências associadas ao desenvolvimento web, deixando programadores livres para desenvolverem aplicações sem precisarem reinventar a roda. O Django tem como principais vantagens a rapidez e facilidade para se criar uma aplicação, a quantidade de módulos embutidos para cuidar de tarefas comuns no desenvolvimento web, segurança, boa escalabilidade e versatilidade, facilidade de customização e documentação extensa e de boa qualidade.

2.2.1 Termos e Conceitos

Ao longo do trabalho, vários termos e conceitos relacionados ao Django aparecem frequentemente; os mais importantes/recorrentes são brevemente definidos a seguir.

Projetos e *apps*

No Django, um site completo é chamado de projeto. Em termos técnicos, um projeto precisa ter um pacote Python (um diretório com arquivos com código Python) contendo as configurações para uma instância do Django, desde configurações do banco de dados até preferências específicas do Django ou de aplicações que fazem parte do site.

Essas aplicações que compõem um site são chamadas de *apps*. Assim, a estrutura de um projeto é um conjunto de configurações e *apps*. Um *app* é basicamente uma aplicação web que tem um propósito definido. Da mesma forma que um projeto pode ter vários *apps*, um *app* pode ser escrito de modo a ser reutilizável e, assim, figurar em diferentes projetos [6].

Modelos

No Django, modelos são a fonte de informações sobre a estrutura dos dados. Um modelo é uma classe em Python (subclasse de uma classe base do próprio Django) que representa uma tabela no banco de dados. Assim, cada campo da tabela corresponde a um atributo da classe. O Django fornece classes representando os tipos de dados de um banco de dados, como número inteiro, texto, data, chave estrangeira etc. Assim, num modelo, os atributos são instâncias dessas classes, definindo propriedades como tamanho máximo, possibilidade de ser nulo e outras [7].

Além disso, as classes de modelos podem ter uma classe interna `Meta` que contém alguns detalhes extras, como o nome da tabela no banco e o nome usado para a descrição daquele modelo (por exemplo, para o modelo `Acepcao`, o nome usado na descrição seria "Acepção").

Templates

Em Django, *templates* são a forma usada para gerar conteúdo dinâmico. Uma *template* é um arquivo de texto (na grande maioria dos casos, HTML) que junta partes estáticas com alguns comandos especiais que descrevem como o conteúdo dinâmico deve ser inserido. As páginas HTML de uma aplicação em Django são primeiro manipuladas por ele para aplicar as ações especificadas pelos comandos na *template* e depois renderizadas como HTML [8].

Dentro do diretório `templates` de cada *app*, existe um diretório com o nome do *app* e somente dentro deste último se encontram as *templates*. O motivo dessa estrutura é que, ao procurar uma *template* com um certo nome (`index.html`, por exemplo), o Django devolve a primeira *template* que ele acha. Se dois *apps* tivessem *templates* com este mesmo nome, diretamente no diretório `templates` de cada, o Django não teria como diferenciar as duas. Assim, o diretório com o nome do *app* é usado para criar um *namespace*, de modo que a *template* possa ser identificada como `nome-app/index.html`, acabando com a ambiguidade.

Arquivos estáticos

Arquivos estáticos são arquivos como CSS, JavaScript, imagens, PDFs e arquivos de outros tipos. Esses arquivos podem ser referenciados nas *templates* por meio de uma das *tags* que representa um comando especial. Eles ficam em diretórios chamados `static` que, assim como no caso dos diretórios das *templates*, devem conter dentro deles um outro diretório

com o nome do *app* e, dentro deste, os arquivos. É comum também que os arquivos sejam separados em subdiretórios como `css`, `js` e `img`, principalmente nos casos em que estão em grande quantidade [9].

Views

Uma *view* é uma função ou classe em Python que recebe uma requisição HTTP e devolve uma resposta. As *views* estão diretamente ligadas a URLs e, quando uma requisição é feita para uma URL, ela é repassada para a respectiva *view*, que faz o processamento necessário para gerar e retornar a resposta. As *views* costumam ficar em arquivos (em cada *app*) chamados `views.py` [10].

Muitas vezes, a lógica dentro de uma *view* inclui acessar e/ou modificar informações no banco de dados, o que é feito por uma API gerada automaticamente a partir dos modelos. Também é bastante comum que a resposta gerada pela *view* seja o resultado da renderização de uma *template*. Assim, a arquitetura do Django segue um padrão *Model-Template-View*, já que as *views* frequentemente servem como a ponte entre os modelos e as *templates*.

URLs

Assim como em qualquer sistema web, no Django existe um mapeamento entre URLs e programas/páginas correspondentes. O nome dado pelo *framework* para esse mapeamento é URLconf, e o objetivo do Django é que ele seja simples de ser feito pelo desenvolvedor e elegante para o usuário, evitando URLs difíceis de ler, confusas, com extensões de arquivos no final ou com outros problemas [11].

As URLs são definidas na URLconf por meio de expressões regulares (na sintaxe de Python), que determinam os padrões que serão associados a uma certa *view*. Elas permitem que argumentos sejam capturados e passados para as *views*, tanto de forma posicional quanto por nome. Os arquivos com URLconfs geralmente são chamados `urls.py`.

ORM

A ferramenta de ORM (*Obejct-Relational Mapping* ou Mapeamento Objeto-Relacional) do Django converte a estrutura relacional de um banco de dados para objetos em Python, e vice-versa; assim, ela faz a ligação entre as tabelas e entradas no banco de dados e os modelos definidos do Django. De acordo com o que está definido nos modelos, o Django disponibiliza automaticamente uma API que permite a criação, o acesso e a modificação dos dados no banco. Assim, ao invés de ter que escrever código SQL, o desenvolvedor pode escrever diretamente código em Python, o que, na maioria dos casos, é menos trabalhoso [12].

Script de administração

O *script* de administração `manage.py`, que vem incluso em qualquer projeto Django, permite ao programador executar diversas ações, como iniciar o servidor, iniciar um *shell* Python com acesso aos modelos e funções do Django, organizar os arquivos estáticos, aplicar mudanças no banco de dados etc. [13]

Interface de administração

O Django fornece uma interface bastante completa para os administradores do sistema, acessível (por padrão) a partir de `http://endereco-do-site.com/admin`. Dentro

dessa interface, é possível inspecionar, editar e criar entradas de cada modelo, e também usuários. Também é possível fazer buscas nos conjuntos de objetos. Para configurar um modelo para aparecer no admin, bem como certas propriedades (ordenação padrão, campos que são usados na busca e várias outras), é usado o arquivo `admin.py` em cada *app*. Cada modelo precisa de uma classe relacionada nesse arquivo, contendo as propriedades e configurações relevantes, e que é registrada no admin [14].

Migrações

Toda vez que uma mudança é feita em algum dos modelos, é preciso propagar esta mudança para o banco de dados. No Django, isso é feito com as chamadas migrações, que tentam deixar o processo o mais simples possível para o desenvolvedor. A parte principal desse processo é feita com dois comandos no *script* `manage.py` [15].

O primeiro, `makemigrations`, inspeciona os modelos e, se houver alguma mudança, cria um arquivo de migração contendo os detalhes das mudanças e as ações necessárias para aplicá-las no banco. Já o segundo, `migrate`, aplica essas migrações descritas nos arquivos, comunicando-se diretamente com o banco de dados e fazendo os ajustes necessários. Assim, as migrações servem como uma espécie de sistema de controle de versão do esquema do banco de dados.

`virtualenv`/`pip`

O `virtualenv` (que é um conceito mais de Python do que de Django) [16] é um programa que cria ambientes isolados para executar código em Python, permitindo que se instalem bibliotecas somente para um ambiente, sem ser necessário instalá-las globalmente. Após criar e ativar o ambiente, basta usar o `pip` [17] (o gerenciador de pacotes do Python, que fica disponível nos ambientes do `virtualenv`) para instalar as bibliotecas listadas. Geralmente, projetos em Python contêm um arquivo `requirements.txt` listando as bibliotecas utilizadas, que é lido pelo `pip` no momento de instalá-las.

Capítulo 3

O Sistema Atual

Para poder fazer uma reescrita do sistema atual[1], assim como identificar problemas e pontos a melhorar nele e procurar soluções, foi imprescindível entender sua história, estrutura e funcionamento. O sistema está ativo no momento, sendo usado pelos pesquisadores envolvidos nos projetos Retrodatação e DELPo. Ele teve sua base feita entre 2013 e 2015, e continuou a ser melhorado e ter funcionalidades novas criadas depois disso. Foi desenvolvido pelo Prof. Dr. Marco Dimas Gubitoso em conjunto com alguns alunos de graduação da USP, e o código está majoritariamente em PHP, mas com algumas partes importantes, como o Moedor, em Perl.

Para assimilar o funcionamento e a estrutura, algumas das atividades feitas foram o estudo tanto do próprio código-fonte quanto do manual (que descreve as funcionalidades e explica aos pesquisadores como usar o sistema). No entanto, a atividade mais importante foi a atuação na manutenção do sistema.

No início do ano, isso se deu por meio apenas da participação nas reuniões regulares com o Prof. Dr. Marco Dimas Gubitoso, o Prof. Dr. Mário Eduardo Viaro (coordenador do NEHiLP) e Antonio Augusto Abello, aluno de graduação co-responsável pela manutenção até a metade de 2017. Após algumas reuniões, ocorreram sessões de programação pareada com o Antonio para resolver *bugs* e adicionar novas funcionalidades. Por fim, com a saída dele do projeto no meio do ano, a responsabilidade de auxiliar o Prof. Dr. Marco Dimas Gubitoso na manutenção foi assumida integralmente. Esta atividade, embora fora do escopo deste trabalho de conclusão de curso, foi bastante útil para seu desenvolvimento, por permitir contato prático com o sistema, o que ajudou a entender melhor seu funcionamento e questões relacionadas a ele.

A seguir, são explicadas as funcionalidades e a organização do programa e a estrutura do banco de dados e, por fim, são expostos alguns problemas e oportunidades de melhora identificados (e posteriormente abordados na criação da versão nova do sistema).

3.1 Funcionalidades e Estrutura

Em termos de funcionalidades e páginas do site, o sistema do NEHiLP pode ser dividido em quatro categorias: administração (páginas de perfil, configurações), conteúdo (páginas com texto/conteúdo estático), fórum e, por último, as páginas e funções "principais" (busca,

[1] O sistema já existente é chamado neste texto de sistema atual, principalmente pelo fato de que ele era o sistema ativo durante o período em que o trabalho foi feito e continuaria o sendo por um certo tempo, mesmo após a conclusão deste trabalho. O sistema desenvolvido no trabalho é chamado de sistema novo ou sistema em Django.

inserção, edição, visualização). As páginas, programas e funções de cada uma dessas categorias são descritas abaixo e, depois, a estrutura de pastas e arquivos que compõem o sistema é explicada.

3.1.1 Administração

Alguns dos componentes mais importantes de administração são as funcionalidades de autenticação (*login*, *logout*, cadastro, recuperação de senha). Além delas, há uma página de perfil do usuário, onde ele pode visualizar e modificar seus dados, e uma página com a lista de colaboradores (pesquisadores). Para os usuários que são administradores, estão disponíveis também uma página para listar a produtividade (em termos de datações inseridas num dado período) de pesquisadores bolsistas, e uma página para modificar o nível e as permissões de cada usuário, bem como outras configurações gerais.

3.1.2 Conteúdo

As páginas de conteúdo são textos, listas ou *links* relacionados aos projetos. Existem as páginas de descrição do NEHiLP, dos projetos de Retrodatação e do DELPo; as páginas contendo informações sobre a série bibliográfica Arquivos do NEHiLP (apresentação, corpo editorial, normas, volumes, informações de contato); as páginas de bibliografia, créditos, parcerias, pesquisadores e endereço do núcleo; as páginas de agenda e eventos; as páginas que reúnem dissertações e teses, textos, livros, reuniões, *links* e cursos online.

3.1.3 Fórum

O fórum é um espaço para os usuários conversarem e tirarem dúvidas relacionadas ao sistema e ao trabalho de pesquisa. Ele possui uma página de Perguntas Frequentes, uma página que lista os sub-fóruns (no momento, apenas etimologia e computação) e, dentro de cada sub-fórum, encontram-se os tópicos criados pelos usuários, que podem ser lidos e respondidos por outros usuários. Além disso, os administradores podem editar as páginas do fórum, apagando comentários e adicionando mais itens nas Perguntas Frequentes. O fórum é, atualmente, bem pouco usado e pode ser descontinuado no futuro próximo.

3.1.4 Principal

As funcionalidades, programas e páginas "principais" do sistema, ou seja, os que estão diretamente relacionados à coleta e análise de dados no contexto dos projetos Retrodatação e DELPo, podem ser subdivididos em três partes: a que trata da inserção de dados (o que inclui o Moedor de textos), a que trata da busca de informações no banco de dados e da modificação de informação já inserida, e uma última, menos definida, que inclui outros programas com funções de vários tipos.

Inserção e Moedor

A inserção de dados no sistema ocorre, no nível mais baixo, por meio da inserção de entradas no banco de dados. Isto pode ser feito pelos usuários de duas maneiras principais: pelo Moedor e por meio da inserção manual.

O Moedor, primeira parte idealizada e implementada do sistema original, é um programa que recebe um arquivo de texto, o processa e retorna uma página que apresenta alguns dos diferentes contextos que compõem o texto e as ocorrências relacionadas que ele identifica

como “interessantes”, seja por representarem uma retrodatação de uma palavra, por serem palavras novas etc.

As interações do usuário com o Moedor se dão por meio de algumas páginas. A de sugestão de textos é a página na qual pesquisadores podem inserir um arquivo para ser moído e as informações sobre o texto contido nele. Essa sugestão não vai imediatamente para o Moedor, mas sim para os coordenadores, que fazem a revisão do texto e usam a página de inserção de texto, que é similar à de sugestão, mas o insere diretamente no Moedor. Já a página de sessões abertas permite a um usuário ver todos os textos que ele inseriu no Moedor e as sessões correspondentes (e, para os administradores, textos de todos os usuários). Por fim, as páginas de cada sessão permitem que os usuários analisem os resultados da moagem e façam inserções de novas variantes, flexões, acepções e metalemas.

