Universidade de São Paulo
Instituto de Astronomia, Geofísica e Ciências Atmosféricas
Departamento de Astronomia

Affonso Gino Amendola Neto

Banco de Dados *BeaconDB*

São Paulo
2021

Affonso Gino Amendola Neto

Banco de Dados *BeaconDB*

Trabalho de conclusão de curso apresentado ao Departamento de Astronomia do Instituto de Astronomia, Geofísica e Ciências Atmosféricas da Universidade de São Paulo como requisito parcial para a obtenção do título de Bacharel em Astronomia.

Área de Concentração: Astronomia
Orientador: Prof. Dr. Alex Cavaliéri Carciofi (IAG/USP)

São Paulo

2021

Ao meu Pai.

Agradecimentos

À minha mãe, que sempre me encorajou a dar o melhor de mim na faculdade.

Ao meu Orientador, Alex Carciofi.

Ao Dream Team, sempre presente.

Aos meus colegas do IAG.

Aos professores e professoras do IAG, em especial, Alessandro, Jane e Roberto, mas todos me ensinaram muito!

Esta tese/dissertação foi escrita em LaTeX com a classe IAGTESE, para teses e dissertações do IAG.

"*Be Excellent To Each Other!*"

Bill S. Preston, Esq., 2688

"*I reject your reality and substitute my own!*"

Adam Savage, originally from *The Dungeonmaster* (1984)

Resumo

Bancos de Dados de imagens e dados astronômicos se provaram essenciais no repertório de ferramentas de um astrônomo moderno. Bancos como o BeSS (Be Star Spectra Database) são usados diariamente por pesquisadores para armazenar novos dados aguardando análise, ou aplicar novas técnicas e conhecimentos em dados antigos.

Dados de estrelas Be no momento se encontram separados em alguns bancos de dados, o BeSS, o do BeSOS (Be Stars Observation Survey), o do ESO (European Southern Observatory), etc. Nosso objetivo é construir um novo banco de dados, tentando unificar os dados de todos esses bancos em um só lugar.

O escopo desta monografia de Trabalho de Graduação é estudar possíveis soluções para o desenvolvimento deste novo banco de dados usando softwares existentes, propor formas de integrá-los e estabelecer suas funcionalidades básicas mais importantes. Este estudo e o texto que o acompanha servirá de base para a implementação futura da base de dados beaconDB.

Abstract

Astronomical data and image databases have proved themselves essential in the repertoire of tools of a modern astronomer. Databases like BeSS (Be Star Spectra Database) are used daily by researchers to store new data awaiting analysis, or to apply new techniques and knowledge on older data.

Data from Be Stars are found separated in a few Databases, BeSS, BeSOS (Be Stars Observation Survey), ESO (European Southern Observatory), and etc. Our objective is to create a new database, unifying this data in a single place.

This work's scope is to study possible solutions for the development of this new Database using existing software, proposing ways of integrating them and establishing their most basic functionalities. This study and text will serve as the basis for the future implementation of the BeaconDB Database.

Lista de Figuras

Sumário

Capítulo 1

Introdução

Estrelas Be são estrelas do tipo B que mostram linhas de emissão de hidrogênio, muitas vezes com variações no espectro, de forma que o fenômeno de emissão pode ser transitório (Rivinius et al. 2013). A variabilidade de estrelas Be deu luz a alguns projetos de banco de dados criados com o objetivo de armazenar a grande quantidade de dados oriundas de observações desse tipo de estrela, já que múltiplas observações são necessárias, e toda observação pode ser valiosa num contexto futuro. Dois projetos se destacam: o BeSS (Be Star Survey Database, Neiner et al. 2011) e BeSOS (Be Star Observation Survey, Vanzi et al. 2012).

O banco de dados do projeto BeSS é o maior banco de dados desse tipo de estrela atualmente em existência. Possuindo mais de 250 mil entradas de espectros. Ele é mantido pelo LESIA (Laboratoire d'Études Spatiales et d'Instrumentation en Astrophysique) do Observatório de Paris. Boa parte dos dados disponíveis foram gerados por astrônomos amadores, cuja contribuição à pesquisa de estrelas Be e outros alvos brilhantes tem se tornado qualitativamente melhor e quantitativamente maior nos últimos anos. A disponibilidade de tempo dos amadores é algo importante para observações de estrelas Be, já que são estrelas variáveis, e eventos interessantes podem acontecer a qualquer momento.

Por outro lado, o projeto BeSOS (Vanzi et al. 2012) armazena dados de somente o espectrógrafo PUCHEROS (Pontificia Universidad Catolica High Echelle Resolution Optical Spectrograph, Vanzi et al. 2012), mas possui excelentes ferramentas de visualização de dados, como plotagem interativa e extração de quantidades relevantes do espectro, tais como a largura equivalente.

O objetivo deste trabalho é a criação de um banco de dados moderno, combinando o volume de dados presente no BeSS com as ferramentas de visualização, análise e cross-

