UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

THIAGO FIGUEIREDO LOPES DE CASTRO

# UM REPOSITÓRIO DE DADOS PERSONALIZADO EM *DOCKER SWARM* PARA O PROJETO PRESENÇAS

RIO DE JANEIRO
2025

THIAGO FIGUEIREDO LOPES DE CASTRO

# UM REPOSITÓRIO DE DADOS PERSONALIZADO EM *DOCKER SWARM* PARA O PROJETO PRESENÇAS

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Profa. Maria Luiza Machado Campos

RIO DE JANEIRO

2025

CIP - Catalogação na Publicação

C355r

Castro, Thiago Figueiredo Lopes de
Um repositório de dados personalizado em Docker Swarm para o Projeto Presenças / Thiago Figueiredo Lopes de Castro. -- Rio de Janeiro, 2025.
114 f.

Orientadora: Maria Luiza Machado Campos.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2025.

1. Conjunto. 2. Docker. 3. Swarm. 4. Python. 5. Javascript. I. Campos, Maria Luiza Machado, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

THIAGO FIGUEIREDO LOPES DE CASTRO

UM REPOSITÓRIO DE DADOS PERSONALIZADO EM *DOCKER SWARM* PARA O PROJETO PRESENÇAS

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 09 de julho de 2025

BANCA EXAMINADORA:

gov.br Documento assinado digitalmente
MARIA LUIZA MACHADO CAMPOS
Data: 06/08/2025 21:10:24-0300
Verifique em https://validar.iti.gov.br

Prof.ª Dra. Maria Luiza Machado Campos
Orientadora
Universidade Federal do Rio de Janeiro

gov.br Documento assinado digitalmente
VIVIAN DOS SANTOS SILVA
Data: 22/07/2025 14:16:53-0300
Verifique em https://validar.iti.gov.br

Prof.ª Dra. Vivian dos Santos Silva
Universidade Federal do Rio de Janeiro

gov.br Documento assinado digitalmente
CAROLINA GIL MARCELINO
Data: 04/08/2025 10:47:19-0300
Verifique em https://validar.iti.gov.br

Prof.ª Dra. Carolina Gil Marcelino
Universidade Federal do Rio de Janeiro

Dedico este trabalho aos meus pais Claudio e Marcia, que trouxeram-me ao mundo para contribuir com vigor e empenho à sociedade, à minha avó Terezinha, que contribuiu integralmente no financiamento dos meus estudos, e ao meu irmão André e aos meus amigos que sempre estão presentes como alegradores e companheiros nos melhores e piores momentos.

## AGRADECIMENTOS

Agradeço à Maya Inbar e toda a equipe do Projeto Presenças do CAp UFRJ que permitiram o uso do conteúdo, confiaram no meu trabalho e cujo conteúdo é um dos pilares desta obra.

Agradeço também à minha orientadora Maria Luiza, que auxiliou-me com suas opiniões, conselhos e conhecimento de anos de profissão para que o melhor fosse obtido na versão final desta obra, e agradeço às professoras Carolina Gil Marcelino e Vivian dos Santos Silva por participarem da banca examinadora e contribuirem para o trabalho final e pelos elogios prestados.

Agradeço aos meus companheiros de equipe da Divisão de Serviços de Tecnologia de Informação e Comunicação e da Superintendência Geral de Tecnologia de Informação e Comunicação e à UFRJ por disponibilizarem e auxiliarem-me com o espaço para construção dos servidores usados neste trabalho.

*"Sometimes I feel like giving up, but I just can't. It isn't in my blood."*

**(*In My Blood*, Shawn Mendes)**

## RESUMO

Com a digitalização da informação, é de suma importância que ela seja acessível, organizada e reutilizável. A busca deve ser facilitada tanto para um usuário comum quanto para um sistema automatizado. Dessa forma, ferramentas devem ter interfaces úteis e simples e componentes estruturais seguindo o mesmo princípio, mas ao mesmo tempo eficientes e capazes de lidar com quaisquer complexidades. Se necessário, a construção de outras deve ser feita para complementar funcionalidades daquelas já existentes e construídas a partir do esforço comunitário no contexto de *software* abertos e livres. A disponibilização de descritores junto aos dados publicados e o uso de padrões de metadados são requisitos ainda não amplamente seguidos, mas de fundamental importância. Este trabalho teve como objetivo o desenvolvimento de uma aplicação para facilitar a alimentação de um repositório para divulgação do acervo de obras de artistas negros, como fotografias de acontecimentos, pinturas e esculturas do Projeto "Presenças: práticas artísticas(+)visuais indígenas e negras no Brasil" do Colégio de Aplicação da Universidade Federal do Rio de Janeiro, visando auxiliar o trabalho dos gestores dos dados. Para melhoria da experiência do usuário, uma extensão específica para o repositório também foi desenvolvida, permitindo que a busca seja filtrada e novas propriedades sejam adicionadas aos itens do repositório. Toda essa estrutura será colocada em um conjunto de máquinas que almejam a disponibilidade, evolução do armazenamento, a segurança e o isolamento dos serviços.