A inserção manual, por sua vez, divide-se entre a inserção de ultralemas e o Papavero. A primeira é uma página simples que, como indica o nome, permite que um usuário de nível mais alto faça uma inserção manual de um ultralema ou um ultralema remoto, incluindo comentários etimológicos sobre ele.

O Papavero, por outro lado, é um sistema mais completo para a inserção manual de informações no banco de dados. Ele permite que sejam adicionados manualmente informações de metalema, acepção, variante, flexão, ocorrência, contexto e até novas obras. Para usá-lo, também é necessário ter um nível alto no sistema, o que protege o banco contra erros. A inserção se dá em uma única página.

Buscas e Edição

O sistema permite ao usuário realizar diversos tipos de buscas. Em várias delas, é possível (de acordo com o nível do usuário) editar dados sobre os resultados das buscas, permitindo a correção de erros. Cada busca tem sua própria página.

A busca morfológica permite ao usuário escolher uma classificação morfológica dentre as opções disponíveis e ter acesso a todas as flexões com essa classificação, além de informações sobre elas. Essa busca não permite edição dos resultados.

A busca semântica procura acepções com uma dada classificação semântica (também escolhida dentre as opções existentes no sistema) e exibe-as junto com algumas informações relacionadas. Essa busca também não permite que seus resultados sejam editados.

A busca de abreviaturas permite buscar os hemilemas: ela procura no banco o que for digitado no campo de pesquisa e retorna informações sobre os hemilemas encontrados, como interpretações e contextos. Como a estrutura de hemilemas não está sendo usada por enquanto, os resultados desta busca tendem a ser poucos, limitando-se a dados inseridos no início da operação do sistema, quando as funções relacionadas a abreviaturas foram testadas. Os resultados dessa busca não são editáveis.

A busca de lemas permite que o usuário busque os hiperlemas e mostra informações sobre eles, como sua etimologia. É possível também ver os verbetes do DELPo relacionados aos hiperlemas buscados. Não é possível editar os resultados dessa busca.

A busca de origem permite a pesquisa de ultralemas. Por padrão, ela já mostra todos disponíveis, mas pode-se filtrar os resultados usando o campo de pesquisa. Informações como cognatos e comentários etimológicos aparecem para cada resultado, e é possível também (dependendo do nível do usuário) apagar e editar os ultralemas.

A busca de obras mostra por padrão todas as obras disponíveis, junto com informações como método de inserção, autor, e, quando disponíveis, outras como localização, publicação, concedente etc. É possível filtrar os resultados digitando-se um nome (ou pedaço de nome) de obra no campo de pesquisa. Além disso, ao clicar-se em um resultado, usuários com a

devida permissão podem editar alguns dos detalhes relacionados à obra.

A busca geral tem como objetivo permitir a pesquisa de informações etimológicas úteis para os pesquisadores, como as definições de acepções, as etimologias, etimologias de flexões, comentários etimológicos e outros. A busca permite o uso de expressões regulares para ampliar o escopo do que pode ser pesquisado. Os resultados dessa busca não podem ser editados.

A busca dicionário é uma das mais importantes, já que é a mais utilizada e que agrega maior quantidade de informações. Ela permite que se pesquise palavras e mostra os verbetes das acepções correspondentes, junto com dados de contexto, flexões, variantes, *terminus a quo* e outras. Para administradores, é possível também pesquisar todas as acepções que tiveram participação de um determinado usuário. Os resultados podem ser editados (tanto as informações sobre as acepções quanto os dados das flexões e variantes relacionadas), com os campos editáveis variando de acordo com o nível do usuário.

A busca por datas compartilha muitas características com a busca dicionário. Ela exibe fichas bastante completas, com dados sobre acepções, flexões, variantes, abonação, atualizações e outras. Além disso, estas informações também podem ser editadas. A grande diferença dessa busca para a dicionário é que, ao invés de se buscar uma palavra ou um usuário, são inseridas datas, e os resultados mostram acepções que tenham flexões/variantes com data de ocorrência entre as datas pesquisadas, e não todas as flexões e variantes.

Outros Programas

Além destes programas descritos (de inserção e busca/edição), existem mais alguns programas no sistema do NEHiLP, com propósitos variados. São eles o N-Gram, o Metaplasmador e o Concordanciador.

O N-Gram, inspirado pelo Google Ngram Viewer [18], é uma ferramenta que mistura busca e visualização. Ele permite que se pesquise palavras e apresenta um gráfico mostrando a quantidade de ocorrências de cada uma com o passar do tempo, de acordo com os registros no banco de dados. É possível também digitar mais de uma palavra e comparar as frequências. Este tipo de informação tem o potencial de servir como base para análises bastante interessantes, mas, por enquanto, tem utilidade reduzida devido à quantidade pequena (para esse propósito) de dados no banco do NEHiLP. É provável que no futuro a importância deste programa cresça. Uma funcionalidade planejada (e cuja implementação deve ser facilitada por este trabalho) é a pesquisa de um metalema, por exemplo, e a geração do gráfico para todas as variantes correspondentes, algo que seria possível graças à estrutura do banco de dados do NEHiLP e que não está disponível na versão do Google.

O Metaplasmador é um programa que não utiliza o banco de dados do NEHiLP, e que foi pensado para auxiliar os pesquisadores no momento de preencher informações nos campos de etimologia de acepções e flexões. Ele recebe uma palavra em latim como entrada e aplica uma série de transformações pré-definidas por regras para tentar reproduzir o processo que levou a palavra original em latim até a versão atual em português. Evidentemente, o processo não resulta em palavras válidas em português para qualquer entrada em latim (mais detalhes sobre o funcionamento podem ser vistos no manual), mas fornece dados úteis aos pesquisadores nos casos em que é possível chegar ao resultado correto.

Por fim, o Concordanciador reúne várias funcionalidades em um único programa, permitindo uma combinação de busca, edição e visualização dos dados do banco. A ideia por trás dele é dar acesso a contextos ignorados pelos outros programas, já que o Moedor não mostra os contextos que não contêm palavras novas nem possíveis retrodatações, e a maioria das buscas só mostra o contexto da ocorrência mais antiga. O Concordanciador recebe uma

palavra como parâmetro de busca e retorna todos os contextos em que ela ocorre, junto com data e obra. Cada ocorrência pode ser selecionada e, com isso, é possível inserir novas informações relacionadas, usando os mesmos formulários de inserção/edição de acepções e flexões que aparecem nas sessões do Moedor. Além disso, cada busca no Concordanciador gera um gráfico como o do N-Gram, para facilitar a visualização de informações sobre as ocorrências.

3.1.5 Organização dos Arquivos

Todas estas funcionalidades descritas estão contidas em diversos programas em PHP, Perl e JavaScript. Os arquivos com estes programas estão dispostos em vários diretórios no servidor. Por ser uma das partes que mudou consideravelmente na nova versão, a estrutura de diretórios e programas do sistema atual é descrita a seguir, junto com comentários sobre o planejamento da reescrita dos programas de cada diretório.

Além dos diretórios contendo os programas do sistema, existem alguns contendo bibliotecas externas utilizadas, que são discutidos no final desta seção. Os diretórios principais, nos quais programas e arquivos estão contidos, são os seguintes:

- **arquivos**
- **cgi-bin**
- **conexao**
- **consulta**
- **conteudo**
- **css**
- **edit**
- **forum**
- **img**
- **imgs-user**
- **includes**
- **js**
- **programas**

O diretório `arquivos` guarda diferentes arquivos relacionados ao NEHiLP, como atas de reuniões, apresentações de slides sobre etimologia, teses relacionadas aos campos de estudo do núcleo etc. Estes arquivos são servidos junto com o site e são acessíveis através de algumas das páginas de conteúdo. Para poderem ser acessados na versão nova, eles foram copiados.

O diretório `cgi-bin` contém os arquivos/programas em Perl, em especial os que implementam o Moedor. A adaptação destes programas não estava no escopo deste trabalho, mas está planejada para o futuro. A integração da versão atual do Moedor (em Perl) com a versão nova do sistema teoricamente é possível; não houve tempo de fazê-la neste trabalho, mas ela foi um pouco estudada (a integração atual com o código em PHP forneceu informações boas).

O diretório `conexao` contém programas/funções em PHP que não são acessíveis pelo site, mas são usados indiretamente, como auxiliares, pelos programas "principais" com os quais os usuários interagem. De modo geral, cada programa/função nesses arquivos realiza um serviço e depois devolve o controle ao programa que o chamou. As funcionalidades

presentes neles incluem configuração da conexão com o banco, criação e alteração de usuários, busca de informações, alteração/inserção de dados e várias outras.

Se por um lado este diretório consegue reunir a grande maioria dos programas e funções que apenas "realizam um serviço", por outro a quantidade de arquivos nele (por volta de 50) é bem grande, o que gera dúvidas quanto à eficácia desta organização. Assim, a adaptação destes programas para a nova versão ocorreu de acordo com a necessidade (quando um programa que chamava algum destes foi adaptado, o auxiliar também foi) e considerando a possibilidade de modificar a organização.

O diretório `consulta` contém arquivos PHP responsáveis pelos programas de buscas descritos. A adaptação destes programas é, em teoria, simples, já que eles são compostos em sua maior parte por consultas SQL, com pouca lógica de programação. No entanto, para fazer a adaptação de cada, é preciso entender bem as consultas, avaliar se elas poderão ser convertidas para o modelo do Django ou se terão que continuar sendo feitas em SQL (o que é permitido também) e só então reescrevê-las (e também as partes das páginas dedicadas aos resultados). O objetivo inicial foi definido como a adaptação de apenas uma das buscas (não especificada), mais como uma espécie de prova de conceito, para deixar claro o caminho a ser seguido com as demais.

O diretório `conteudo` contém programas em PHP que pegam no banco conteúdo (texto) para ser mostrado nas páginas respectivas. Estas mesmas páginas permitem aos usuários com permissão de edição que façam a modificação deste conteúdo. Como adaptar estes programas se resume a criar páginas no Django que igualmente busquem o conteúdo no banco, o objetivo estabelecido foi trazer todas as páginas de conteúdo para a nova versão.

O diretório `css` contém os arquivos CSS (e alguns outros relacionados, como os do Bootstrap e algumas imagens) usados nas páginas do sistema. Já que continuarão sendo usados na versão nova, estes arquivos foram copiados (com pequenas alterações e adições, quando necessário).

O diretório `edit` contém arquivos em PHP que implementam a edição de vários tipos de dados. Podem ser editados acepção/flexão/variante, hemilema, ultralema (para o qual o mesmo programa também serve para inserir), o conteúdo das páginas, o *banner* com os próximos eventos (que aparece na página principal) e as abonações das acepções. Como já foi descrito, a maioria destas funcionalidades de edição está vinculada diretamente a algumas das buscas; com isso, o objetivo para a reescrita dos programas de edição foi a adaptação de pelo menos um deles (relacionado à(s) busca(s) que for(em) adaptada(s)), e também da edição de conteúdo, que não tem associação com nenhuma busca e cuja lógica é bem mais simples.

O diretório `forum` contém os programas em PHP responsáveis pelo funcionamento do fórum. Eles tratam de tópicos, comentários etc. Como o fórum praticamente não está sendo utilizado, ele tem prioridade bastante baixa na reescrita do sistema e, assim, sua adaptação não entrou nos objetivos deste trabalho.

O diretório `img` contém imagens usadas no sistema, como ícones de botões, logotipos, imagens que aparecem em páginas etc. As imagens foram copiadas para o sistema novo.

O diretório `imgs-user` contém imagens inseridas por usuários no sistema. São imagens descrevendo acepções, uma função que é possível no sistema, mas não é usada no momento. A pasta não foi recriada para este trabalho, até porque, para funções como essa, seria interessante explorar o conceito de *media* do Django, usado para guardar/servir arquivos estáticos enviados pelo usuário.

O diretório `includes` contém arquivos em PHP que incluem elementos de PHP/HTML/CSS/JavaScript que são usados em várias páginas. Alguns exemplos são o menu, a lista de classificações morfológicas, o cabeçalho/rodapé das páginas, HTML para incluir arquivos

CSS e JavaScript que são reutilizados, *pop-up*s (de *login*, cadastro etc.), configurações, e funções (PHP) utilitárias. As configurações foram copiadas (e adaptadas de acordo com as diferenças entre o PHP e o Django), assim como os elementos das páginas. As funções utilitárias foram adaptadas conforme a necessidade na adaptação de outros programas.

O diretório `js` tem bibliotecas/plugins especificamente de JavaScript, como jQuery/ jQuery UI, `formoid.js`, `jqpagination.js` e `zclip.js`. Além disso, também estão neste diretório arquivos com código JavaScript usados no sistema: `database.js` contém variáveis muito grandes, que são listas de classificação semântica, universo discursivo e gênero, usadas como dicionário para auto completar; `controller.js` tem diversas funções "de controle", como verificação dos dados em formulários, mudança dinâmica de design e várias outras; `ajax.js` tem diversas funções que fazem chamadas AJAX para enviar informação para o banco ou receber dele, tratando a informação e mostrando-a na interface gráfica sem ser necessário recarregar ou mudar de página.

Uma lógica bastante usada no sistema é: um elemento (em HTML) de alguma página tem um `onClick` *handler* ligado a uma função no arquivo `ajax.js`; essa função faz uma chamada AJAX a um programa relacionado do diretório `conexao`, que cuida da integração com o banco, e devolve informações necessárias para atualizar a interface. Assim, da mesma forma que os programas em `conexao`, as funções contidas em `ajax.js` (e também em `controller.js`) foram adaptadas para a versão nova conforme foi necessário, de acordo com os programas que estavam sendo adaptados. Como estas funções são em JavaScript, a princípio podiam ser copiadas, mas foi necessário fazer algumas alterações devido a modificações nos programas que as chamam e os que são chamados por elas.

As bibliotecas JavaScript no diretório `js` foram copiadas, com a exceção de algumas (como `zclip.js`) que pareciam não estar mais sendo utilizadas. Apesar de não terem sido incluídas a princípio, se for constatada a necessidade podem ser copiadas no futuro. Além disso, esta estrutura de todos os arquivos JavaScript juntos, com dois grandes arquivos principais com muitas funções dentro (`ajax.js` e `controller.js`), foi mudada na versão em Django.