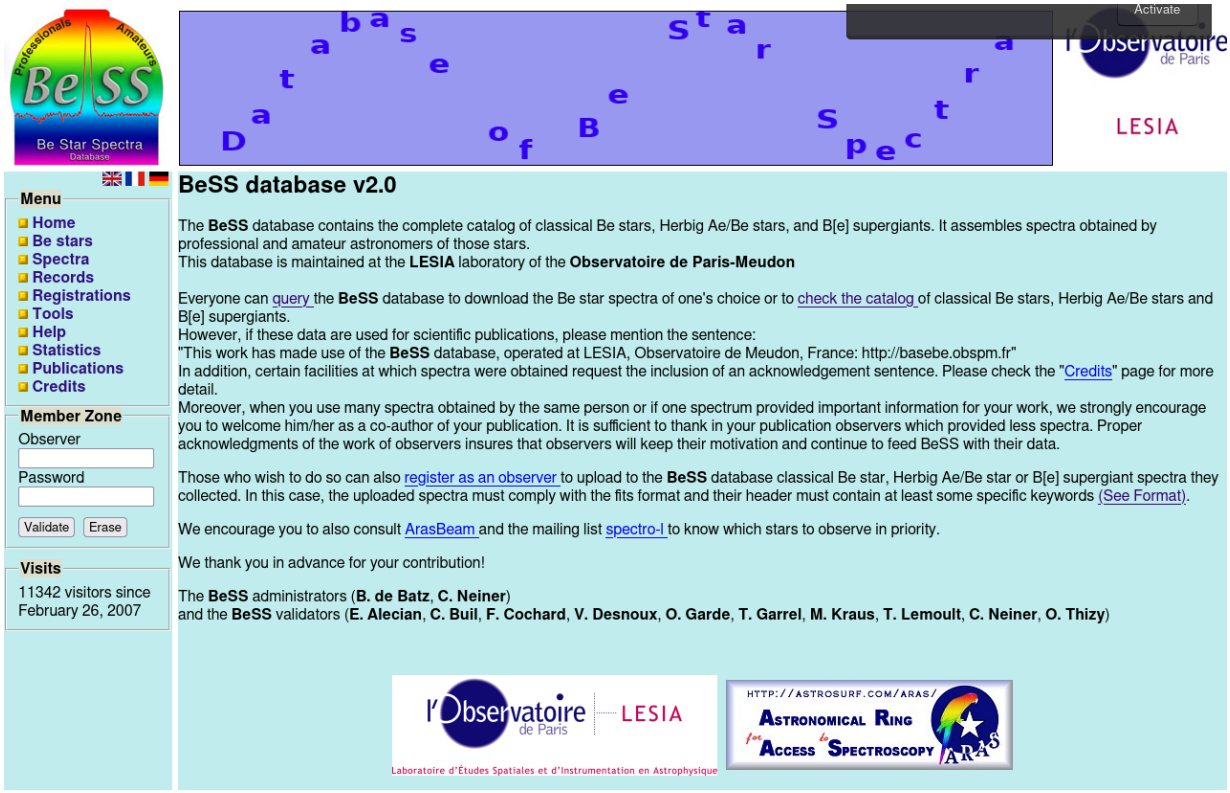

Figura 1.1: Captura de tela da interface do web-site do projeto BeSS.

matching do BeSOS. Como a concretização de um projeto deste tipo é um projeto de grande porte, além do escopo do um Trabalho de Graduação, nesta monografia desenvolvemos um plano para esta base de dados, explicando e definindo todos os passos necessários para a criação do software necessário similar.

A grande maioria da funcionalidade requerida por um software desse tipo já existe livremente disponível na internet, na forma de bibliotecas FOSS (*Free and Open-Source Software*, software livre e de código aberto), então a maior parte do trabalho necessário consiste em escrever um código que permita a interação harmoniosa entre as diferentes bibliotecas e providenciar uma interface de acesso ao usuário.

A monografia seguirá da seguinte forma. No Capítulo 2 serão expostos os conceitos básicos relevantes para a construção do software, no Capítulo 3 será definida a proposta para a estrutura do banco de dados e decisões relevantes ao bom funcionamento do software. Finalmente, no Capítulo 4 o protótipo e as partes do software que já estão funcionando são descritas.

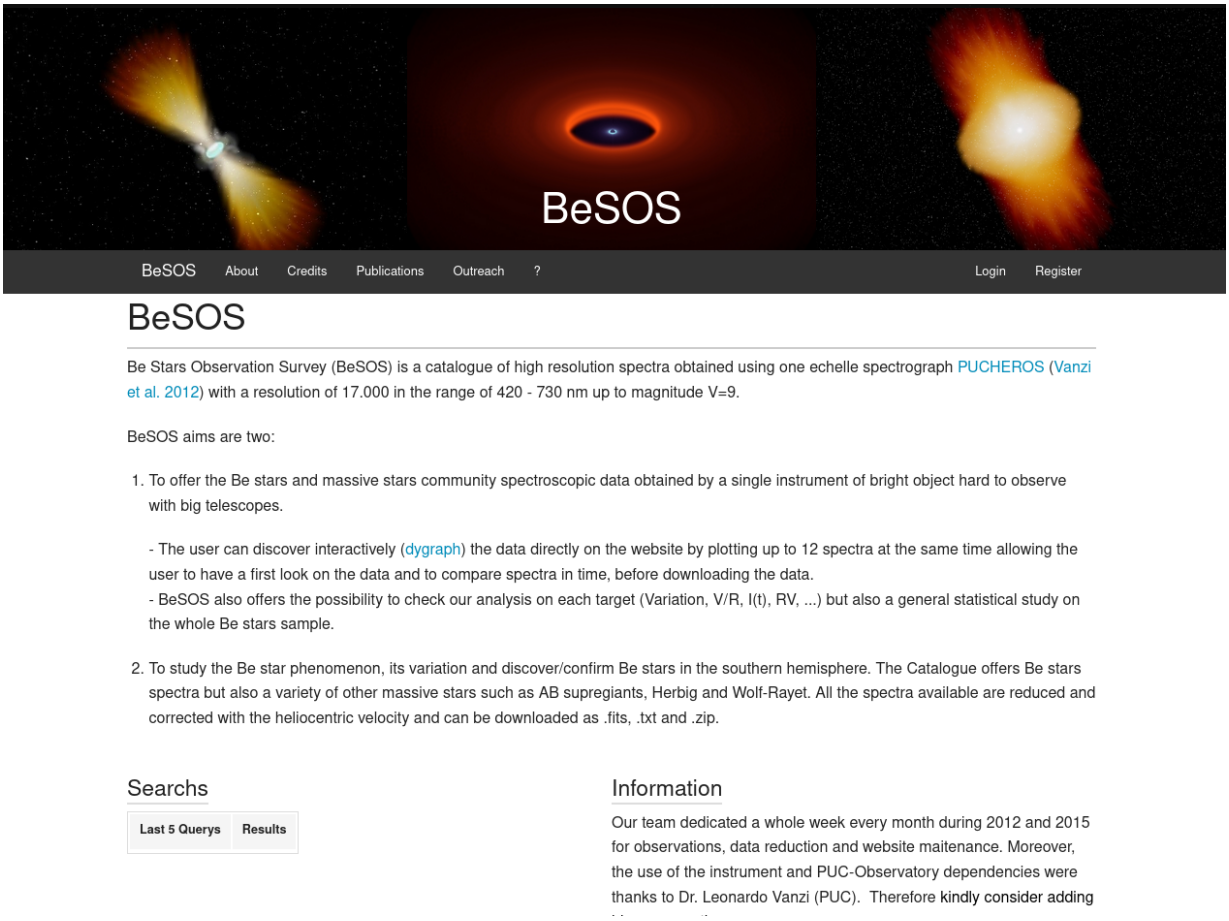

Figura 1.2: Captura de tela da interface do web-site do projeto BeSOS;

Capítulo 2

Conceitos Básicos

Quando falamos de "Software de Banco de Dados Astronômicos", geralmente nos referimos a uma combinação de vários tipos diferentes de programas, trabalhando em conjunto para nos trazer a funcionalidade desejada. Esse tipo de software é geralmente dividido em duas partes, chamadas de *Back-End*, e *Front-End*, a combinação das duas é também chamada de *Stack*, ou *Full-Stack* (Smith 2012).

O *Back-End* geralmente consiste na estrutura que certo software precisa ter para atingir seus objetivos, como Gerenciadores de Bancos de Dados (MySQL[1], MongoDB[2], PostgreSQL[3]), servidores web (Apache[4], Flask[5]) e conteinerização (Docker[6], Kubernetes[7]). O Front-End consiste na parte do software que é visível ao usuário, como páginas web (HTML [8], JavaScript [9] e CSS[10] e aplicativos nativos de certo sistema operacional (Windows, Linux, MacOS).