**Palavras-chave**: docker. conjunto. swarm. haproxy. ckan. dcat. rdf. vrrp. flask. python. javascript. metadados.

## ABSTRACT

In the era of digital information, it's extremely important that it would be accessible at all moments, organized and reusable. The search must be easy for both the end user and an automated system. In this way, tools must have useful and simple interfaces and structural components following the same principle, but at the same time efficient and capable to handle any complexities. If necessary, other tools should be made to complement funcionalities that already exist and which are built under the effort of the community in the context of free and open software. The availability of descriptors aside the published data and the use of metadata standards are requisites that aren't widely followed, but they have a crucial importance. This work has as purpose the development of an application to make easier the feeding of a repository to disseminate the collection of works of black artists such as photographs of real events, paintings and sculptures of the Projeto "Presenças: práticas artísticas(+)visuais indígenas e negras no Brasil" from the Colégio de Aplicação da Universidade do Rio de Janeiro, aiming to help the work of data managers. For a better user experience, a specific extension to the repository were developed too, allowing the search to be filtered and new properties added to the items in the repository. This whole structure will be in a set of machines that focus on availability, evolution of the storage, security and services isolation.



**Keywords**: docker. cluster. swarm. haproxy. ckan. dcat. rdf. vrrp. flask. python. javascript. metadata.

# LISTA DE ILUSTRAÇÕES

# LISTA DE CÓDIGOS

## LISTA DE QUADROS

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| AAT | *Art & Architecture Thesaurus* |
| AJAX | *Asynchronous Javascript and XML* |
| API | *Application Programming Interface* |
| CDWA | *Categories for the Description of Works of Arts* |
| CKAN | *Comprehensive Knowledge Archive Network* |
| CSS | *Cascading Style Sheets* |
| DCAT | *Data Catalog Vocabulary* |
| HTTP | *Hypertext Transfer Protocol* |
| JFIF | *JPEG File Interchange Format* |
| JPG | *Joint Photographic Experts Group* |
| JPEG | *Joint Photographic Experts Group* |
| JSON | *JavaScript Object Notation* |
| LDAP | *Lightweight Directory Access Protocol* |
| NAT | *Network Address Translation* |
| NFS | *Network File System* |
| OAI-PMH | *Open Archives Initiative Protocol for Metadata Harvesting* |
| OJS | *Open Journal System* |
| OSI | *Open Systems Interconnection* |
| PKP | *Public Knowledge Project* |
| PNG | *Portable Network Graphics* |
| QUDT | *Quantities, Units, Dimensions and Types* |
| RAM | *Random Access Memory* |
| RDF | *Resource Description Framework* |
| RFC | *Request For Comments* |

| | |
|---|---|
| ROAR | *Registry of Open Access Repositories* |
| RPC | *Remote Procedure Call* |
| SAN | *Storage Area Network* |
| SSH | *Secure Shell* |
| SQL | *Structured Query Language* |
| TCP | *Transmission Control Protocol* |
| UDP | *User Datagram Protocol* |
| UML | *Unified Modeling Language* |
| URI | *Uniform Resource Identifier* |
| URL | *Uniform Resource Locator* |
| VRRP | *Virtual Router Redundancy Protocol* |
| XHTML | *eXtensible Hypertext Markup Language* |
| XML | *Extensible Markup Language* |

**SUMÁRIO**

# 1 INTRODUÇÃO

Acesso aberto é a difusão do conhecimento deliberado sem cobranças ou qualquer obstáculo para tal ação (FACHIN; BLATMANN; CALDIN, 2019). Existem construções que têm como função permitir essa distribuição, sejam como aplicações ou sejam como organizações que administram os dados que geram esse conhecimento. De forma complementar, existem movimentos que também auxiliam na divulgação e encorajamento de dados abertos, a fim de que ocorra um acesso democrático a eles. Dentre os papéis deles, estão a discussão de políticas de acesso, campanhas de adesão ao livre acesso e respeito à legalidade das plataformas e à disponibilização dos dados (FACHIN; BLATMANN; CALDIN, 2019).