O diretório `programas` contém arquivos em PHP que implementam os programas e páginas "principais" descritos anteriormente, e também alguns dos de administração; alguns destes são: Concordanciador, inserção manual (Papavero), lista de sessões no Moedor, inserção/sugestão de obra no Moedor, N-Gram, página do perfil, redefinição de senhas, lista de usuários e produtividade dos usuários. A reescrita de todos os programas de administração estava dentre os objetivos deste trabalho, considerando que eles são mais simples do que os programas principais. Já estes últimos não foram incluídos no escopo inicial do trabalho.

A página inicial está no mesmo nível que todos estes diretórios descritos. Ela contém um pouco de texto, alguns *links* e referências a programas e o *banner* com informações sobre eventos. A adaptação dela fazia parte dos objetivos estabelecidos para a nova versão, possivelmente com algumas mudanças. Outro detalhe é que o Metaplasmador tem o programa correspondente em PHP dentro do diretório `programas`, mas tem vários arquivos JavaScript e CSS próprios em subdiretórios de `js` e `css`; ele também estava dentre os objetivos de adaptação para a versão nova, incluindo uma reorganização destes arquivos para uma configuração melhor.

Por fim, existem também alguns diretórios que são usados para armazenar bibliotecas externas: `ckeditor`, `pChart`, `phpmailer`, `recaptcha`, `tinymce` e `tokeniput`. Duas das bibliotecas (CKEditor e Tokeninput) são usadas apenas no e fórum foram ignoradas, já que o fórum não será adaptado por enquanto e, além disso, a primeira aparentemente pode ser substituído facilmente pelo TinyMCE e a segunda talvez nem esteja de fato sendo usada. Já o pChart também foi ignorado por não estar sendo usado. O PHPMailer, que é

responsável pelo envio de e-mails no sistema, foi substituído pelo serviço de envio de e-mails do próprio Django. O TinyMCE, que implementa um editor de texto rico, foi copiado e incluído normalmente na versão nova, já que é uma biblioteca de JavaScript. Por último, o reCAPTCHA não foi usado neste trabalho, mas poderá ser implementado usando diretamente a API REST do Google.

3.2 Banco de Dados

O banco de dados é uma parte essencial do sistema do NEHiLP, permitindo o armazenamento das informações inseridas pelos pesquisadores, além de consultas e modificações a essas informações. Assim, também foi essencial entender sua estrutura para poder criar os modelos na versão em Django, e aproveitou-se esta reescrita do sistema para melhorar alguns aspectos da organização do banco. Trinta e uma tabelas estão no banco da versão original; abaixo, são brevemente descritas, seguidas de informações sobre as relações entre elas e outros comentários.

3.2.1 Tabelas

As tabelas **ultralema**, **acepcao**, **hiperlema**, **metalema**, **flexao**, **variante**, **ocorrencia**, **contexto** e **hemilema** guardam entradas das entidades respectivas e algumas informações relacionadas.

As tabelas **historicoacepcao** e **historicoflexao** armazenam, respectivamente, detalhes adicionais sobre as acepções e flexões. Além disso, elas estão estruturadas como listas ligadas, permitindo que novas versões destas informações não tenham que sobrescrever (e referenciem) as anteriores.

As tabelas **acep_ultra**, **hiper_ultra** e **hemilema_acepcao** guardam, respectivamente, as informações sobre todas as relações entre acepções e ultralemas, hiperlemas e ultralemas e hemilemas e acepções. Já a tabela **acepcao_acepcao** guarda as informações sobre as relações de subordinação entre acepções.

A tabela **obra** contém as informações sobre as obras; da mesma forma, a tabela **autor** guarda os dados sobre autores, a tabela **editora** sobre editoras e a tabela **cidade** sobre cidades. Além disso, as tabelas **obra_autor** e **obra_editora** guardam informações sobre todos os autores e todas as editoras de cada obra, respectivamente.

A tabela **usuario** guarda todos os dados dos usuários. A tabela **email** guarda alguns endereços de e-mail associados a certos eventos (por exemplo, nova retrodatação, novo usuário cadastrado) para que um e-mail seja enviado ao endereço correspondente quando um destes eventos ocorre. A tabela **banner** guarda informações sobre os próximos eventos do NEHiLP para mostrar na página inicial. Já a tabela **conteudo** guarda o texto que é apresentado nas páginas de conteúdo estático.

A tabela **colaboracao** armazena dados sobre a quantidade de colaborações feita por um usuário (número de acepções, hemilemas e ultralemas inseridos). A tabela **datacao**, por sua vez, registra cada contribuição (retrodatação/flexão nova/acepção nova) dos usuários, indicando a ocorrência relacionada e a data e horário da contribuição.

As tabelas **resultado**, **sessao** e **ocorrenciatemp** são de uso interno do Moedor. A tabela **resultado** guarda os resultados da análise feita pelo Moedor, com as sugestões de metalema derivadas das palavras das obras. A tabela **ocorrenciatemp** relaciona as entradas de **resultado** com as ocorrências das quais elas tratam. Já a tabela **sessao** guarda as sessões abertas no Moedor, indicando qual é a obra, o pesquisador que a está analisando e os nomes dos arquivos com o código HTML dos relatórios gerados.

Por fim, a tabela **msg** é usada para armazenar as mensagens do fórum e dados relacionados, como horário de criação, autor (usuário), mensagens relacionadas e outros.

3.2.2 Relações e Comentários

As relações presentes no banco de dados são de vários tipos. Algumas são $1 \to n$ e outras $n \leftrightarrow m$. Se chamarmos de A e B as duas tabelas envolvidas numa relação (da esquerda para a direita), $1 \to n$ indica que cada entrada de A pode ter n entradas de B relacionadas, enquanto cada entrada de B só terá sempre uma entrada de A relacionada (no banco, B contém a chave estrangeira para A). Já $n \leftrightarrow m$ indica que cada entrada de A pode ter n entradas de B relacionadas, e cada entrada de B pode, também, ter m entradas de A relacionadas (no banco, uma tabela extra representa a relação e cada entrada dela tem chaves estrangeiras para as entradas de A e B).

As relações **hiperlema** → **acepcao**, **metalema** → **acepcao**, **acepcao** → **flexao**, **flexao** → **variante**, **variante** → **ocorrencia**, **ocorrencia** → **variante**, **contexto** → **ocorrencia**, **variante** → **ocorrencia**, **hemilema** → **ocorrencia**, **obra** → **contexto**, **historicoacepcao** → **acepcao**, **historicoflexao** → **flexao**, **cidade** → **editora**, **autor** → **obra**, **editora** → **obra**, **cidade** → **autor**, **autor** → **autor**, **usuario** → **colaboracao**, **usuario** → **conteudo**, **usuario** → **datacao**, **ocorrencia** → **datacao**, **datacao** → **datacao**, **historicoflexao** → **historicoflexao**, **historicoacepcao** → **historicoacepcao**, **ultralema** → **ultralema**, **usuario** → **msg**, **obra** → **sessao** e **usuario** → **sessao** são todas $1 \to n$. Alguns detalhes importantes sobre certas relações nesta lista são os seguintes:

- Enquanto a relação **variante** → **ocorrencia** representa a variante de cada ocorrência, a relação **ocorrencia** → **variante** representa a ocorrência mais antiga de cada variante.

- A relacão **autor** → **obra** está presente quatro vezes em **obra**. São quatro chaves estrangeiras, representando, respectivamente, um autor intelectual, um autor material, um organizador e um editor. Estes autores com as chaves em **obra** são geralmente (mas não necessariamente) os principais do tipo, já que cada obra pode ter vários autores de cada tipo (a relação completa se encontra em **obra_autor**). Um detalhe neste caso é que, apesar de o tipo "impressor" de autor estar definido no manual (e existir em algumas entradas de **obra_autor**), não há chave estrangeira representando o impressor principal. Por fim, um outro detalhe é que estas quatro chaves estrangeiras não estão de fato marcadas no banco como chaves, o que pode piorar a performance e fazer com que problemas de integridade não sejam percebidos.

- De forma similar, a relação **editora** → **obra** e a chave correspondente em **obra** representam uma editora (geralmente, mas nem sempre, a principal) da obra, com a lista completa sendo acessível por meio de **obra_editora**. Também como no caso dos autores, a chave estrangeira em **obra** não está marcada como uma chave.

- A relação **autor** → **autor** representa o pseudônimo de um autor, que pode ser outro autor já registrado. No entanto, esta relação nunca foi usada; um outro campo em **autor**, que permite escrever o pseudônimo (sem estabelecer uma relação com outro autor) é usado.

- Na relação **usuario** → **colaboracao**, a chave estrangeira em **colaboracao**, que representa o usuário do qual aquela entrada trata, não está marcada como chave. O mesmo

acontece com a relação **usuario** → **conteudo** e a chave em **conteudo**, que representa o último usuário que atualizou aquele conteúdo.

- A relação **usuario** → **datacao** está presente duas vezes em **datacao**, uma representando o usuário responsável pela contribuição, outra representando um moderador associado. Este segundo, no entanto, não está sendo usado. Além disso, a relação **datacao** → **datacao** representa a possibilidade de uma datação ser uma versão mais recente ou mais antiga de outra, por meio de uma estrutura de lista ligada; o mesmo ocorre com as relações **historicoflexao** → **historicoflexao** e **historicoacepcao** → **historicoacepcao**.

- A relação **ultralema** → **ultralema** representa a possibilidade de um ultralema ser a origem remota de outro.

- Na relação **usuario** → **msg**, que representa o usuário que é o autor de uma mensagem no fórum, a chave estrangeira em **msg** não está marcada como uma chave. O mesmo ocorre com as chaves em **sessao** nos casos das relações **obra** → **sessao** e **usuario** → **sessao**, que representam, respectivamente, o usuário analisando uma sessão do Moedor e a obra associada à sessão.

As relações $n \leftrightarrow m$ são **acepcao**, ↔ **acepcao** (por meio de **acepcao_acepcao**), **acepcao** ↔ **ultralema** (por meio de **acep_ultra**), **hemilema** ↔ **acepcao** (por meio de **hemilema_acepcao**), **hiperlema** ↔ **ultralema** (por meio de **hiper_ultra**), **obra** ↔ **autor** (por meio de **obra_autor**) e **obra** ↔ **editora** (por meio de **obra_editora**). O único detalhe menos evidente sobre elas, importante de se lembar, é que **acepcao**, ↔ **acepcao** representa a subordinação entre duas acepções.

3.3 Pontos a Melhorar e Corrigir

Neste trabalho de reescrita, a correção de problemas e melhora de certos pontos do sistema foram passos essenciais, que devem garantir um desenvolvimento mais fluido para o futuro do projeto e facilitar a implementação de novas funcionalidades. Dada esta visão mais completa do funcionamento e da estrutura do sistema atual, os principais pontos identificados para serem corrigidos ou melhorados são descritos a seguir.

3.3.1 Banco

Conforme visto na seção anterior, algumas tabelas do banco apresentam um mesmo problema: elas têm campos que, na teoria (na definição do que é cada entidade e das relações entre elas) e na prática (no código usado nos programas) são chaves estrangeiras, mas na definição do esquema no banco não estão marcadas como chaves.

Os problemas relacionados a isso são dois. Em primeiro lugar, performance: se a chave não está definida, o MySQL não cria automaticamente um índice, e um índice traria um ganho em velocidade para casos de uma relação que aparece com bastante frequência nas consultas. Além disso, sem a definição da chave não ocorre a checagem de integridade automática quando as tabelas relacionadas são modificadas; sem ela, é possível que algumas entradas no banco fiquem com dados inválidos, o que pode levar a erros cuja causa é difícil de achar.

De fato, neste processo de estudo da estrutura e das relações no banco e na manutenção do sistema atual, foram encontradas várias inconsistências; apesar de a origem delas não ser de todo clara, é seguro dizer que pelo menos algumas poderiam ter sido evitadas se essas

relações estivessem efetivamente marcadas como chaves. Alguns exemplos de inconsistências encontradas foram: obras cujo campo do autor intelectual apontava para um autor inexistente (mas que continham um autor correto na tabela **obra_autor**); entradas da tabela **datacao** apontando para ocorrências inexistentes; datas 0000-00-00 em campos de data (o que é válido em algumas versões do MySQL, mas não é uma boa prática).

3.3.2 Estrutura/Organização

A estrutura (de diretórios e arquivos) do NEHiLP passou recentemente por uma refatoração, que deu origem ao esquema descrito anteriormente. Esta refatoração contribuiu bastante para facilitar a compreensão da estrutura e agilizar o desenvolvimento. Ainda assim, é possível apontar vários pontos a serem melhorados; como a mudança para Python/Django já implicaria uma mudança de estrutura, aproveitou-se para tentar corrigir estes problemas, descritos a seguir.

Um primeiro ponto, que deixa mais difícil a manutenção e a compreensão de partes do sistema, é a falta de documentação em certos programas. Em uma parte considerável dos arquivos, há comentários explicando a maioria do código. No entanto, alguns programas/arquivos têm uma quantidade muito pequena de comentários e explicações, fornecem informações confusas/desatualizadas ou possuem nomes de variáveis pouco claros. Em funções e programas maiores, este problema fica mais sério, pois o tempo para buscar algo no programa e compreender o que cada pedaço do código faz torna-se maior. Assim, uma documentação mais extensa melhoraria a produtividade dos processos de manutenção e desenvolvimento do sistema.

Outro ponto são os arquivos grandes demais, como `ajax.js` e `controller.js`. Juntos, eles dois somam mais de 2000 linhas, o que dificulta bastante a busca de uma função específica dentro deles, deixando o desenvolvimento mais lento. Um bom exemplo de um caso em que isto não ocorre é na pasta `conexao`, onde há diversos programas mais curtos.

Mais um problema a ser citado é o do "esquema" bastante usado no sistema (programa principal, cujos elementos HTML chamam funções JavaScript em `ajax.js`, que chamam programas auxiliares no diretório `conexao`). Apesar de isto estar bem implementado, conforme o número de programas auxiliares e o tamanho de `ajax.js` aumentam, a opção de deixar as funções AJAX e os programas auxiliares mais "próximos" dos respectivos programas principais torna-se mais interessante. Com eles juntos (ou agrupados por tema), seria mais fácil achar os arquivos para modificar na manutenção ou desenvolvimento de novas funções. Também seria possível, com isso, melhorar a questão dos arquivos JavaScript grandes demais.