O *Front-End* se comunica com o *Back-End* através de protocolos de comunicação, o mais usado sendo o HTTP, mas outras formas também existem, como o SQL[11]. Esta pode ser usada para se comunicar diretamente com um banco de dados, sem a necessidade de passar por algum outro servidor (Smith 2012).

O *Stack* que propomos deverá ser executável em qualquer servidor moderno usando o

[1] https://www.mysql.com/
[2] https://www.mongodb.com/
[3] https://www.postgresql.org//
[4] https://httpd.apache.org/
[5] https://flask.palletsprojects.com/en/2.0.x/
[6] https://www.docker.com/
[7] https://kubernetes.io/
[8] Hyper-Text Markup Language, https://html.spec.whatwg.org/multipage/
[9] https://tc39.es/ecma262/
[10] Cascading Style Sheets, https://www.w3.org/Style/CSS/
[11] Structured Query Language

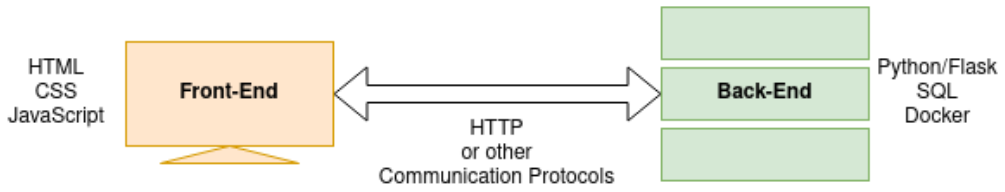

Figura 2.1: Ilustração do conceito de Front-End e Back-End, e a comunicação entre eles (fonte: Autoria Própria).

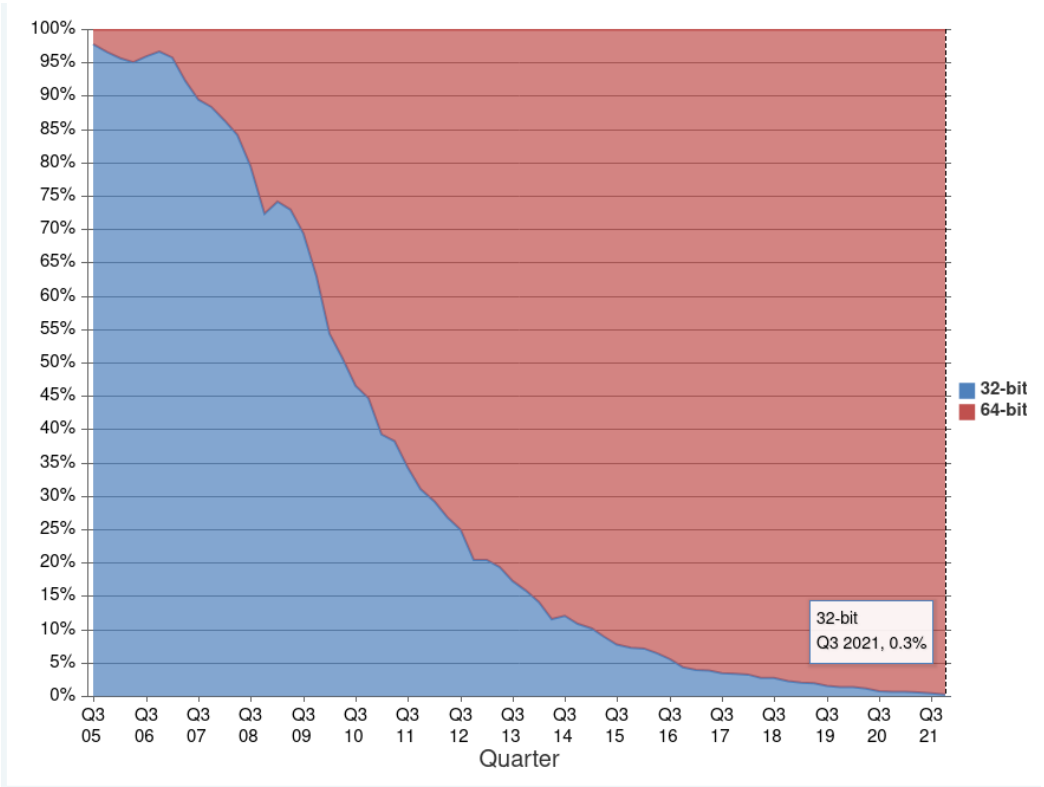

Figura 2.2: Gráfico de uso histórico de arquiteturas 64-bits e 32-bits, em específico do uso de Sistemas Operacionais Windows nessas arquiteturas (fonte: https://www.pcbenchmarks.net/os-marketshare.html).

sistema operacional Linux ou derivados, com o requisito de ser construído para arquiteturas modernas 64-bits, já que arquiteturas 32-bits já começaram a ser classificadas como obsoletas (Figura 2.2).

2.1 Escolha de Software

A escolha correta de software pré-existente é essencial para um bom processo de desenvolvimento. Primeiramente, como a comunidade astronômica vem adotando *Python* como sua linguagem de programação favorita (Momcheva & Tollerud 2015), foi determinado que soluções usando *Python* seriam priorizadas.

O *Python* é uma das linguagens que mais crescem no meio astronômico e científico; é uma linguagem de programação de alto nível, interpretada e fracamente tipada (Van Rossum & Drake 2009). Ou seja, ela está "longe" do hardware; não precisa de um compilador, somente de um interpretador compatível com o sistema operacional de determinado computador onde o código deseja ser executado. Além disso, variáveis não precisam ser declaradas como sendo algum tipo antes de serem usadas, permitindo a fácil mudança de seus valores por valores de tipos diferentes (Van Rossum & Drake 2009).

Essas características, juntamente com a simplicidade de sua escrita e facilidade de adição de novas bibliotecas, podem estar ligadas ao crescimento vertiginoso do *Python* no meio científico (Momcheva & Tollerud 2015).

A simplicidade na escrita do código *Python* permite rápida prototipagem, especialmente em computadores a base de **nix* (*Unix, Linux*, etc.) e possuí várias bibliotecas extremamente robustas e rápidas, sendo elas compiladas geralmente em linguagens mais rápidas, de baixo nível como *C* ou *Rust*, usando a capacidade de *Python* de introduzir *bindings*[12] a código compilado de *C* ou outras linguagens, permitindo que funções dessas linguagens possam ser chamadas pelo interpretador (Van Rossum & Drake 2009).

O desenvolvimento Web em *Python* pode ser feito usando uma *"Micro-Framework"* chamada *Flask*, que adiciona novos recursos à linguagem que permitem "anotar" certas partes do código para serem iniciadas quando algum pedido *HTTP* chega no *Flask*, basicamente permitindo o código atender a estes pedidos, transformando o interpretador *Python* num servidor web completo. Ela também possuí recursos para comunicação com gerenciadores de Bancos de Dados como *MySQL*, *MongoDB* e outros, algo que também será necessário para este projeto.