Como exemplo, existe o *Registry of Open Access Repositories* (*ROAR*), um website criado em 2003 e que atua como um catálogo para outros repositórios. Dentre os metadados, está o ano de criação, instituição pertencente e tipo de software. Como uma plataforma exemplo de dados abertos e utilizada na UFRJ no "Portal de Periódicos"[1], destaca-se o *Open Journal System* (*OJS*), desenvolvido no Canadá pela *Public Knowledge Project* (*PKP*) (Public Knowledge Project, 2025). É uma ferramenta que gerencia submissões de artigos e trabalhos científicos, permite revisões de pessoas específicas, agendamentos de prazos e a distribuição através de plataformas que atuam como catálogos, como o *Google Scholar*. Por fim, um outro exemplo é o *Dataverse*, que é um sistema de repositório em código aberto que surgiu em 2006 para compartilhamento de dados acadêmicos e é utilizado por instituições como Harvard, por exemplo. Esta plataforma apresenta uma estrutura hierárquica de repositórios e coleções de dados. Um repositório *Dataverse* hospeda coleções *Dataverse* que contêm coleções de dados efetivamente e os metadados que as descrevem. Essa estrutura hierárquica é o que permite uma ótima distribuição dos dados, já que a plataforma reúne dados dos autores e permite catalogação devido a essa característica de coleções de repositórios (The Dataverse Project, 2025b).

Neste trabalho, é descrito brevemente o Projeto "Presenças: práticas artísticas(+)visuais indígenas e negras no Brasil" do Colégio de Aplicação (CAp) da Universidade Federal do Rio de Janeiro (UFRJ), gerador do material que constitui o acervo alvo deste trabalho, o qual tem no portfólio fotografias e imagens de obras físicas, com descrições dessas fotos, assim como dos artistas que as criaram ou criaram as obras alvo das fotografias. Logo, são dados diferentes de planilhas, arquivos de textos estruturados ou mesmo vídeos.

[1] https://revistas.ufrj.br

Escolher um padrão de metadados dentre vários é também um desafio, principalmente no caso do Projeto Presenças, porque muitos são adaptados para casos específicos e necessitam de extensões de outros padrões e ontologias[2]. Esses padrões são importantes porque organizam metadados (ou até dados) através das semânticas deles, de forma que os contextos já são pré-definidos pelos padrões. Ademais, por se tratar justamente de um padrão, isso permite a facilitação da identificação das características dos recursos descritos e estruturados por eles (ALASEM, 2009). E, ainda, deve-se respeitar todos os conceitos e regras definidos neles. No caso do Projeto, de fato existe esse problema, pois a plataforma escolhida e descrita neste trabalho fornece um padrão específico que necessita de outros padrões/vocabulários/ontologias, que abriguem metadados geométricos como dimensões de "área" e "comprimento" e metadados pessoais como "endereço profissional" por exemplo.

Quanto à facilidade de uso para o usuário, implementar filtros de busca é de grande importância também. A plataforma de repositório não tem uma funcionalidade nativa de cadastramento de metadados ou para inclusão de novos. Por isso, buscou-se desenvolver uma aplicação que sirva para cadastrar os artistas e as obras, capaz de ser interligada com a interface fornecida pelo repositório e também de ser usada pelos gestores do projeto, para que eles não tenham que lidar com a dificuldade descrita. Então, não apenas visamos a facilidade para o usuário, mas também para o gestor dos dados. Outro requisito é ter essa estrutura em um conjunto de servidores com boa capacidade de armazenamento, eficiência na resposta de requisições e disponível todos os dias, além de segurança, para que os dados não sejam alterados de forma maliciosa, ferindo a confiabilidade dos usuários. Para melhoria da segurança, iremos utilizar o conceito de containers e cluster de containers, que criam isolamento das aplicações, além de facilitarem a sua instanciação.

O objetivo deste projeto é fornecer um repositório simples, mas eficaz para um usuário da comunidade que tenha interesse no acervo do Projeto Presenças. Que ele esteja disponível com a maior integralidade possível, que seja de fácil acesso, com dados e metadados organizados, com uma interface simples e com uma busca que seja possível ser filtrada com base nesses dados e metadados.