Uma característica comum que aparece em diversos pedaços da estrutura e dos programas do sistema, que foi um dos focos neste processo de melhoria vinculado ao desenvolvimento da versão nova, é a inconsistência. Há várias abordagens organizacionais diferentes dentro do mesmo projeto, diretório ou até arquivo.

Um exemplo bem evidente é o fato de haver vários programas pequenos para cada ação no diretório `conexao`, e dois programas muito grandes reunindo diversas funções em `js`. Outro exemplo é que apesar de as funções em JavaScript estarem em sua maioria nos arquivos em `js`, às vezes aparecem dentro dos arquivos PHP dos programas e páginas. Mais um exemplo é o programa `User.php`, em `conexao`, que cuida de várias funções relacionadas ao usuário (cadastro, mudança de senha e de nível etc.). Este programa foi feito em um estilo orientado a objeto, ao contrário de todos os outros programas e funções que tratam das demais entidades.

Principalmente neste último caso, mas possivelmente em outros também, este conflito de abordagens tem como principal causa o fato de que os colaboradores do projeto mudam

constantemente. Sem uma definição clara de qual direção organizacional seguir, o projeto tem diferentes partes com estruturas norteadas por diferentes ideias. Assim, foi importante neste trabalho de reescrita não só tentar adaptar as partes seguindo uma visão mais fixa de organização, como também detalhar os princípios desta visão para, no futuro, ela servir de guia para a continuação do desenvolvimento.

Capítulo 4

Planejamento e Desenvolvimento Inicial

Com as mudanças de estrutura necessárias por conta do funcionamento do Django e os planos de realizar outras alterações para melhorar a organização do sistema, a definição das ideias por trás desta reorganização foi uma das prioridades desde o início do trabalho. Apesar de algumas ideias, em forma mais básica, já terem sido pensadas antes de se começar o desenvolvimento, elas se solidificaram com o início real da reescrita do sistema novo.

Um dos princípios da organização do sistema em Django, fortemente relacionado ao uso do próprio *framework*, é a divisão do sistema em algumas "categorias", os *apps* do Django. Com base na divisão estabelecida, foi definida a estrutura de cada *app* e, assim, a do sistema. Um outro passo muito importante, que permitiu a prototipagem de estruturas e ideias, foi o teste com um único *app*, contendo um subconjunto dos modelos. Após este teste, foi possível criar os outros *apps* e os modelos relacionados. Os detalhes destes passos são descritos a seguir.

4.1 Separação em *Apps* do Django

Como explicado anteriormente, um *app* em Django é uma divisão de um projeto maior (como um site completo). Assim, é natural que, dado o uso do Django nesta versão nova do sistema, seja criada também uma divisão em *apps*.

Em relação a funcionalidades, é fácil enxergar algumas divisões no sistema do NEHiLP. Como mostrado no capítulo anterior, existe uma distinção mais evidente entre páginas de administração, de conteúdo, do fórum e das funções "principais". O fato de essa separação ser já clara no sistema facilitou o processo de definição dos *apps* da versão nova.

A divisão escolhida foi: administração, conteúdo, obras e principal. Os *apps* **administracao** e **conteudo** não necessitam de muita explicação, reunindo, respectivamente, as páginas e funções de administração (perfil, mudança e redefinição de senha, *login*/*logout*/cadastro, lista de usuários etc.) e as de conteúdo estático (descrições dos projetos, *links* para atas de reuniões, agenda, informações de contato e outras). Quanto aos modelos/tabelas do banco, **administracao** inclui **usuario** e **email**, e **conteudo** inclui **conteudo** e **banner**.

Os *apps* **obras** e **principal**, por outro lado, requerem um pouco mais de explicação. Os modelos/tabelas em **obras** são **obra**, **autor**, **editora**, **cidade**, **obra_autor** e **obra_editora**. Este conjunto foi separado do resto das tabelas relacionadas por dois motivos principais. O primeiro é que, como a adaptação do Moedor não está nos objetivos do trabalho, não faria sentido criar um *app* para ele e, sendo assim e considerando a relação próxima entre os dados do Moedor e os das obras e das tabelas relacionadas, um *app* de obras poderia no futuro receber os modelos e programas relacionados ao Moedor. Assim, por enquanto, este *app* contém apenas páginas de busca/edição de obras.

Outro motivo é que as tabelas contidas em **obra** formam um subconjunto praticamente fechado, ou seja, elas não têm relações com nenhuma tabela fora deste grupo, com a exceção da relação **obra** → **contexto**. Isto facilita a implementação de um único *app*, incluindo apenas os modelos relacionados a ele e alguma página de teste para garantir que a integração com o banco funciona. Assim, considerando que os planos para o início do desenvolvimento já previam um caso assim, de criar primeiro alguns modelos e testar a comunicação com o banco, decidiu-se por ter um *app* **obras** separado.

Por fim, o *app* **principal** contém os modelos/tabelas **acepcao**, **acepcao_acepcao**, **acep_ultra**, **colaboracao**, **contexto**, **datacao**, **flexao**, **hemilema**, **hemilema_acepcao**, **hiperlema**, **hiper_ultra**, **historicoacepcao**, **historicoflexao**, **metalema**, **ocorrencia** e **variante**. As páginas de busca e edição/inserção, além das páginas de programas com exceção do Moedor (Concordanciador, Papavero e Metaplasmador) também fazem parte deste *app*. Um detalhe importante sobre os *apps* é que relações entre modelos de *apps* diferentes são permitidas, assim como um *app* acessar uma classe, função ou qualquer arquivo de outro. As divisões ajudam na organização, mas a interação não só é possível como ocorre em vários casos.

4.2 Estrutura de Arquivos e Diretórios

Considerando a divisão nos quatro *apps*, a estrutura de arquivos, diretórios e programas pensada para a versão do sistema em Django busca aproveitar as estruturas básicas de *apps* em Django e melhorar a organização em relação ao sistema atual. Para entender a nova estrutura, primeiro explica-se qual é a estrutura padrão de um *app* em Django e depois são explicadas as outras ideias que completam o modelo adotado.

4.2.1 Estrutura de um *app*

Assim como grande parte dos componentes que formam o Django, os *apps* são bastante adaptáveis. Dessa forma, não há uma única estrutura que eles têm que seguir; desde que cumpram certas regras, funcionarão corretamente independentemente da estrutura. No entanto, existe uma organização *default*, e que é (com variações menores) a mais usada.

Esta estrutura inclui, na variante mais básica, os arquivos `urls.py`, `views.py`, `admin.py`, `models.py` e os diretórios `templates`, `static` e `migrations` (os conceitos representados por estes arquivos estão entre as definições feitas em 2.2.1). Além deles, existem os arquivos `test.py` (para testes), alguns arquivos de configuração que não costumam ser modificados (como `apps.py`) e outros que aparecem de acordo com as necessidades do *app* (como `forms.py`).

Por fim, um último detalhe é que se arquivos como `views.py` ou `urls.py` estiverem muito grandes ou confusos, eles podem ser separados em diversos arquivos, cada um agrupando conteúdo relacionado, por exemplo.

4.2.2 Organização Proposta

Dessa forma, a organização proposta para a versão nova do sistema passa por dois pontos principais. O primeiro é a organização padrão dos *apps* no Django, descrita acima. Já o segundo pode ser basicamente definido por: tentar evitar arquivos muito grandes, separando grupos de funções em arquivos menores; e deixar arquivos relacionados juntos (respeitando a organização do *app*).

Com isso, chamadas AJAX relacionadas a um programa, por exemplo, ficariam dentro do diretório `static/js` do *app* onde está o programa; se o *app* tiver vários programas, poderiam haver subdivisões dentro dessa pasta `static/js`. Da mesma forma, uma função que mostra um *pop-up*, também relacionada ao mesmo programa, ficaria no mesmo diretório; se o número de arquivos e funções começar a crescer bastante, poderiam ser criados subdiretórios `util` e `ajax`, por exemplo, de modo a deixar a estrutura mais clara. Programas e funções em Python ficariam no diretório raiz do *app*, mas também poderiam ser separados em subdiretórios se estivessem aumentando demais em número ou ficando confusos.

No diretório raiz do projeto, além dos diretórios dos *apps*, existem alguns outros diretórios e arquivos, que representam elementos que são reaproveitados em mais de um *app*. Um exemplo é que existem no diretório raiz subdiretórios `static` e `templates`, guardando, por exemplo, arquivos CSS e *templates* como os responsáveis pelo menu, cabeçalho e rodapé, elementos presentes (sempre iguais) em todas as páginas do sistema. Também neste diretório raiz encontram-se outros arquivos mais relacionados a configurações, como o `settings.py`, com as configurações gerais do projeto, e o `urls.py` do projeto, que normalmente importa as URLconfs dos *apps* e também define URLs não relacionadas a nenhum *app* específico.

O exemplo abaixo ajuda a ilustrar uma estrutura como a proposta para a versão nova do sistema. O exemplo conta com nomes de *apps* e arquivos genéricos, sem qualquer relação com os reais. Outro detalhe importante é que ele é bastante simplificado, omitindo alguns arquivos como `admin.py`, `apps.py` e o conteúdo de diretórios como `static`, `templates` e `migrations` e outros *apps* e subdiretórios do diretório raiz.

- **app1**
  - `urls.py`
  - `views.py`
  - `admin.py`
  - `models.py`
  - **templates**
  - **static**
  - **migrations**
- **app2**
  - `urls_x.py`
  - `urls_y.py`
  - `views_x.py`
  - `views_y.py`
  - `admin.py`
  - `models.py`
  - **util**
    - `foo.py`
    - `bar.py`
  - **templates**
  - **static**
  - **migrations**
- `...`

No exemplo, é possível ver que ambos os *apps* contêm os arquivos padrões de *apps* do Django, como `urls.py`, `views.py` e outros, além de diretórios como `templates` e `static`. No entanto, enquanto o primeiro *app* é mais simples, composto apenas por estes arquivos, o segundo mostra um caso um pouco mais complexo, com a URLconf e as *views* divididas em dois aquivos cada; `urls_x` e `urls_y` e `views_x` e `views_y`, respectivamente, separam as URLs e *views* relacionadas a "x" e as relacionadas a "y". Além disso, este segundo *app* contém um diretório extra, `util`, com arquivos que guardam funções utilitárias.

4.3 Teste com *app* de Obras

A primeira tentativa de exportar os modelos para o Django e interagir com o banco foi feita com a criação do *app* **obras**. A seguir são descritos o processo de criação dos modelos a partir das tabelas do banco de dados original e o de criação de funções que realizam uma interação mínima com o banco, além das dificuldades que surgiram nestes processos e outras observações relevantes.

4.3.1 Modelos e Alterações

Para iniciar o processo da criação dos modelos, foi utilizada a ferramenta `inspectdb` do *script* de administração do Django [19]. Ela recebe nomes de tabelas, conecta-se ao banco de dados especificado em `settings.py` e gera uma sugestão dos modelos correspondentes. Evidentemente, a sugestão precisa de modificações, mas ela forneceu uma boa base inicial, acertando a correspondência entre quase todos os tipos do MySQL e os respectivos `Fields` do Django.

Após a criação destes modelos, foi criado um novo banco, com cópias das tabelas do banco original que fazem parte deste subconjunto. Além dele, foi criado mais um banco, para ser o usado pelo sistema em Django. Com este último configurado em `settings.py`, o *script* de administração do Django foi usado para criar as tabelas vazias de acordo com os modelos definidos. No entanto, no momento de copiar os dados das tabelas do banco original para estas tabelas recém criadas, alguns problemas (descritos abaixo) ocorreram. Com isso, o segundo banco, com a cópia das tabelas do banco original, foi usado para que fossem feitas algumas alterações nos dados de modo a resolver os problemas. Após estas alterações, os dados foram copiados do banco alterado para o novo.

Um dos problemas encontrados foi relacionado às chaves estrangeiras. Como já exposto, há alguns campos que logicamente correspondem a chaves estrangeiras mas não estão especificados como chaves no banco original. Nos modelos do Django, este problema foi resolvido marcando os campos correspondentes como `ForeignKey`, o campo para chaves estrangeiras.

Um detalhe é que, ao definir um campo `ForeignKey`, é preciso especificar seu atributo `on_delete`, que indica o que deve acontecer com a entrada quando a outra entrada (a que a chave estrangeira referencia) é apagada. Para chaves que podem ser nulas, foi usada a opção que deixa o campo nulo. Para relações em que não faz sentido manter a entrada após a relacionada ter sido apagada, foi usada a opção *cascade*, que apaga também a entrada que continha a chave (exemplo: uma entrada de **obra_autor** cuja obra é apagada não tem motivo para continuar existindo). Por fim, para relações em que a chave não pode ser nula e não faz sentido apagar a entrada quando a relacionada é apagada, foi usada a opção *protect*, que impede que a entrada relacionada seja apagada (exemplo: uma entrada de **obra** não pode ter seu `autorint`, o principal autor intelectual, apagado).

Outro problema encontrado em relação a chaves estrangeiras foi que algumas apontavam para entradas já removidas. Isso pode ter sido consequência direta de algum erro na definição do `ON DELETE` no banco original, mas também pode ter relação com testes feitos no início do desenvolvimento do sistema atual. Para resolver esse problema, as chaves que podiam ser nulas foram trocadas por `NULL`, enquanto os casos em que ela não podia foram analisados e resolvidos diretamente no banco original, antes de ser feita a cópia.

Mais uma questão relacionada a chaves que surgiu, dessa vez sobre chaves primárias, foi a das tabelas que representam relações $n \leftrightarrow m$, **obra_autor** e **obra_editora**. No banco original, estas tabelas têm chaves primárias compostas. No entanto, no Django, não é possível ter chaves primárias compostas, apenas unicidade composta (especificar que determinados campos sejam únicos em conjunto). Assim, a solução encontrada foi modificar as tabelas, criando uma chave primária `id` e deixando os campos que eram usados como chave primária composta com uma restrição de unicidade composta.