Já que *Flask* usa a linguagem *Python*, todas as bibliotecas que estão disponíveis para

[12] Código específico referenciando chamadas em bibliotecas nativas de uma maneira que possam ser usadas em linguagens interpretadas como Python.

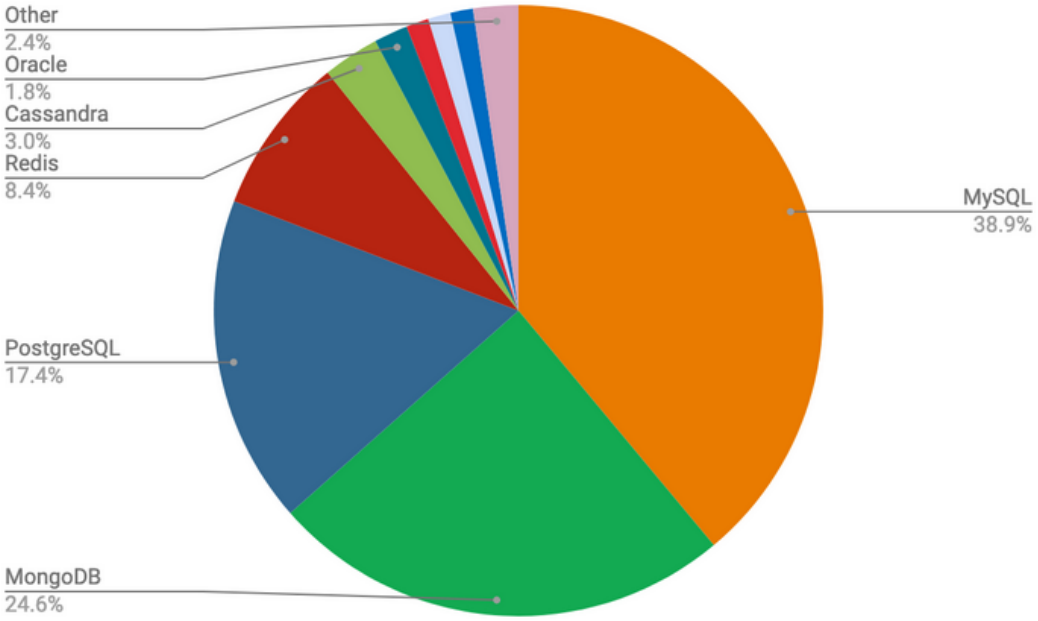

Figura 2.3: Comparação do uso de sistemas de banco de dados, auto-reportado por desenvolvedores que atenderam a conferência DeveloperWeek 2019 (Fonte: https://scalegrid.io/blog/2019-database-trends-sql-vs-nosql-top-databases-single-vs-multiple_database-use/)

Python também estão disponíveis e acessíveis por *Flask*, pois sua implementação é basicamente a de uma biblioteca padrão, como todas as outras [5]. Portanto, a *"Micro-Framework" Flask* foi escolhida para servir de base para a *Back-End*.

Uma escolha essencial a ser feita é a de qual sistema de banco de dados seria o mais compatível com as intenções do *Stack* proposto. O Sistema de Banco de Dados Relacional *Open-Source MySQL* é um desses tipos de sistema e o grande número de projetos usando tanto ele quanto a linguagem de consulta *SQL* indica uma grande probabilidade dele ser bem mantido por muitos anos por vir[13].

Outros sistemas existem, como *MongoDB* e *PostgreSQL*, mas a grande proporção de uso de *MySQL*, a existência de boa documentação e a compatibilidade quase imediata com *Flask*, faz dela a melhor opção.

Outro requisito importante do projeto é o de fácil migração entre servidores, já que isso permite que software seja desenvolvido e testado em um computador, sabendo que ele operará de forma muito similar no servidor intencionado ao projeto. O software *Docker*

[13] https://www.mysql.com/

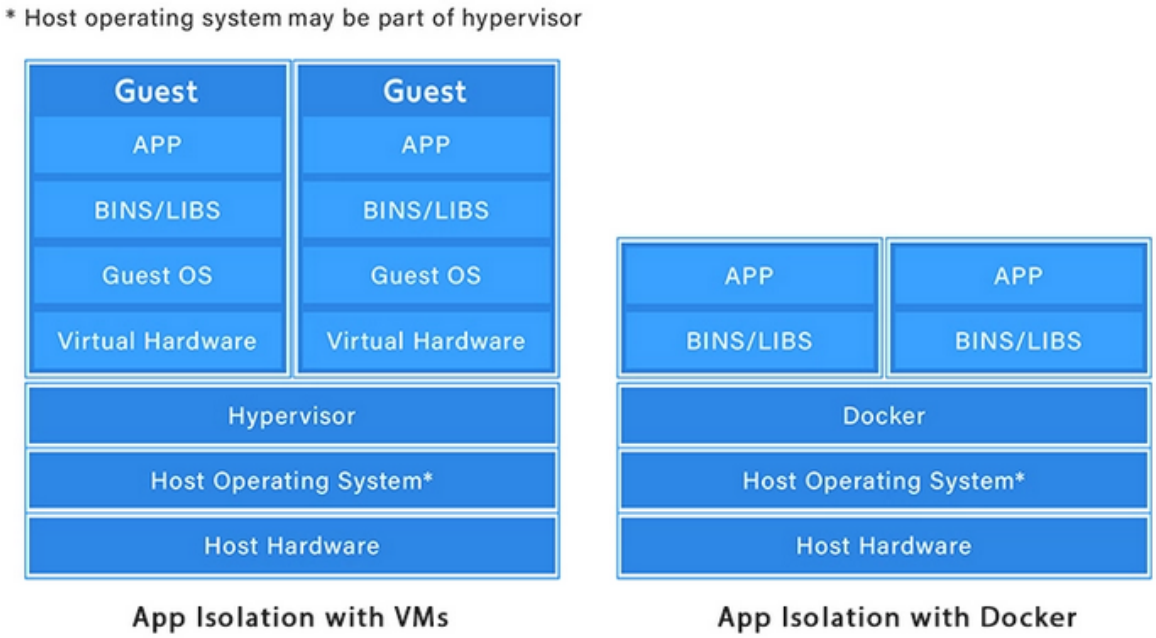

Figura 2.4: Comparação do isolamento de ambiente de certo aplicativo em Máquinas Virtuais Tradicionais, e usando o software Docker. Fonte: (https://www.codeclouds.com/blog/an-intro-to-docker-basic-image-setup-and-deploying-your-first-container/)

permite a conteinerização de *Stacks* de servidores e diferentes formas de outros *softwares*, tornando possível que todo o ambiente e infraestrutura que esses programas usam seja uniformemente acessível a partir de várias configurações diferentes, contanto que o software *Docker* também esteja presente. Isso é possível através da criação de imagens *Docker* que são usadas com sistemas de virtualização existentes no *software* [6].