Inicialmente, os requisitos deste projeto, de acordo com a coordenadora Maya Inbar, eram apenas uma página *web* construída com linguagens de programação e soluções modernas, um banco de dados e uma interface dinâmica com grande interação para filtros de busca como, por exemplo, um mapa do Brasil ou linha do tempo. No entanto, por questões de experimentação e estudos, portabilidade, automação e conhecimentos de plataforma,

[2] Segundo (GUARINO, 1998)): "Uma ontologia descreve uma hierarquia de conceitos relacionados por uma relação de subsunções; em casos mais sofisticados, axiomas apropriados são adicionados para expressar outras relações entre conceitos e restringir a interpretação deles.". Ontologias permitem formalizar e estruturar padrões de metadados. Por exemplo, dois padrões podem ter termos com nomes diferentes, porém semânticas iguais. Isso pode ser determinado através de uma relação formal definida em uma ontologia base para os dois padrões.

optou-se por utilizar uma aplicação já existente para repositório de dados abertos, que pode ser estendida por um desenvolvedor. Além disso, a aplicação desenvolvida mencionada acima para este projeto que conecta-se ao repositório também serviu como fonte de ponderações e observações. Também é importante citar que o requisito de filtro de busca solicitado foi simplificado por dois critérios: complexidade da solicitação e necessidade de conhecimentos de *design*, sendo que esse último também não foi o foco deste trabalho tanto na aplicação desenvolvida quanto na plataforma de repositório.

Este trabalho tem relevância porque reúne novos conceitos para estudo e demonstrações no mundo real. Além disso, a plataforma de repositório considerada é extremamente simples. Podem haver alternativas mais robustas, a exemplo da plataforma *Dataverse*, mas deve-se considerar que também exigem maior esforço para sua implantação e manutenção (CAMPêLO; NETO, 2020). No entanto, demonstra-se que há viabilidade com a ferramenta utilizada e que a manutenibilidade não é complicada. Ademais, este trabalho atua na disponibilidade de dados de um projeto real da UFRJ, que deve ser exposto à comunidade geral e que hoje não possui um ambiente atendendo a isso.

Este trabalho encontra-se organizado segundo 7 capítulos, além desta introdução: O capítulo 2 explica o que é e como está atualmente o Projeto Presenças e seus dados. O capítulo 3 exibe e comenta as ferramentas e padrões que utilizamos, bem como motiva o que e por que considerar o uso de *containers* e *Docker*, além de discutir seus usos. O capítulo 4 descreve a solução *Comprehensive Knowledge Archive Network* (*CKAN*) utilizada para o projeto e mostra a organização dos metadados dentro do padrão escolhido. O capítulo 5 descreve a criação da aplicação para cadastros, explica como funciona, como foi desenvolvida e como está organizada. O capítulo 6 mostra o desenvolvimento de uma extensão com base nas interfaces de programação oferecidas pela plataforma do repositório para que novos filtros sejam implementados e correções em outra extensão sejam aplicadas. O capítulo 7 explica e dá um exemplo de construção de um conjunto de servidores para abrigar as aplicações descritas neste trabalho. Mostra-se a esquematização e a configuração dos sistemas e das ferramentas. Por fim, o capítulo 8 conclui o trabalho resumindo o que foi dito, expondo limitações encontradas e sugestões para trabalhos futuros.

# 2 PROJETO PRESENÇAS

Neste capítulo, iremos falar brevemente do Projeto Presenças, o que é ele, como surgiu e se estrutura como projeto de extensão da UFRJ e como os dados apresentam-se e estão dispostos hoje.

## 2.1 DESCRIÇÃO

O Projeto Presenças, do Colégio de Aplicação da UFRJ, tem como objetivo pesquisar e disseminar a produção artística negra e indígena no Brasil. Baseia-se em debates, leituras e encontros para a fundamentação dessa pesquisa. Além disso, como consequência, o projeto permite que ocorram uma propagação da cultura negra e indígena e a criação de referências para novos pesquisadores e público em geral nessa área. É um projeto de extensão e foi criado em 2019 e está em atividade até hoje. Alguns dos objetivos são:

a) Criação de um catálogo de livre acesso contendo parte da produção visual e artística negra e indígena no Brasil;

b) Constituir-se um espaço de produção e pensamento antirracista;

c) Fortalecer a formação docente para as relações étnico-raciais, ampliando os repertórios pedagógicos nas Artes Visuais, bem como contribuir para uma formação voltada à diversidade e com um embasamento nas pedagogias, cosmovisões e expressões artísticas negras e indígenas.

O projeto possui um site simples[1] com descrição e eventos relacionados.