Um outro problema que apareceu várias vezes foi o das datas 0000-00-00. Em alguns campos, como a data de nascimento e a de morte de autores, muitas entradas continham esse valor. Embora ele seja teoricamente válido no MySQL como uma espécie de valor nulo, pode ser muito confuso se aparecer para usuários e foi decidido trocá-lo por `NULL`.

Com estes problemas corrigidos, foi possível copiar os dados para o novo banco criado. Um último detalhe relativo a isso é que houve outras duas mudanças nos modelos. Em primeiro lugar, foi adicionado em **obra** um campo `impressor_id`, seguindo o modelo dos campos dos outros autores principais, visto que o impressor já era citado no manual como um dos tipos possíveis de autor e inclusive figurava em algumas entradas em **obra_autor**; este campo foi deixado com valor nulo em todas as entradas existentes, podendo ser modificado depois para os casos em que já existem impressores. A outra mudança foi a remoção do campo `pseudonimo_id`, que representa uma entrada de autor que é um pseudônimo de outro. Este campo foi removido pois já existe o campo `pseudonimo` para guardar o pseudônimo (em forma de texto), e este é o único que é usado. Caso no futuro seja necessário usar um campo como o removido, com o Django será fácil inseri-lo de volta no modelo, bastando modificar o arquivo e aplicar uma migração.

4.3.2 Integração Básica

Para garantir que a comunicação (tanto a leitura quanto a escrita) com o novo banco criado estava funcionando corretamente, foram feitos dois passos: o primeiro foi a criação das páginas de administração para os modelos do *app* **obras**, enquanto o segundo foi a criação de uma versão simplificada da busca de obras.

As páginas de administração permitem que se veja as listas de entradas de cada modelo, inclusive seus detalhes, além de permitirem a modificação destas entradas e a criação de entradas novas. Por meio das páginas, foi possível ver que os dados estavam presentes e com os valores corretos e, principalmente, foi possível confirmar que a criação de novas entradas funcionava corretamente (e também a remoção).

A implementação da busca de obras, por sua vez, serviu para mostrar que consultas minimamente complexas ao banco também funcionavam corretamente, exibindo todos os resultados esperados e os detalhes associados. Esta versão inicial da página permite ver todas as obras ou buscar por um título (ou parte dele) e exibir os resultados de acordo com as opções de ordenação. Em relação à implementação do sistema original, ficaram de fora desta versão de teste alguns detalhes sobre a exibição das obras e também a função de edição de obras.

4.4 Criação dos *apps* Restantes

Após o sucesso nos testes feitos com o *app* **obras** e o subconjunto de modelos correspondente, o passo seguinte foi a criação dos outros *apps*. Este processo é descrito a seguir, incluindo os detalhes sobre a adaptação dos modelos e criação de páginas e funcionalidades simples de teste, com foco nos desafios e soluções encontradas durante o caminho.

4.4.1 Modelos e Alterações

Assim como no *app* das obras, o processo da criação dos modelos começou com o uso da ferramenta `inspectdb` do *script* de administração do Django. As sugestões foram analisadas e modificadas onde necessário e, após isso, foi feito o mesmo processo que no caso anterior, de criação de dois novos bancos, um com cópia dos dados para poder ser alterado e outro para criar as novas tabelas e usar no sistema em Django. Os problemas e outros pontos interessantes no processo de cópia, bem como as mudanças nos modelos em relação ao banco original são descritos abaixo para cada *app*.

App de administração

Para recriar os modelos de usuários, o processo foi um pouco mais complicado, devido ao caráter mais importante desta tabela e ao modo de funcionamento do Django. Diferentemente da maioria das partes do *framework*, a de usuários/autenticação é de difícil customização. A classe padrão `User` do Django costuma satisfazer as necessidades de boa parte dos sistemas. No entanto, ela tem seus próprios campos e não pode ser estendida. Para fazer uma classe personalizada, o trabalho extra é considerável, sendo necessário criar subclasses de outros componentes que participam deste processo da autenticação [20].

A solução recomendada e, aparentemente, mais usada, é a de criar um modelo a mais (geralmente chamado de `Profile`, um perfil), que armazena as informações adicionais desejadas sobre o usuário e tem uma relação `OneToOne` com o modelo User padrão (cada `User` é vinculado a exatamente um `Profile`). Isso garante o acesso fácil a estas outras informações e mantém a praticidade associada ao uso do modelo padrão. Por simplicidade, esta foi a opção escolhida, com o modelo de perfil sendo chamado de `Usuario` e contendo todos os campos originalmente na tabela **usuario**, exceto nome de usuário, nome completo, senha e e-mail.

Um dos problemas encontrado no processo tem relação com o armazenamento dos *hashes* das senhas. No sistema atual, as senhas passam por um algoritmo MD5 e o *hash* é armazenado. No entanto, o MD5 não é muito seguro, e o padrão do Django é usar o PBKDF2 com SHA256. Como o MD5 é suportado pelo Django, os *hashes* poderiam simplesmente ser copiados e a autenticação funcionaria corretamente. Mais ainda, quando os usuários fizessem *login*, o Django automaticamente atualizaria os *hashes* para os do PBKDF2. Apesar disso, há usuários que deixaram o projeto e provavelmente não vão fazer login nunca mais, então não teriam seus *hashes* atualizados. Levando isso em conta e com a vantagem de deixar o sistema ainda mais seguro, a opção escolhida foi a de usar o PBKDF2 como um *wrapper*, armazenando os *hashes* de sua aplicação em cima dos *hashes* do MD5 [21]. Graças ao Python e à documentação bastante completa do Django, esta conversão foi feita sem dificuldades.

Outra questão foi a do nome completo. No sistema atual, existe apenas um campo para o nome inteiro, enquanto no `User` do Django existem dois campos, `first_name` e `last_name`, ambos com tamanho máximo de 30 caracteres. Assim, para copiar os nomes, foram copiados os primeiros 30 caracteres em `first_name` e os restantes em `last_name`,

e depois foi rodado um *script* para "montar" de volta o nome, deixando o máximo de nomes/sobrenomes possível em `first_name` e o resto em `last_name`; de qualquer forma, sempre que é necessário obter o nome completo, foi usada a função `get_full_name()`, que funciona para ambos os casos. Um outro detalhe relacionado é que o código que ajusta os nomes também os normaliza, deixando todos os nomes/sobrenomes (menos partes como de/da/do(s)) com apenas a primeira letra maiúscula.

Por fim, outra modificação que teve que ser feita foi em relação aos nomes de usuário. Alguns deles continham espaços, algo não permitido pelo Django (e uma má prática no geral). Assim, os nomes de usuário também foram processados, removendo-se espaços no começo ou no fim e substituindo-se espaços no meio do nome por *underscores*. Além disso, esse processamento normalizou os nomes de usuário, deixando todos apenas com letras minúsculas e sem acentos. No *login*, não há diferenciação entre letras com ou sem acentos, nem entre maiúsculas e minúsculas; assim, escrever `joao` ou `João` não faz diferença, mas o padrão de letras minúsculas sem acento foi adotado pois é o mais comum e de melhor visualização. Para facilitar para os usuários cujos nomes de usuário continham espaços, o formulário de *login* inclui código JavaScript para trocar automaticamente espaços por *underscores* antes de enviar os dados para o servidor.

App de conteúdo

A criação dos modelos do *app* de conteúdo (apenas as tabelas **conteudo** e **banner**) foi simples, comparada às outras. Basicamente apenas duas dificuldades foram encontrados: mais datas 0000-00-00 e conteúdos vazios.

No caso das datas, várias entradas de **conteudo** continham 0000-00-00 como data da última atualização daquela entrada, indicando que ela nunca havia sido atualizada (ou possivelmente nunca depois de ter sido adicionada a função que marca a última atualização). No entanto, este campo da data da última atualização não podia ser nulo. A solução foi modificá-lo para poder ser nulo e trocar os valores 0000-00-00 por `NULL`. Isto foi feito diretamente no banco original, antes de se copiarem os dados.

Já a questão dos conteúdos vazios é que alguns conteúdos tinham a versão em inglês vazia (ou seja, uma *string* vazia), sendo que estes campos da versão em português e em inglês do conteúdo não podiam ser nulos. No Django, campos de texto nulos são representados pela *string* vazia. Assim, ao definir um campo de texto, dizer que ele pode ficar em branco equivale a dizer que ele pode ser nulo. Para resolver estes casos, a solução foi declarar que o campo pode ficar em branco, o que é um caso que realmente pode acontecer, por exemplo quando a versão em inglês ainda não foi preparada.

Um último detalhe em relação a este *app* é que o campo `atualizado_por`, que indica o último usuário que atualizou um conteúdo, estava definido no banco original como um campo de texto, mas guardava `ids` de usuários. No modelo em Django, o campo foi definido corretamente como uma chave estrangeira (que pode ser nula, para casos em que não há registros de quem foi o último a atualizar o conteúdo).

App principal

A criação do *app* **principal**, devido à quantidade maior de modelos associados, foi trabalhosa. No entanto, a maioria das dificuldades que apareceram já havia aparecido nos outros *apps* e, assim, foi relativamente simples superá-las.

Um dos primeiros problemas encontrados foi o de algumas inconsistências nos dados no banco, por exemplo: algumas acepções apontavam para entradas de **historicoacepcao** inexistentes, algumas entradas de **historicoflexao** tinham o campo de classificação morfológica

(que não pode ser nulo) com uma *string* vazia, e algumas entradas de **datacao** referenciavam ocorrências inexistentes. Estes casos (e mais alguns outros não citados) foram analisados e corrigidos diretamente no banco do sistema atual, antes dos dados serem copiados.

Outra questão foi, novamente, a das chaves estrangeiras que não estavam marcadas como chaves. Ela foi facilmente corrigida marcando corretamente os campos como `ForeignKey` na definição dos modelos do Django.

Também houve algumas outras modificações nos modelos em relação às tabelas originais. Em **colaboracao**, o campo `acepcao` foi removido, já que a contagem de colaborações em acepções estava sendo feita somente por meio das entradas em **datacao**. Além disso, os campos `hiperlema` e `ultralema` foram modificados para não poderem ser nulos e, ao invés de `NULL`, usar simplesmente o número 0, já que representam quantidades de colaborações.

Por fim, um último detalhe é que, no momento de copiar os dados, em alguns casos duas tabelas continham chaves estrangeiras uma para a outra, dificultando um pouco o processo de cópia. Para conseguir copiar os dados sem problemas, foi necessário primeiro copiar a primeira tabela, modificando os dados no campo que referenciava a segunda para serem todos nulos, e depois copiar a segunda e ajustar os dados do campo da primeira para ficarem com as referências corretas.

4.4.2 Integração Básica

Para todos os *apps*, um dos passos para verificar o funcionamento da integração entre páginas e o banco de dados foi a criação das páginas de administração para os modelos correspondentes. Com elas, foi possível ver os dados e principalmente testar a inserção e remoção. Além disso, algumas outras páginas foram criadas como testes em cada *app*.

App de administração

No caso do *app* **administracao**, ao invés de criar páginas diretamente acessíveis como teste, foi feito o código para permitir o *login* e o *logout*. Os elementos visuais (o formulário de *login* e o botão de *logout*) foram incluídos em todas as páginas do sistema, na barra superior, como na versão atual. Isso foi possível por meio de um `context_processor`: quando uma *template* é renderizada por uma *view*, ela recebe um contexto, ou seja, um conjunto de variáveis; um `context_processor` deixa determinadas variáveis acessíveis para todas as páginas, o que permite que o formulário de login seja acessível de qualquer ponto do site. Ao acessar uma página que precisa de *login* sem estar autenticado, o usuário é redirecionado para a página principal, e, se ele fizer *login* depois disso, será redirecionado para a página que havia tentado acessar. Ao fazer *login* em outra situação, o usuário continua na página em que estava.

App de conteúdo

Para testar a integração com o banco no caso do *app* **conteudo**, foram criadas as páginas de conteúdo editável. Uma função genérica e uma *template* relacionada foram reaproveitadas por todas as páginas desse tipo, tornando o processo bastante simples e permitindo que se garantisse que o funcionamento estava correto. Além disso, para mostrar o campo de edição do conteúdo da página, foi usado o TinyMCE, cuja integração no sistema (por meio da inclusão do arquivo JavaScript) foi realizada com sucesso também.

Em relação à versão do sistema atual, não foram feitas neste teste as funções que permitem ver e editar a versão em inglês (somente a versão em português era exibida), nem a implementação da verificação de última atualização e a página de edição do *banner*.

App principal

No caso do *app* **principal**, como os modelos associados são usados em páginas mais complexas, como as de busca e edição, o que foi feito para teste foram páginas genéricas que listavam as primeiras cem entradas de cada modelo, junto com algumas informações. Em conjunto com as páginas de administração, isto permitiu que se verificasse razoavelmente bem que os dados tinham sido copiados corretamente e as consultas ao banco funcionavam.

Capítulo 5

Desenvolvimento

Após o sucesso na criação dos modelos e na integração simples com o banco (um dos principais objetivos do trabalho), o desenvolvimento continuou, buscando atingir os outros objetivos de adaptação de páginas e programas. Primeiro foram feitas as páginas e funções restantes de administração. Depois, foram concluídas as páginas de conteúdo. Após isso, foram feitos alguns testes relativos ao processo de migração do sistema atual para o novo. Por fim, foram implementadas algumas das buscas mais simples (e a busca de obras melhorada) e, no final, voltou-se o foco para a adaptação de outros programas, como um dos principais (Concordanciador).

5.1 Administração

No desenvolvimento inicial do *app* de administração, haviam sido criados apenas os modelos e as funções de *login* e *logout*. Os passos seguintes foram a implementação das funções de cadastro, troca e redefinição de senha e algumas páginas, como a de produtividade e a de usuários. A seguir são descritos os pontos considerados mais interessantes deste processo.

Na criação da função de cadastro, aproveitou-se para melhorar a validação das informações, de modo a se tentar impedir a inserção de dados inválidos já na hora que o usuário tenta enviar o formulário, o que foi feito por meio de JavaScript. Mesmo assim, há também a validação no servidor, para impedir que dados inválidos sejam salvos. Além disso, para casos em que a validação só pode ser feita no servidor (por exemplo verificar se o e-mail já existe), são exibidas mensagens de erro após o usuário enviar o formulário e o servidor validar, para que o usuário saiba o que deu errado. Isso é feito por meio da função de mensagens do Django, em que uma *view* manda mensagens com um tipo (sucesso, erro etc.) e um nome, e a *template* renderizada consegue acessar as mensagens [22]. Na *template*, é definido código JavaScript que mapeia os nomes das mensagens para textos correspondentes e, assim, para cada mensagem, é exibido um alerta com um texto explicando o que aconteceu.