O *Docker* é uma solução alternativa a Máquinas Virtuais pois opera dentro de um sistema operacional, e não o substituindo como em máquinas virtuais (Figura 2.4)., permitindo melhor performance, com o custo de uma maior dificuldade de implementação em sistemas menos usados. Entretando, como pretende-se usar um servidor *Linux*, isso não será um problema [6].

Infelizmente ainda existem muitas restrições no que é possível rodar num navegador (*browser* – *Internet Explorer*, *Safari*, etc.); basicamente todo o *Front-End* está limitado a usar uma combinação de *HTML*, *JavaScript* e *CSS* (Schaefer 2012). O problema surge ao notar que o *HTML* é uma linguagem totalmente estática, com quase toda a parte dinâmica do *website* sendo adições posteriores ao padrão *HTML*, primeiramente na forma de *JavaScript* (Schaefer 2012), e posteriormente na forma de *CSS*.

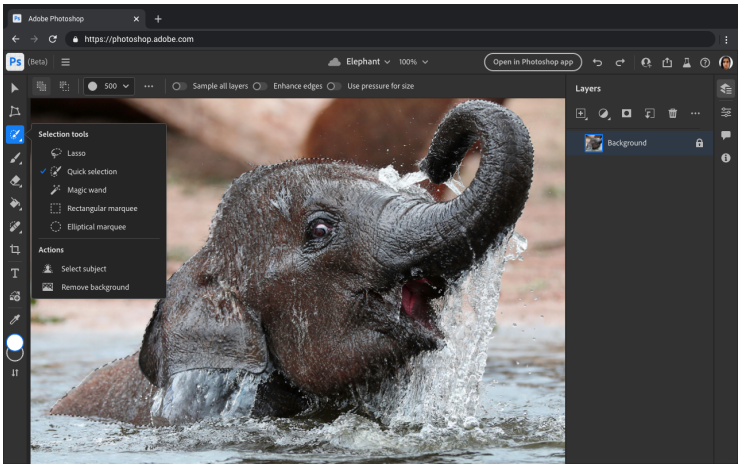

Figura 2.5: Exemplo do tipo de funcionalidade que podemos esperar de WebAssembly no futuro próximo. (Fonte: Adobe)

Existe um padrão de código aberto (*open-source*) moderno chamado *WebAssembly* (*wasm*) que visa disponibilizar uma interface mais baixo-nível em qualquer navegador, permitindo que basicamente qualquer programa escrito em qualquer linguagem possa ser compilado para WebAssembly e funcionar nativamente em um navegador, como se ele fosse um sistema operacional[14]. Ele foi usado com grande sucesso nos projetos *WebArchive*[15], e mais recentemente em um *port*[16] das ferramentas Photoshop da Adobe para navegadores, como pode ser visto na Figura 2.5, que demonstra várias funcionalidades antes restritas a *software* operando nativamente, agora sendo possíveis em um *website*, acessível por um navegador comum.

No entanto, o projeto *WebAssembly* ainda está em sua infância; apesar de promissor, ainda existem várias questões e dificuldades em sua utilização, como variações em implementações de motores de *wasm* e o fato do padrão ainda estar na sua primeira versão, 1.0. com uma versão 1.1 em estágio de rascunho. Talvez no futuro possamos rever como confeccionamos páginas web, e iremos para um futuro onde podemos usar qualquer linguagem de programação na sua elaboração [14].

Felizmente não precisaremos de nenhum recurso exclusivo de padrões como *WebAssembly*, com a *Front-End* podendo ser escrita na forma padrão da internet atual, usando *HTML*, *JavaScript*, e *CSS* , contornando os problemas que surgem ao tomar essa decisão,

[14] https://webassembly.org/

[15] http://web.archive.org/

[16] Versão de um software específico para um novo sistema operacional ou uma nova arquitetura.

como a falta de flexibilidade e dinamismo da página usando recursos modernos de *CSS* e *JavaScript*.

Capítulo 3

Análise e Desenvolvimento

Neste capítulo propomos o desenho principal de um software que supre os requisitos do projeto, explicando o porquê de cada decisão tomada durante o desenvolvimento, de forma que os *softwares* utilizados consigam operar de forma apropriada entre si, nos dando, ao final, um *software* robusto, seguro, de simples utilização e que tem todas as ferramentas desejadas.

3.1 Arquitetura

O *Stack* precisa ser totalmente contido em uma imagem *docker* para fácil transporte entre diferentes sistemas permitindo uma alta taxa de iteratividade no desenvolvimento[6]. A maneira que isso pode ser feito é determinada majoritariamente pelo conteúdo de um arquivo chamado *Dockerfile* e usando o aplicativo *Docker-Compose* [6] para criar duas imagens, uma de um sistema completo para o *software* de Banco de Dados, e uma para o Servidor *Web*.

As imagens são separadas e se comunicam somente através de protocolos de comunicação específicos do sistema de banco de dados (Figura 3.1), permitindo uma camada adicional de segurança: acesso não autorizado ao sistema web não implica imediatamente no acesso ao banco de dados [6].

É necessário garantir que as imagens possuam todos os *softwares* e bibliotecas necessárias para a operação do programa. Isso é facilmente resolvido em sistemas *Docker* combinados com *Python*, usando um arquivo de texto chamado *requirements.txt*, que possui uma lista das dependências da imagem e de suas versões. O programa *Docker*, ao "construir"as imagens, pode por exemplo, enviar essa lista de dependências ao software

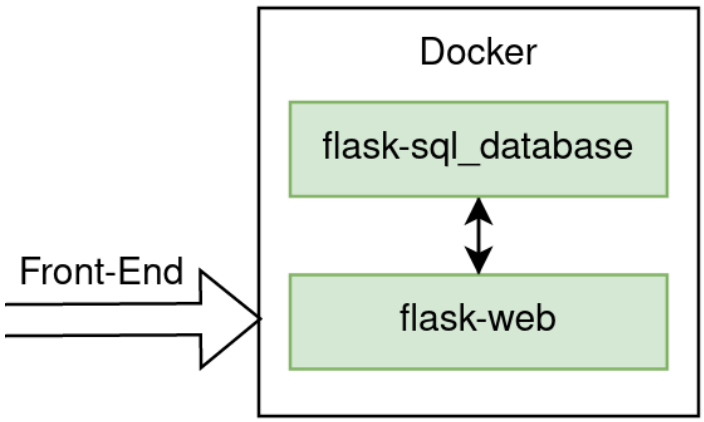

Figura 3.1: Arquitetura dos contêineres *Docker*, mostrando a separação de comunicação entre mundo externo (*Front-End*) e as duas imagens, em verde (fonte: Autoria Própria).

pip, um Gerenciador de Pacotes de *Python*, que por sua vez instala as dependências na imagem [1].