Um cenário comum no site é que uma página (ou certas funções/partes de uma) não sejam acessíveis para usuários de níveis mais baixos. No sistema atual, isto é implementado guardando-se o nível do usuário nos dados da sessão e verificando com código PHP dentro das páginas. Para a versão em Django, há algumas diferenças: em primeiro lugar, um objeto com o usuário está disponível nos dados das requisições (`request.user`) e, a partir dele, é possível obter o nível, permitindo a verificação de nível dentro das *templates*. Se é preciso apenas verificar se o usuário acessando a página está autenticado, existe uma função (`user.is_authenticated`), que permite checar isso na *template*. Nos casos em que a página inteira não é acessível para usuários que não fizeram *login* ou não têm o nível necessário, a própria *view* já impede o acesso. Isto é feito por meio de *decorators* na fun-

ção que é a *view*; para *login*, há o `@login_required`, e para condições mais complexas, o `@user_passes_test`, que recebe uma função anônima que retorna verdadeiro/falso e tem o usuário como argumento. Ambos foram configurados para redirecionar para a página principal em caso de falha [23].

Tanto para o cadastro quanto para a redefinição de senha, existe a necessidade de se enviar e-mails (para o usuário, em ambos os casos, e para o administrador também quando há um novo cadastro). No sistema atual, isso é feito por meio da biblioteca PHPMailer; já para a versão nova, o próprio Django inclui funções que permitem enviar e-mails de forma simples [24]. As funções responsáveis pelo envio de cada tipo de e-mail foram implementadas em um arquivo `email.py` dentro do *app* **administracao**, e são chamadas pelas *views* quando necessário.

Na redefinição de senha, é necessário gerar um *link* único (personalizado e que expira após ser acessado pela primeira vez) para que o usuário possa redefinir seguramente sua senha. O Django fornece um mecanismo para gerar esses *links*, que foi usado na implementação desta funcionalidade. Para comparar, no sistema atual o *link* gerado pode ser acessado diversas vezes sem deixar de funcionar, gerando um possível problema de segurança.

Na página de perfil do usuário, foram feitas algumas mudanças em relação ao sistema atual. Em primeiro lugar, os dados são realmente exibidos (no atual, eles aparecem em branco, provavelmente por conta de algum *bug*). Além disso, da mesma forma que com o cadastro, a validação foi melhorada e mensagens de alerta indicam o que está acontecendo. Por fim, também foi adicionada ao perfil a possibilidade de mudar a senha, algo ausente no sistema atual; a interface para isso é a padrão do Django.

Uma função que existe no perfil é a de o usuário verificar sua produtividade entre duas datas. Esta mesma função existe na página de produtividade (que é acessível apenas para administradores), para ver a produtividade de todos os usuários cadastrados como "produtivos" (os bolsistas, geralmente). Para ambas as páginas, a validação das datas foi melhorada, impedindo (tanto no cliente quanto no servidor) datas inválidas e mostrando por padrão a produtividade para o período entre 01/01/2013 e o dia atual. Outra mudança relacionada é que, na página de produtividade, a inserção de um novo usuário como produtivo teve sua interface melhorada: no sistema atual, aparece uma lista simples dos usuários, sem nada indicando que era possível clicar neles, mas bastava clicar para inserir; na versão nova, há botões indicando claramente que se está fazendo uma inserção.

5.2 Conteúdo

Para completar o desenvolvimento do *app* de conteúdo, foram implementadas as funções de mostrar e editar a versão em inglês das páginas de conteúdo e mostrar e atualizar o texto que indica o último usuário a editar uma página. Além disso, algumas páginas que antes tinham conteúdo estático mas não armazenado no banco e cuja edição não era permitida foram adaptadas para ficarem iguais às outras, com conteúdo editável.

Sobre as versões em inglês do conteúdo, elas aparecem automaticamente quando o *locale* do navegador está em inglês, mas também podem, como no sistema atual, ser ativadas/desativadas clicando no ícone de bandeira dos Estados Unidos/Brasil na barra superior. Já os menus e outros elementos com versão em inglês não foram adaptados por enquanto, mas, com o sistema de internacionalização do Django, deve ser possível implementar isso de forma mais simples e limpa do que no sistema atual (e também deve ficar mais fácil o processo de criar novas traduções para outros elementos/páginas do site) [25].

Uma mudança em relação ao sistema atual que foi implementada para as páginas editáveis foi uma otimização no momento em que o usuário seleciona a opção de salvar a edição: se o

conteúdo não foi alterado, ele não é salvo.

Um detalhe sobre a conversão de algumas páginas com conteúdo estático *hardcoded* para páginas de conteúdo editável é que a maioria delas continha imagens ou *links* para outros tipos de arquivos no meio do texto. Como nas *templates* em Django é boa prática usar a *tag* `static` para indicar caminhos para arquivos estáticos, foi preciso converter os caminhos para imagens e arquivos nessas páginas para usarem essa *tag*, permitindo que o Django achasse o caminho real.

5.3 Migração

Na fase de criação dos *apps* para a versão em Django do sistema, foi necessário modificar alguns dados do banco antes de copiá-los, como descrito no capítulo anterior (4.4.1). Para facilitar este processo no momento em que ocorrer a transição do sistema atual para o novo, as mudanças foram anotadas e, posteriormente, começaram a ser integradas dentro de um *script* de migração, que busca automatizar os passos necessários para transferir os dados e preparar o novo sistema.

Infelizmente, não foi possível agregar no *script* todas as ações e automatizá-las; as ações manuais necessárias antes ou depois de sua execução também foram anotadas para servirem como referência no momento da migração. Ainda assim, muitos dos passos foram automatizados.

A execução do *script* gera seis *scripts* auxiliares que realizam diferentes etapas da migração e são executados de dentro do principal. É possível também fornecer argumentos indicando que se deve apenas gerar (e não executar), ou até pular alguma(s) parte(s). Foram implementadas também pausas entre as partes e entre tarefas dentro de cada parte para perguntar ao usuário se ele quer continuar executando, além de indicadores de atividade para mostrar que as tarefas mais demoradas estão rodando e não travaram.

A primeira parte pede ao usuário o nome de um arquivo contendo um MySQL *dump* das tabelas e dados do banco do sistema atual e, a partir disso, cria uma cópia desse banco.

A segunda parte configura o `virtualenv`, instala as bibliotecas necessárias (inclusive o Django) e depois roda o *script* de administração do Django para executar as migrações (do Django), criando o banco que será usado pelo sistema e as tabelas de acordo com os modelos definidos (mas vazias).

Após a criação destes dois bancos, são feitas as modificações nos dados e a cópia deles para o novo banco, o que ocorre nas partes 3, 4, 5 e 6, uma para cada *app* do sistema em Django.

A terceira parte é a responsável pela alteração e cópia dos dados relativos ao *app* de obras. As principais alterações são a troca de datas 0000-00-00 e chaves estrangeiras referenciando elementos inexistentes por `NULL`, e a de campos de texto com `NULL` pela *string* vazia (o padrão do Django para representar texto nulo). Um detalhe importante é que, após copiar os dados de cada tabela, é preciso também mudar o `auto_increment` dela para garantir que está igual ao da tabela original. Isso tudo é feito com código SQL.

A quarta parte é relacionada à migração dos dados do *app* de administração. Além da cópia de dados, troca de `NULL` por *string* vazia e outras alterações desse tipo, são executados alguns outros *scripts* auxiliares em Python para normalizar os nomes de usuários e nomes completos e dar acesso de administrador aos usuários certos.

A quinta parte cuida da cópia dos dados do *app* de conteúdo. Também há troca de `NULL` por string vazia e outras alterações do tipo. Um *script* auxilar em Python é executado para corrigir as referências a arquivos dentro dos conteúdos, adicionando a *template tag* `static` junto com o caminho.

Por fim, a sexta parte é a responsável por fazer alterações e copiar os dados para o *app* principal. Apesar de este *app* ser o que tem mais modelos (e entradas), o procedimento é parecido com o dos outros *apps*, ocorrendo a troca de texto `NULL` por strings vazias, troca de referências a elementos inexistentes por `NULL` e outras alterações similares.

5.4 Buscas

Dentre as diferentes buscas presentes no sistema atual, as que foram adaptadas neste trabalho foram a busca morfológica (classificação morfológica), a semântica (classificação semântica), a de origem (ultralemas e ultralemas remotos) e a de obras, que já tinha sido parcialmente implementada na fase de desenvolvimento inicial e foi melhorada nesta etapa subsequente.

5.4.1 Busca Morfológica e Busca Semântica

A busca morfológica permite que o usuário escolha uma classificação morfológica (substantivo, adjetivo, verbo, gênero, número etc.) e retorna todas as flexões com aquela classificação. Os resultados incluem também a acepção à qual a flexão está vinculada e a abonação (o contexto da ocorrência). De forma similar, a busca semântica permite que o usuário escreva uma classificação semântica (o sistema oferece sugestões conforme se digita, de acordo com a lista de classificações usada) e retorna as acepções com aquela classificação, além da definição e abonação de cada acepção.

Devido a esta semelhança entre as buscas, foi possível reutilizar uma *template* de base para as duas (e também para as outras buscas, de origem e de obra). Essa *template* define uma parte da página com o formulário de busca e outra para mostrar os resultados, fixando aspectos como o estilo, e as *templates* de cada busca a complementam, fornecendo detalhes necessários sobre o que aparece no formulário e em cada resultado.

Durante a implementação da busca morfológica (a primeira a ser feita), surgiu uma questão bem importante e que apareceu também nas outras buscas e até em outras páginas do sistema: as buscas que devolviam um número grande de resultados estavam lentas, era preciso fazer alguma otimização. Por meio do uso da extensão Django Debug Toolbar [26], foi possível descobrir que o problema era que havia inúmeras consultas repetidas.

A causa deste problema é a maneira como o Django trata as consultas. Considerando o seguinte exemplo: na *view* é feita uma consulta que busca as flexões cujas classificações morfológicas sejam iguais à selecionada na busca, e na template existe um *loop* iterando por essas flexões e, para cada, buscando a acepção à qual ela está relacionada. Em cada iteração do *loop*, o Django faria uma consulta nova para pegar os detalhes da acepção relacionada à flexão, resultando em n consultas, quando poderia fazer uma só, já que a lista de flexões já é conhecida antes do início do *loop*.

O problema foi resolvido usando as funções `select_related` e `prefetch_related`. Ambas servem para otimizar as consultas, permitindo que se especifique relações e atributos de modelos relacionados para serem carregados junto com a consulta. Enquanto a primeira função permite selecionar atributos de modelos para os quais o modelo da consulta tem chaves estrangeiras, a segunda permite seguir relações reversas, ou seja, carregar informações sobre modelos que têm a relação da consulta como chave estrangeira. Por exemplo, as flexões têm uma chave estrangeira para acepção, enquanto as variantes têm uma para flexão. Numa consulta que retorna flexões, `select_related` pode ser usado para carregar dados das acepções que as flexões referenciam e `prefetch_related` para carregar dados das variantes que referenciam as flexões [27] [28].

Para casos em que é necessário percorrer relações com maior complexidade, como acontece, por exemplo, para achar a abonação de uma acepção (é preciso, basicamente, descobrir qual é a flexão com a variante que tem a ocorrência mais antiga), é possível usar a classe `Prefetch` [29], que permite pré-carregamentos com mais opções do que a função `prefetch_related`. Os objetos desta classe aceitam nomes de campos e relações, da mesma forma que a função, mas também podem receber uma consulta já filtrada para servir de fonte para carregar os dados, e um nome de atributo para guardar estes dados no objeto contendo o resultado da consulta. O objeto `Prefetch` foi usado na busca morfológica e o modo como ele foi usado foi adaptado com relativa facilidade para uso na busca semântica. Assim, acredita-se que as implementações destas buscas devem servir como uma boa referência para demais situações que demandem o uso de um padrão como este.

5.4.2 Busca de Origem e Busca de Obras

A busca de origem permite que o usuário busque por ultralemas e veja, nos resultados, os comentários sobre a etimologia deles ou exemplos de palavras que os têm como origem. Os resultados podem ser apagados ou editados por usuários de níveis mais altos. Já a busca de obras permite que se busque por título ou outras informações de obras e mostra dados como autor, local de publicação e localização do texto. Nela, os resultados podem ser editados também, mas, na implementação feita, ainda não podem ser apagados.

No aspecto visual/organizacional, estas duas buscas seguem o mesmo esquema das outras implementadas, com uma área com um formulário e outra com os resultados (assim, a *template* de base foi de novo reaproveitada). Já na parte do funcionamento, a possibilidade de edição é uma grande diferença. No caso da busca de origem, a edição leva a uma nova página; no caso da busca de obras, a edição é feita em um *pop-up* na própria página da busca.

A página/área de edição também é composta por um formulário, com os campos (ou parte deles) do modelo em questão. O Django possui várias funções e classes para criar e manipular facilmente formulários [30]. Uma das classes, `ModelForm`, fornece os campos do modelo especificado e uma maneira fácil de salvar alterações em um objeto deste modelo. Este tipo de classe foi usado em ambas as buscas, com bons resultados.

No entanto, no sistema atual estes formulários de edição permitem também a edição/inserção de alguns campos de modelos relacionados (por exemplo, editar o autor de uma obra, ou adicionar um novo ultralema para ser o ultralema remoto do que está sendo editado). Usando o `ModelForm` do Django, os campos que representam chaves estrangeiras aparecem como listas para o usuário escolher um valor dentre os existentes daquele modelo, mas sem a possibilidade de inserir uma nova entrada ou editar as propriedades de uma que já existe. Para fazer isso, seria necessário incluir novos campos no formulário e tratá-los individualmente na *view* ou combinar mais de um formulário em uma única interface.