Essa lista pode ser facilmente alterada, e a imagem regenerada com a lista nova, permitindo grande oportunidade de prototipagem, sendo possível testar outras bibliotecas ou outras configurações [1], e, em combinação com softwares de versionamento como Git, reverter para pontos no passado se as mudanças não funcionaram como intencionado (Chacon & Straub 2014).

O servidor precisa de uma estrutura de arquivos específica para funcionar, são necessárias diferentes pastas em diferentes lugares para interagir com o programa (Figura 3.3), como por exemplo uma pasta onde todos os arquivos enviados pelos usuários serão armazenadas.

3.2 Programação do Back-End

O software proposto consiste primariamente de um código *Python*, usando a *Micro-Framework Flask* para permitir a interação entre as diferentes partes do software. í*Flask* permite a definição de funções específicas, marcadas com *"Decorators"* permitindo a execução de uma parte especifica de código *Python* de acordo com algum tipo de *request HTTP* chegando no servidor [5].

[1] https://docs.docker.com/

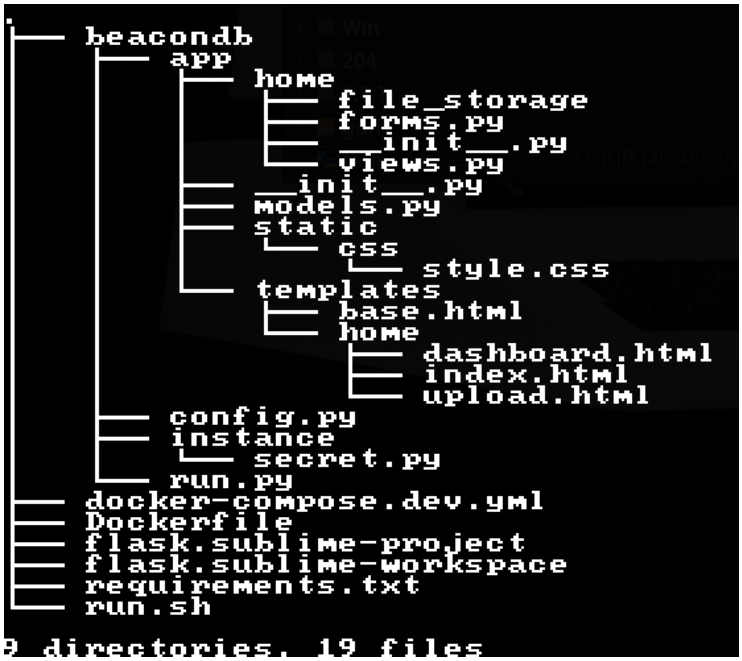

Figura 3.2: Estrutura de Arquivos do Servidor, algumas pastas precisam estar em locais relativos definidos com nomes específicos para o correto funcionamento do *Flask*.

```
@home.route('/')
def homepage():
    return render_template("home/index.html", title="BeaconDB")
```

Figura 3.3: Exemplo de um *decorator* do *Flask*. Com esse simples código, responde-se a um pedido HTTP ao endereço base do domínio (Ex. http://beacondb.com/) com uma página *HTML index* localizada na pasta home do servidor (fonte: Autoria Própria).

Isso nos dá a possibilidade de receber um pedido HTTP engatilhado por um usuário numa página web ou app, e dependendo do conteúdo do pedido, podemos agir sobre o sofware MySQL, pedindo informação sobre algum arquivo especifico, com o qual podemos executar qualquer código[5], seja ele em *Python* ou em outra linguagem de programação (usando *bindings* de *Python*) em qualquer thread do computador abrigando o servidor, e, se for parte do pedido, enviar os arquivos ou informações requisitadas.

Os pedidos, portanto, precisam ser bem definidos. Já que o *Front-End* e o *Back-End* são tecnicamente independentes, é necessário que ambos estejam falando a mesma língua, o *Front-End* não pode enviar um pedido de uma maneira que o *Back-End* não saiba tratar, e o *Back-End* não pode responder um pedido de uma maneira inesperada.

O principal pedido solicitado pelo *Front-End* para esse tipo de página, será um pedido customizado com algum tipo de código *SQL*, pois esta é a maneira mais simples de extrair

dados do banco. Este código precisa ser verificado para brechas de segurança, e não pode ser encaminhado diretamente ao banco de dados, já que código *SQL* puro pode conter instruções danosas, como tentar obter informações sem acesso permitido, como, por exemplo, uma tabela de *hashes*[2] de senhas, que não é planejada no momento, mas poderá ser adicionada no futuro caso um sistema de *Login* seja implementado para identificar usuários que podem fazer *upload* de arquivos.

Um ataque muito comum em meios de segurança de dados é o de *SQL-Injection* onde um *hacker* mal-intencionado envia uma sequência de código *SQL* de uma maneira especifica tentando burlar medidas de segurança mal implementadas, com consequências catastróficas, como vazamento de dados ou destruição de toda a tabela [3].

Outra seção do código vulnerável a ataques é o de *upload* de arquivos, se aproveitando de como sistemas operacionais lidam com nomes de arquivos. Isso, no entanto, pode ser facilmente corrigido usando algumas funções pré-implementadas pelo *Flask*, como $secure\_filename()$, que automaticamente remove caracteres perigosos como "." e "..", e transforma nomes de arquivos referenciais para absolutos [4].

3.3 Tratamento De Dados

Os bancos de dados existentes, mencionados na Introdução, possuem certas características relevantes ao desenvolvimento do nosso *Stack*, em especial na forma como os dados são guardados. O banco de dados BeSS, especificamente, disponibiliza um documento que descreve entradas de cabeçalho que precisam existir em arquivos *.FITS* enviados ao projeto.

O software proposto também precisará receber dados de qualquer fonte, seja ela amadora ou profissional, portanto é interessante manter os dados com o mesmo nível de qualidade dos dados do BeSS, talvez até usando o mesmo formato, já que o projeto BeSOS, apesar de ter a mesma uniformidade de qualidade de dados, não tem seus padrões definidos em nenhum lugar pois, como seus dados vieram todos da mesma fonte, eles têm o mesmo conjunto de cabeçalhos.

[2] Maneira como senhas são guardadas de forma segura em bancos de dados, um hash é uma forma de encriptação.

[3] https://owasp.org/www-community/attacks/SQL$_{I}njection$

[4] https://owasp.org/www-community/vulnerabilities/Unrestricted$_{F}ile_{U}pload$

Cada um dos projetos usa uma nomenclatura diferente para os cabeçalhos do arquivo FITS, como o projeto BeSS que usa BSS_NAME para o cabeçalho do nome do alvo [5], diferentemente do usual $OBJNAME$ ou $NAME$, usados pelo projeto BeSOS.

Existem diferentes soluções para permitir o recebimento desses dados. Uma delas seria converter todos os arquivos que entrariam ao banco de dados para um formato único; essa provavelmente foi a solução adotada pelo projeto BeSS, como sugerem os elementos customizados em seu cabeçalho.

Outra solução é escrever a seção de código que lida com a extração de informação de uma maneira que a deixa capaz de lidar com esses elementos customizados no cabeçalho, e providenciar ao usuário uma maneira de informar o programa que o dado que ele pretende enviar ao servidor é um dado originário de um certo projeto, e portanto seu arquivo .FITS tem headers específicos que possibilitariam a coleta de dados.

Podemos usar as funções da biblioteca Astropy para abrir os arquivos .FITS e extrair informações tanto da seção de dados do arquivo, quanto da seção de header (Astropy Collaboration et al. 2018).

Já que estamos lidando com no mínimo 250 mil entradas (Neiner et al. 2011) e não podemos passar pelo processo de abrir todos os arquivos quando, por exemplo, um usuário requisitar todas as estrelas Be em volta de um certo ponto no céu, é necessário obter certas informações dos arquivos em momento de upload, e marcar as colunas de dados onde essas informações são salvas, como indexáveis, para "ajudar" o gerenciador de banco de dados a encontrar esses dados de maneira mais rápida.

Usuários deverão ter a opção de enviar seus próprios dados ao servidor, passando pela moderação de alguém versado na ciência por trás das observações. Isso requer a existência de uma área no *web-site* onde usuários preenchem um formulário com informações relevantes e possam iniciar a operação de *upload* de um arquivo do seu próprio computador, usando os recursos de *drag-and-drop* de navegadores e sistemas operacionais modernos.

3.4 Programação do Front-End

É proposta a confecção de um *web-site*, acessível pelos navegadores modernos (*Chromium*, *Firefox*, *Safari* e *Edge*), onde o usuário poderá requisitar os espectros que desejar,

[5] http://basebe.obspm.fr/basebe/

ou fazer upload de suas próprias observações.

O website proposto será servido pelo software *Flask*, que providencia uma interface entre *Python* e a combinação de *HTML*, *JavaScript* e *CSS* que compõe o ecossistema de *webpages*.

Ele precisará ter uma seção onde o usuário tem a opção de inserir um pequeno trecho de código *SQL* (Ou possivelmente *ADQL*[6], extensão de *SQL* específica para Astronomia) e requisitar informações diretamente do banco de dados.

Essa seção é crítica em termos de segurança de dados; providenciar ao usuário acesso direto a um banco de dados apresenta um sério risco de vazamento ou destruição de dados, portanto qualquer *Query* precisa ser feita de forma segura. Esta checagem é realizada na *Back-End*, uma vez que um pedido é enviado [3].

Juntamente com a capacidade de enviar *Queries*, é necessário uma página onde o usuário consiga visualizar os resultados da sua *Query*, na forma de uma lista com os objetos que satisfazem as limitações impostas.

Também propomos uma maneira em que o usuário possa manipular e visualizar dinamicamente os dados obtidos com suas *Queries*, usando bibliotecas que façam uso de recursos modernos de navegadores como a biblioteca *D3.js* que faz uso de *JavaScript* e *CSS* para renderizar tabelas, gráficos e outras formas de visualização de dados, dinamicamente num navegador comum[7].

Essas possíveis manipulações não podem alterar o conteúdo de arquivos guardados no banco de dados, mas é possível criar uma cópia deles em memória que possa ser manipulada pelo usuário em tempo real.

Devido às limitações com o ecossistema web e sua dependência a *HTML* (Schaefer 2012), o código de um sistema de manipulação dinâmica de espectros no navegador é extremamente complexo, sendo mais possível usando padrões como *WebAssembly*, previamente mencionado, mas podemos pré-definir certas ações específicas que podem ser realizadas com os dados requeridos tanto no momento de *Upload*, pré-calculando certos valores que no futuro podem ser requisitados, quanto no momento de *Download*, alterando a versão do arquivo residente na memória *RAM* e retornando esta versão modificada ao invés da

[6] Astronomical Data Query Language, https://www.ivoa.net/documents/ADQL/20180112/PR-ADQL-2.1-20180112.html

[7] https://d3js.org/

original.

Isso é possível devido à baixa quantidade de acessos deste tipo de software; neste caso, seria possível marcar certas entradas no banco de dados como requerendo pré-cálculo ou análise automatizada, e realizar essas análises automaticamente quando o servidor não estiver sob carga muito grande.

Capítulo 4

Protótipo

Usando as propostas do Capítulo 3, foi começado o desenvolvimento de uma prova de conceito. A Stack proposta foi construída e está operando, ela tem várias funcionalidades esperadas em variados estados de desenvolvimento, ainda não sendo acessível ao público.

O protótipo mostra o perfeito funcionamento do Back-End do servidor, sendo ele implementado usando Python, Flask e MySQL. Ele está usando um modelo simplificado de pedidos de arquivos, por enquanto somente sendo possível fazer o download de um arquivo desejado por vez.

No entanto, o protótipo mostra o funcionamento da parte mais importante, a interoperabilidade da página web com código Python. Já que podemos transformar pedidos em chamadas de funções de Python, podemos executar qualquer análise desejada na biblioteca Astropy, "emprestando" o poder de processamento do servidor ao cientista. No momento o software só usa essa habilidade para extrair informações de arquivos .FITS enviados ao servidor na página de upload, mas junto dessas funções do astropy existem outras, com funcionalidades muito mais avançadas.

No entanto é necessário algum cuidado usando essa funcionalidade, pois ao operar um web-site há que se vigiar a quantidade de processamento disponível. A habilidade de "emprestar" o poder de processamento do servidor para fazer alguma análise pode acabar sobrecarregando o servidor, travando outras facetas de sua operação.

Existem maneiras de se defender contra esse tipo de problema. Podemos, por exemplo, isolar algumas das análises mais comuns e pré-calculá-las para dados novos, em horários oportunos ao servidor, como no meio da noite quando menos pessoas estão trabalhando com ele.

Podemos, também, simplesmente impedir o usuário de fazer alguma análise, avisando

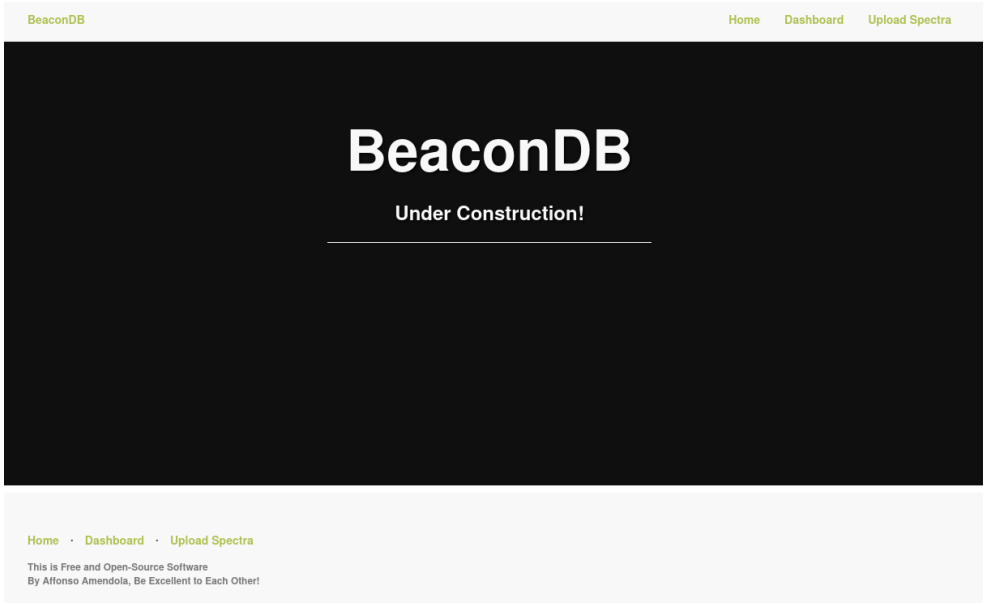

Figura 4.1: Captura de tela do website protótipo.