Além de o problema não ser muito simples, ainda mais em casos como o da obra, em que podem haver múltiplos autores e editoras, a solução no sistema atual é confusa, variando de caso para caso e às vezes nem funcionando corretamente. Considerando isso e pelo fato de o problema ter sido encontrado no final do trabalho, quando já havia pouco tempo restante, a decisão foi por deixá-lo em aberto, com o entendimento de que seria melhor chegar a uma solução boa e definitiva no futuro do que tentar resolver algo rapidamente, sem muita reflexão.

No caso das obras, o campo da editora aparece no formulário de edição, mas sem poder ser alterado, enquanto nenhum dos campos de autor aparece no formulário (o autor intelectual principal aparece nos resultados). Já na busca de origem, o campo de ultralema remoto

é editável, mas só é possível escolher um ultralema dentre os que existem, sem nenhuma maneira de inserir um novo.

Um último detalhe é que a possibilidade de remover obras, apesar de existir no sistema atual, não está finalizada nele, e é algo bastante complicado (mas importante). Sua implementação na versão nova ficou também para ser feita no futuro.

5.5 Outros Programas

Além das páginas e programas descritos nas seções anteriores, foram adaptados o Metaplasmador e partes do Concordanciador. O Metaplasmador foi escolhido para ser adaptado principalmente pela simplicidade, já que não inclui consultas ao banco nem se comunica com o servidor. Já o Concordanciador foi escolhido para ser adaptado por ser importante e reunir vários tipos de funcionalidade em um só lugar.

Como esperado, o processo de adaptação do Metaplasmador, na verdade, foi praticamente apenas uma cópia, já que ele não conversa com o banco e sua lógica de programação está toda em JavaScript, havendo apenas alguns elementos estáticos da página para definir em uma *template*. Uma mudança feita em relação ao sistema atual é que os arquivos que o compõem foram mais bem organizados, eliminando alguns arquivos CSS e JavaScript que não eram usados, e separando os arquivos para ficarem em diretórios adequados (no sistema atual havia alguns arquivos JavaScript que estavam dentro do diretório `css` e outros casos parecidos).

Outro detalhe sobre a adaptação do Metaplasmador é que um dos programas e um arquivo CSS tiveram que ser criados como *templates*, e incluídos dentro das respectivas *tags* HTML (`script` e `style`) na *template* principal. Isso foi necessário porque esses dois arquivos continham referências a outros arquivos estáticos (no caso do programa, aos arquivos de texto com as regras das transformações e, no caso do CSS, a uma imagem), e, para o Django localizar corretamente estes arquivos estáticos, é preciso usar a *template tag* `static`. Um caso parecido ocorreu também na implementação do *login*/cadastro. Embora a solução aplicada resolva o problema, é interessante ficar atento a estes casos para pensar em soluções mais elegantes para serem implementadas no futuro, de modo a evitar que uma única linha referenciando um arquivo estático obrigue um arquivo que também seria estático a ser transformado numa *template*.

A adaptação do Concordanciador era bem importante, mas considerada bastante complexa desde o início, até pela variedade de funções contidas nele. Olhando para estas funções, a implementação foi dividida em três partes: a busca pelas ocorrências, os gráficos estilo N-Gram, e as inserções de variantes, flexões e acepções a partir dos resultados. Devido ao fato de a adaptação ter começado já no final do trabalho, e também considerando que a parte das inserções e edições apresenta o mesmo problema discutido anteriormente de formulários com campos de modelos relacionados, foi decidido não implementar, neste trabalho, esta terceira parte do Concordanciador.

Para implementar a primeira parte, da busca por ocorrências, não houve muita dificuldade. Um ponto a ser mencionado é que, como a busca permite que o usuário use asteriscos como coringa, foi usada a sintaxe de expressões regulares do MySQL para fazer a pesquisa usando esses coringas (já que, usando apenas o filtro `contains` do Django, não seria possível, por exemplo, usar coringas no meio da palavra). Outro detalhe é que há um tratamento especial no caso de mesóclises, e as expressões regulares relacionadas a isso pareciam complicadas, mas acabaram sendo adaptadas facilmente, já que a sintaxe de expressões regulares do Python é parecida com a de PHP (ambas são baseadas na sintaxe de Perl).

Na adaptação da segunda parte (a geração dos gráficos parecidos com o N-Gram, mostrando a frequência das ocorrências ao longo do tempo), também não houve nenhum grande problema. Como o código de geração dos gráficos era quase todo JavaScript, ele pôde ser copiado, sendo necessário apenas implementar na *view* a coleta dos dados sobre a frequência da palavra em cada período. Uma mudança em relação ao sistema atual foi que a opção que deixava o gráfico em forma de curva foi removida, com os pontos sendo ligados agora por retas.

Outra diferença em relação à versão atual é que, no geral, o número de consultas ao banco diminuiu. Enquanto no sistema atual há uma consulta para saber a quantidade de resultados, outra para buscar os resultados da página atual, mais uma para achar as datas da ocorrência mais antiga e da mais recente, ainda outra para cada um (dos dez) intervalos de datas para saber quantas ocorrências aconteceram nele, e mais uma para achar a acepção à qual a ocorrência pertence, na versão em Django foi possível reduzir isso para apenas duas consultas. A primeira busca as ocorrências do termo pesquisado e já pega os dados de variante/flexão/ocorrência e contexto/obra, enquanto a segunda busca as ocorrências, mas ordenando por data. O resto do processamento (por exemplo, verificar quantas ocorrências são de um dado período), é feito diretamente em Python, diminuindo a carga no banco.

A última parte da implementação do Concordanciador consiste em permitir que o usuário veja e edite/insira os dados sobre variantes, flexões e acepções relacionadas às ocorrências. Como já explicado, esta parte não foi feita. No entanto, na parte da busca das ocorrências, já foi feita a função que destaca, no contexto, as ocorrências e (com outra cor) as ocorrências que estão no DELPo. Assim, quando a parte de edição/inserção estiver feita, basta modificar a geração do código HTML para incluir chamadas às funções em JavaScript que mostrarão os *pop-ups*.

Capítulo 6

Desenvolvimento Futuro e Conclusões

Com o trabalho terminado, é possível destacar pontos importantes para o desenvolvimento futuro do projeto, tanto pensando no curto prazo quanto em mais para a frente. Além disso, é possível tirar conclusões sobre o processo de adaptação como um todo, fazendo um saldo do que foi planejado e o que foi feito e retomando as novidades e diferenças entre o sistema atual e a nova versão construída.

6.1 Desenvolvimento Futuro

Considerando as funcionalidades e programas do sistema atual e o que foi implementado na versão em Django, um dos primeiros passos para a continuidade do projeto é completar a reescrita/adaptação, garantindo que a versão nova contenha todas as funções presentes na versão atual. Em seguida, será preciso fazer a transição de um sistema para o outro, para que a versão em Django possa começar a ser utilizada. Feito isso, poderá se pensar melhor em novas funções para o sistema e começar a implementá-las.

6.1.1 Adaptação do Sistema Atual

A prioridade mais imediata para o desenvolvimento futuro do projeto é completar o processo de adaptação. Para isso, é preciso implementar no sistema em Django as funções e páginas restantes, além de procurar soluções para alguns problemas que surgiram durante a adaptação das outras funções e não puderam ser resolvidos.

Dentre os programas principais, é preciso terminar a adaptação do Concordanciador e implementar o N-Gram, a inserção de ultralemas, o Papavero e o Moedor e as páginas de sugestão e inserção de obras. Além disso, é preciso implementar as buscas restantes (dicionário, lema, abreviaturas, geral e datas), algumas funções de edição/inserção (como os *pop-ups* para acepções, flexões e variantes), a página com o formulário de "fale conosco", a página principal e as traduções de menus e outras partes do site que não são conteúdos editáveis.

A finalização da adaptação do Concordanciador consiste basicamente em implementar os *pop-ups* de edição e inserção de acepções, flexões e variantes e integrá-los aos resultados da busca, permitindo ao usuário a edição/inserção.

A implementação do N-Gram não deve ser muito complicada, já que o código para gerar os gráficos na versão atual é relativamente simples. Será preciso decidir como desenhar o gráfico, já que as implementações na página do próprio N-Gram e na do Concordanciador são diferentes no sistema atual, com a primeira mostrando pontos não conectados e a segunda

conectando eles em formato de curva (existe também a possibilidade de conectar sem deixar curvo, como foi feito no Concordanciador na versão nova e é feito na versão do Google).

A inserção de ultralemas também deve ser relativamente simples de ser implementada, principalmente considerando sua similaridade com a edição de ultralemas, que já foi feita. No sistema atual, o mesmo programa cuida dos dois casos; pensando em organização, talvez seja melhor ter duas *views*/*templates* separadas na versão em Django, mas certamente será possível reaproveitar várias partes do código entre elas.

A adaptação do Papavero deve ser uma das mais complicadas, já que ele envolve a inserção de dados em várias tabelas. Além disso, a interface precisa de melhoras, principalmente nos campos de inserção de obras, que só aparecem quando necessário, mas ficam fora de lugar dependendo do tamanho da tela e do navegador usado.

A adaptação do Moedor também deve ser complexa, mas é bastante importante para o sistema. Uma possibilidade para facilitar o trabalho é dividi-la em duas etapas. A primeira consistiria em fazer o Moedor atual (em Perl) se comunicar com o resto do sistema em Django, criando as *views* e páginas necessárias e mudando as referências no código do Moedor. Dentre as páginas necessárias estão a de sugestão e a de inserção de obras; enquanto a primeira não interage diretamente com o Moedor, a segunda já envia o texto para ser moído. A parte final da adaptação do Moedor seria a conversão do código em Perl para Python e a integração completa dele no sistema, algo mais trabalhoso considerando o tamanho do programa e a sintaxe de Perl.

Existe também um problema mais geral cuja solução está diretamente ligada a algumas das adaptações restantes. É o problema citado anteriormente, de formulários para edição/inserção de dados de um modelo que incluem campos de modelos relacionados. Para prosseguir com a implementação de páginas como algumas das buscas, o Concordanciador e a inserção de obras, é preciso achar uma solução boa para esse problema; o lado positivo é que a solução poderá se transformar em um padrão a ser reutilizado em várias partes do sistema.

Em relação às buscas que ainda têm que ser implementadas, a de abreviaturas deve ser a mais simples, já que as abreviaturas contêm pouca informação para ser mostrada e os resultados da busca não são editáveis. A busca de lema (que faz a pesquisa de hiperlemas) também não tem resultados editáveis, mas é um pouco mais complexa, pois envolve a exibição dos verbetes do DELPo.

A busca geral é bem complexa, tanto por permitir o uso de expressões regulares para a pesquisa quanto por incluir vários tipos de resultados e bastante informação sobre eles. Por fim, a busca dicionário e a busca por data têm uma quantidade considerável de semelhanças, mostrando vários dados sobre acepções, flexões, variantes e ocorrências e permitindo a edição deles. Por isso, a implementação deve ser mais complicada, mas provavelmente será possível reutilizar partes do código.

A página com o formulário de "fale conosco" deve ser bem simples de ser adaptada, bastando criar uma *template* com o código HTML do formulário e configurar o envio do e-mail com o conteúdo inserido.

A adaptação da página principal não deve oferecer muita dificuldade. Uma página provisória já foi criada, contendo apenas o texto dos itens do *banner*. A página completa contém estes itens em um formato mais estilizado, além de outros textos e *links*. Como durante o período em que o trabalho foi feito estavam sendo discutidas possíveis mudanças no conteúdo/layout da página, sua adaptação completa não foi feita.

Por fim, a tradução das partes do site que não são conteúdos editáveis também deve ser relativamente simples. Estas partes são basicamente os itens do menu e algumas outras páginas. O sistema de tradução/internacionalização do Django deve facilitar bastante o trabalho, permitindo que se criem arquivos com os termos e suas traduções e se usem funções

nas *views* e *tags* nas *templates* para fornecer automaticamente a versão no idioma identificado (ou escolhido pelo usuário), sem a necessidade de verificar manualmente o idioma para cada termo traduzido.

6.1.2 Migração de Sistema

Com as funcionalidades todas adaptadas, para poder usar o novo sistema será preciso fazer uma migração dos dados e configurar o Django para rodar no servidor em que o sistema atual está rodando. Os *scripts* de migração criados automatizam praticamente todo o processo de cópia dos dados, mas vai ser necessário também executar alguns passos manuais, tanto antes dos *scripts* quanto depois.

Um documento foi criado contendo os detalhes dos passos que devem ser executados antes dos *scripts* de migração. As etapas a serem executadas depois não foram estudadas em muito detalhe nem testadas, mas a documentação oficial do Django contém bastante informação sobre o processo de *deployment*. Abaixo os principais pontos dos passos antes da cópia de dados e da execução dos *scripts* de cópia são explicados, sem incluir muitos detalhes técnicos.

A primeira ação que precisa ser feita é a instalação do Python e do `virtualenv`. A versão de Python usada é o Python 3, sendo necessário no mínimo a versão 3.2 por causa de algumas funções usadas nos *scripts*.

Além do Python, o MySQL (versão 5.6.6 ou mais recente) precisa estar instalado, mas provavelmente já estará, pois o sistema atual já o utiliza. É necessário também verificar se as configurações do MySQL estão marcadas para aceitar UTF-8 como padrão, algo que também já deve ter sido feito por causa do sistema atual.

Para obter os arquivos com o código do sistema em Django, é preciso apenas clonar o repositório no Git da USP onde o código está hospedado. Até o momento em que este trabalho foi concluído, o repositório não estava público, mas o plano é que ele se torne público no futuro próximo, facilitando o acesso para novos desenvolvedores e interessados em geral.

Para rodar os *scripts* de migração, basta executar o arquivo `migracao.py`, que gera e executa as diferentes partes da migração, além de apresentar instruções e detalhes de progresso na tela. Se ocorrer algum erro, é possível rodar novamente e escolher pular as partes já executadas com sucesso.

Após essas etapas, será preciso executar os passos necessários para fazer o sistema rodar no servidor. Informações sobre elas podem ser encontradas na documentação do Django, com vários exemplos, guias e dicas para garantir que tudo funcione de forma correta e segura [31].

6.1.3 Novas Funcionalidades

Após a cópia dos dados e a configuração do sistema no servidor, espera-se que o projeto entre em uma nova fase mais estável, o que proporcionará a oportunidade de aproveitar a estrutura nova e as facilidades de Python/Django para implementar novas funcionalidades no sistema.