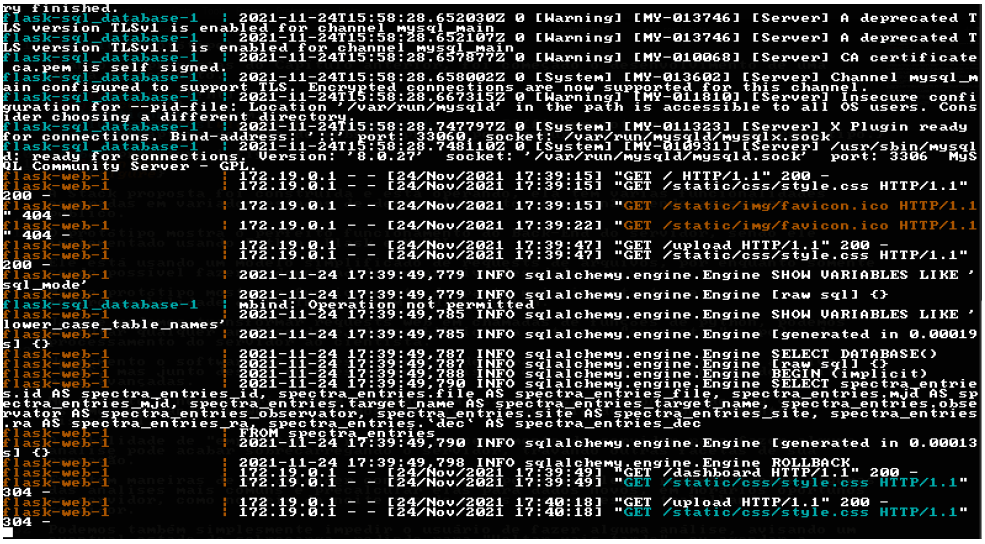

Figura 4.2: Monitoramento da atividade do servidor em execução no host local. É possível notar ambos os contêineres, em laranja e em ciano.

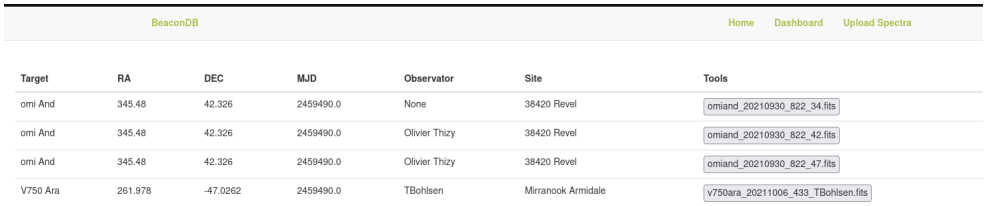

Figura 4.3: Página de download do protótipo.

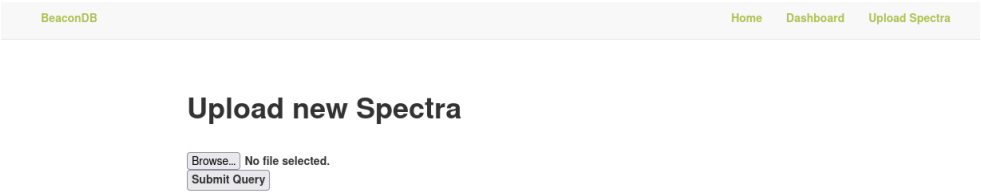

Figura 4.4: Página de upload do protótipo.

um eventual estado de sobrecarga, pedindo para "Voltar mais tarde", ou agendar a análise para um momento oportuno.

Capítulo 5

Conclusões

O objetivo deste projeto foi o desenvolvimento de um plano conceitual de um software de base de dados, específico para estrelas Be, com poder de análise de dados. Foi desenvolvida uma estrutura básica que providencia todas as funções necessárias para este objetivo, com o software em questão atualmente em desenvolvimento e um protótipo que demonstra o funcionamento dessa estrutura.

O protótipo demonstra um grande potencial de sucesso. A fácil interface entre o Front-End e código Python executando no Back-End permite uma grande variedade de ações a serem realizadas em cima de dados. Todas as funções de Astropy estão acessíveis, algumas delas já sendo usadas para ajudar a popular o banco de dados, obtendo, por exemplo, informações relevantes de novos arquivos .FITS.

A capacidade do Flask de atender pedidos http com código HTML, JavaScript e CSS, permite o uso de bibliotecas de análise e visualização de dados escritas usando essas linguagens, como a biblioteca D3.js.

O fator limitante para os tipos de análise que podem ser feitas diretamente do navegador é a complexidade inerente em escrever uma interface gráfica responsiva usando somente HTML, JS e CSS (Schaefer 2012), no entanto isso não significa que uma interface gráfica responsiva não possa ser feita, somente que a complexidade do código pode aumentar drasticamente se cuidados não forem tomados, elevando também os custos de manutenção dessa parte do código.

O desenvolvimento do protótipo em um software completo e usável continua, com uma previsão de conclusão até o começo de 2022.

Referências Bibliográficas

Astropy Collaboration et al., 2018, AJ, 156, 123

Chacon S., Straub B., 2014, Pro git. Apress

Momcheva I., Tollerud E., 2015, arXiv e-prints, p. arXiv:1507.03989

Neiner C., de Batz B., Cochard F., Floquet M., Mekkas A., Desnoux V., 2011, AJ, 142, 149

Rivinius T., Carciofi A. C., Martayan C., 2013, A&A Review, 21

Schaefer R., 2012, SIGSOFT Softw. Eng. Notes, 37, 7–8

Smith P., 2012, Professional Website Performance: Optimizing the Front-End and Back-End. PROGRAMMER TO PROGRAMMER, Wiley, https://books.google.com.br/books?id=eYl2jmcRGSMC

Van Rossum G., Drake F. L., 2009, Python 3 Reference Manual. CreateSpace, Scotts Valley, CA

Vanzi L., et al., 2012, MNRAS, 424, 2770