Uma destas funções novas planejadas, já discutida nas reuniões do projeto há certo tempo, é a expansão do N-Gram. Como descrito anteriormente, a versão atual do N-Gram mostra o gráfico com a quantidade de ocorrências da(s) palavra(s) pesquisada(s) ao longo do tempo. No entanto, para comparar a quantidade de ocorrências de uma flexão ou acepção, ou até metalema e hiperlema, seria preciso pesquisar cada uma das variantes. A expansão planejada permitiria que se buscasse pela flexão/acepção/lema, deixando a ferramenta bem mais poderosa (inclusive uma função como essa não existe na versão do Google).

Outra função já planejada e até parcialmente implementada é a remoção de obras. Devido à estrutura das relações entre os modelos, remover uma obra é algo complicado, pois é preciso analisar as ocorrências relacionadas a ela, e as variantes relacionadas às ocorrências, as flexões relacionadas às variantes e assim por diante, removendo ou atualizando a estrutura de acordo com o caso.

Um programa para fazer isso já existe, o chamado Faxineiro/Removedor. No entanto, ele ainda tem alguns problemas e não está sendo muito usado; inclusive, as inconsistências encontradas no banco no começo da criação da nova versão muito provavelmente estão relacionadas com problemas na remoção de obras ou entradas de outros modelos. Com o ORM do Django e as funções de Python, espera-se que seja mais fácil fazer uma versão completa e funcional do Faxineiro.

Outra funcionalidade nova que seria bastante útil no sistema é uma interface melhor para edição de obras, autores, editoras e, no geral, casos de relações $n \leftrightarrow m$. A edição de obras se dá, por enquanto, por meio de *pop-ups* na página de busca de obras, e a edição de autores e editoras é feita somente por meio desses próprios *pop-ups* com as informações das obras. Seria útil ter uma página dedicada à edição de obras, e também outras com foco exclusivo na edição de autores/editoras.

Já sobre as relações $n \leftrightarrow m$, no geral, também seria muito bom desenvolver uma interface que permitisse uma edição simples e intuitiva dessas relações, ao invés das soluções atuais em que, por exemplo, é possível clicar em um botão para adicionar um novo autor a uma obra e os campos de edição vão se acumulando na tela desorganizadamente. A interface de administração do Django possui um esquema de edição de relações $n \leftrightarrow m$ que pode servir como inspiração para implementar esta função.

Por fim, uma observação a ser feita é que o *front end* como um todo se beneficiaria bastante de uma refatoração/reestruturação. O código JavaScript usa JavaScript puro e jQuery; no entanto, existem atualmente vários *frameworks* mais modernos e interessantes, como React, Angular e outros. Uma possível adaptação para usar um *framework* desses deixaria esta parte do código também mais atualizada, legível e facilitaria a manutenção.

Na parte do CSS (e HTML), apesar de o Bootstrap (que é usado) ainda ser bastante popular, várias páginas foram feitas com padrões de HTML e de estilo que estão já datados. Seria bastante positivo, principalmente para o usuário e considerando que o sistema é também o site do projeto, adaptar o código para usar HTML5, por exemplo, e também atualizar o design das páginas. Paginação também é algo que está muito pouco presente no sistema e seria útil se aparecesse em mais lugares.

O Formoid, usado para manipular e estilizar os formulários no sistema, é uma biblioteca também mais antiga, e a parte JavaScript dela causa vários erros no console do navegador. Embora sejam erros que aparentemente não atrapalham o funcionamento, o fato de eles aparecerem não é um bom sinal. Como a biblioteca está presente em muitas partes do sistema, não haveria tempo para tentar substituí-la neste trabalho, já que existiam muitas outras tarefas com prioridade maior. No entanto, em uma possível refatoração de *front end* seria bom buscar um substituto.

Evidentemente, novas ideias de funcionalidades surgirão com o tempo e serão analisadas somente no futuro, pelos colaboradores que estiverem atuando no projeto no momento. Os pontos descritos aqui são apenas observações, sugestões e possíveis aplicações mais diretas do trabalho feito.

6.2 Conclusões

Considerando os objetivos iniciais do trabalho, de iniciar a adaptação do sistema do NEHiLP para Python/Django e de deixar pronta uma boa base para a continuação deste processo no futuro, é possível dizer que o trabalho foi bem sucedido, atingindo praticamente todos os objetivos propostos e até indo além deles em alguns pontos.

A criação dos modelos em Django a partir das tabelas do banco foi feita por completo e várias páginas e funcionalidades foram implementadas, inclusive algumas que não estavam nos planos iniciais (descritos na Introdução e em mais detalhes na seção 3.1.5), como a adaptação de mais de uma busca e (parcialmente) do Concordanciador.

Também em relação a estes objetivos, as únicas partes que não foram totalmente cumpridas foram a adaptação da página principal e a edição/inserção de algum modelo, feitas ambas apenas parcialmente.

No caso da página principal, foram incluídos apenas o menu/cabeçalho/rodapé e os eventos na forma de texto simples, também pelo fato de se estarem discutindo nas reuniões do NEHiLP mudanças no conteúdo da página. A edição, por sua vez, foi parcialmente implementada para obras e ultralemas, sobrando o problema dos campos de modelos relacionados (citado em 5.4.2 e 6.1.1). Sobre as inserções, não implementadas, é importante destacar que a inserção de ultralemas é muito parecida com a edição, então sua implementação deve ser simples, e também que é possível testar inserção pela interface de administração do Django.

Além disso, outra ausência que merece ser comentada é a de testes automatizados. Embora não estivessem explicitamente citados nos objetivos, a ideia era fazer alguns testes para verificar de modo simples o funcionamento dos programas e páginas. No entanto, os testes acabaram sendo adiados várias vezes e, no final, não houve tempo de fazê-los. Pensando no futuro do projeto, a documentação do Django parece ser uma boa fonte de informações sobre como escrever e executar estes testes [32].

É interessante também, para concluir, comparar o sistema atual e algumas de suas características com certos pontos das soluções implementadas neste trabalho para a versão nova. Com isso, é possível se ter uma noção melhor do que se perde e se ganha com a nova versão e como isso pode nortear a continuidade do trabalho no projeto.

Em primeiro lugar, algo que já era um dos objetivos foi a melhora na organização, deixando o código mais fácil de entender, o que, por sua vez, facilita a manutenção e o desenvolvimento de novas funcionalidades. Outra melhora é em relação à documentação; ela foi melhorada no código em si, com descrições gerais das funções e comentários em partes mais complexas do código. Além disso, a documentação do próprio Django deve ajudar bastante no desenvolvimento futuro do projeto, já que é bem completa e didática, e, como o Django é muito usado (mais do que PHP e Perl atualmente), deve ser mais fácil encontrar material sobre ele, tanto para resolver possíveis dúvidas quanto para estudar outros projetos e padrões.

Um fato a ser discutido é a questão da otimização. Como explicado em 5.4.2, é preciso usar algumas funções específicas e/ou fazer partes da filtragem de dados no próprio Python (ao invés de nas consultas) para obter uma boa performance no Django e não ter consultas redundantes ao banco.

Embora isso aumente um pouco a complexidade de se escrever as consultas, a impressão que se teve durante o trabalho foi que, mesmo assim, o ORM do Django faz com que seja mais fácil criar consultas do que com SQL puro. Como visto no caso do Concordanciador (5.5), também, o uso do ORM do Django pode reduzir bastante o número de consultas feitas, melhorando a performance. Um teste útil a ser feito quando o sistema novo estiver rodando é comparar o tempo de carregamento entre as duas versões na mesma máquina.

Uma das vantagens relacionadas ao que foi feito neste trabalho é a resolução das inconsistências encontradas no banco de dados. Além do fato de que eliminar inconsistências já é uma ação importante por si só, foi possível corrigir estes problemas diretamente no banco que está sendo usado no sistema atual, impedindo novos *bugs* relacionados a isso de ocorrerem nele e resolvendo alguns problemas que já haviam ocorrido por conta disso.

Um ponto interessante a ser mencionado é em relação ao ambiente de desenvolvimento. No sistema atual, existe no servidor uma cópia do sistema usada para testes de desenvolvimento (chamada de `devalfa`). Isso é muito útil, pois é possível testar modificações e correções com uma base de dados igual (ou quase) à real.

Na versão em Django, não existe nenhum conceito parecido. Entretanto, algo introduzido neste trabalho foi o uso de controle de versão (foi usado o Git da USP). Com isso, será possível para os desenvolvedores usarem *branches* ou outras funções do Git para testes similares aos que são feitos no `devalfa`. Mesmo assim, cada desenvolvedor teria que copiar os dados do banco para poder testar suas alterações com os dados reais.

Considerando isto, uma opção proveitosa para o futuro seria pensar em alguma forma de automatizar este processo, o que facilitaria o trabalho dos desenvolvedores. Da mesma forma, também pode ser positivo explorar a ideia de usar integração contínua para atualizar a versão em produção do sistema conforme novos *commits* forem feitos ou algo parecido.

Outra diferença entre o sistema atual e a versão em Django é a interface de administração. A do PHP é o PHPMyAdmin, que permite ver as tabelas e as entradas no banco, editá-las, fazer consultas, exportar dados e várias outras funções. Já a interface do Django apresenta um design mais moderno e simples e abstrai alguns dos detalhes mais técnicos, excluindo a possibilidade de fazer consultas diretamente com código SQL (ou Python). Ela é, no entanto, bastante customizável, o que permite, por exemplo, mostrar os campos dos modelos com nomes mais fáceis de se entender do que os nomes usados no banco.

Assim, é possível dizer que, embora a interface de administração do Django não seja tão poderosa quanto a do PHP, principalmente em termos de busca, ela tem também muitos pontos positivos, como ser mais fácil de usar e permitir a visualização, edição e inserção/-remoção de dados de maneira simples e intuitiva. Além disso, é possível customizar grande parte de seus componentes, o que significa que, se as funcionalidades mais avançadas de busca estiverem fazendo muita falta, deve ser possível implementar algo parecido.

É possível concluir, então, que a nova versão do sistema implementada cumpriu bem os objetivos propostos no início do trabalho e será de fato muito útil para o desenvolvimento futuro do projeto. As funções disponíveis no Django mostraram-se, no geral, boas alternativas para reimplementar os programas do sistema atual, muitas vezes facilitando o processo. Espera-se também que este trabalho possa servir tanto de registro como de guia para este processo de reescrita do sistema. Com isso tudo, o desenvolvimento do projeto deve tornar-se bem interessante e mais simples no futuro, beneficiando tanto os desenvolvedores como os usuários.

Referências Bibliográficas

[1] NEHiLP | Home. <http://www.nehilp.org>. Acessado em: 25/11/2017.

[2] NEHiLP | DELPo. <http://www.nehilp.org/~nehilp/conteudo/delpo.php>. Acessado em: 25/11/2017.

[3] NEHiLP | Retrodatação. <http://www.nehilp.org/~nehilp/conteudo/retrodatacao.php>. Acessado em: 25/11/2017.

[4] Django: The Web framework for perfectionists with deadlines. <https://www.djangoproject.com>. Acessado em: 25/11/2017.

[5] Manual do NEHiLP - USP. <http://www.usp.br/nehilp/infos/manual.pdf>. Acessado em: 25/11/2017.

[6] Applications | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/ref/applications/#projects-and-applications>. Acessado em: 25/11/2017.

[7] Models | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/topics/db/models/>. Acessado em: 25/11/2017.

[8] Templates | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/topics/templates/>. Acessado em: 25/11/2017.

[9] Managing static files (e.g. images, JavaScript, CSS) | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/howto/static-files/>. Acessado em: 25/11/2017.

[10] Writing views | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/topics/http/views/>. Acessado em: 25/11/2017.

[11] URL dispatcher | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/topics/http/urls/>. Acessado em: 25/11/2017.

[12] Making queries | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/topics/db/queries/>. Acessado em: 25/11/2017.

[13] django-admin and manage.py | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/ref/django-admin/>. Acessado em: 25/11/2017.

[14] The Django admin site | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/ref/contrib/admin/>. Acessado em: 25/11/2017.

[15] Migrations | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/topics/migrations/>. Acessado em: 25/11/2017.

[16] Virtualenv – virtualenv 15.0 documentation. <https://virtualenv.pypa.io/en/stable/>. Acessado em: 25/11/2017.

[17] pip – pip 9.0.1 documentation. <https://virtualenv.pypa.io/en/stable/>. Acessado em: 25/11/2017.

[18] Google NGram Viewer. <https://books.google.com/ngrams>. Acessado em: 25/11/2017.

[19] Integrating Django with a legcy database | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/howto/legacy-databases/>. Acessado em: 25/11/2017.

[20] Customizing authentication in Django | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/topics/auth/customizing/>. Acessado em: 25/11/2017.

[21] Password management in Django | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/topics/auth/passwords/>. Acessado em: 25/11/2017.

[22] The messages framework | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/ref/contrib/messages/>. Acessado em: 25/11/2017.

[23] Using the Django authentication system | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/topics/auth/default/>. Acessado em: 25/11/2017.

[24] Sending email | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/topics/email/>. Acessado em: 25/11/2017.

[25] Internationalization and localization | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/topics/i18n/>. Acessado em: 25/11/2017.

[26] Django Debug Toolbar - Django Debug Toolbar 1.9 Documentation. <https://django-debug-toolbar.readthedocs.io/en/stable/>. Acessado em: 25/11/2017.

[27] select_related() | QuerySet API Reference | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/ref/models/querysets/#django.db.models.query.QuerySet.select_related>. Acessado em: 25/11/2017.

[28] prefetch_related() | QuerySet API Reference | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/ref/models/querysets/#prefetch-related>. Acessado em: 25/11/2017.

[29] Prefetch | QuerySet API Reference | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/ref/models/querysets/#django.db.models.Prefetch>. Acessado em: 25/11/2017.

[30] Working with forms | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/topics/forms/>. Acessado em: 25/11/2017.

[31] Deploying Django | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/howto/deployment/>. Acessado em: 25/11/2017.

[32] Testing in Django | Django Documentation | Django. <https://docs.djangoproject.com/en/1.11/topics/testing/>. Acessado em: 25/11/2017.