## 2.2 A ESTRUTURA DOS DADOS

Originalmente, os participantes do Projeto Presenças vêm mantendo todas as informações em pastas no ambiente do *Google Drive*. O projeto é organizado em artistas com subdivisões de obras. Cada artista está em um diretório próprio e dentro de cada um deles constam suas respectivas obras. Cada obra tem um nome que começa com as iniciais do nome e sobrenome do artista, seguido de subtraço, uma numeração e o formato do arquivo. Existe um arquivo em especial em cada pasta, que é o retrato do artista. É uma imagem com praticamente o mesmo padrão de nome, só que com a palavra "retrato" ao invés de uma numeração. Cada diretório contém um documento em formato PDF e formato DOC contendo a documentação do artista e as obras no diretório. Os documentos contêm as seguintes informações associadas (metadados):

a) Nome;

[1] https://presencascapufrj.wordpress.com/sobre-o-projeto/

b) Trajetória;

c) Produção;

d) Referências;

e) Última atualização;

f) Lista de imagens.

Para cada imagem, apresentam-se:

a) Descrição;

b) Fonte;

c) Título;

d) Técnica da obra;

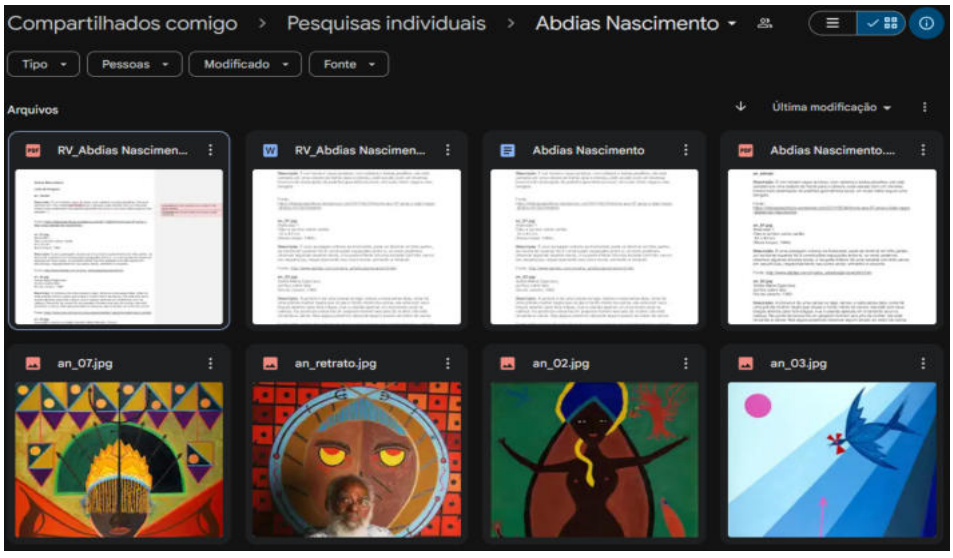
e) Comprimento ou área;

f) Ano.

Alguns documentos não contêm esses campos, outros possuem outros campos como série (categoria) e local de criação da obra. Já alguns artistas não possuem o documento referência associado. Em termos de automatização, isto é um desafio que remete a vários casos especiais para tratamento.

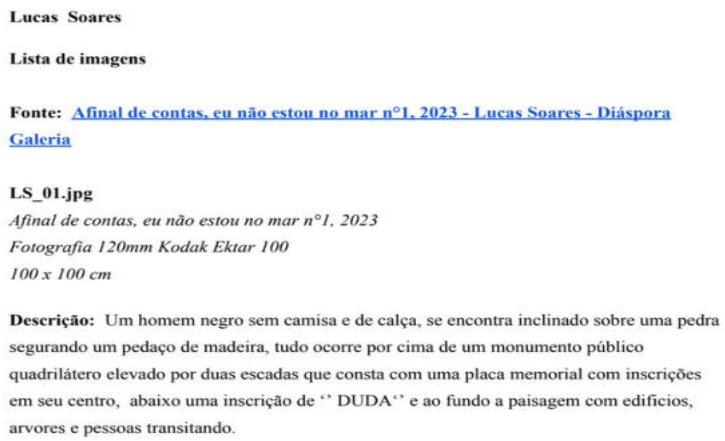
A figura 1 mostra o conteúdo de um diretório de um artista no *Google Drive* como exemplo. As figuras 2 e 3 mostram um exemplo de documento de um artista com os campos mencionados acima. Este artista escolhido, por exemplo, não possui a data de última atualização no documento. O campo "Referências" seria a seção "Fontes" nesse arquivo.

Figura 1 – Exemplo de diretório de um artista




Fonte: Elaboração Própria

Figura 2 – Exemplo de um documento referência de um artista



**Lucas Soares**

**Lista de imagens**

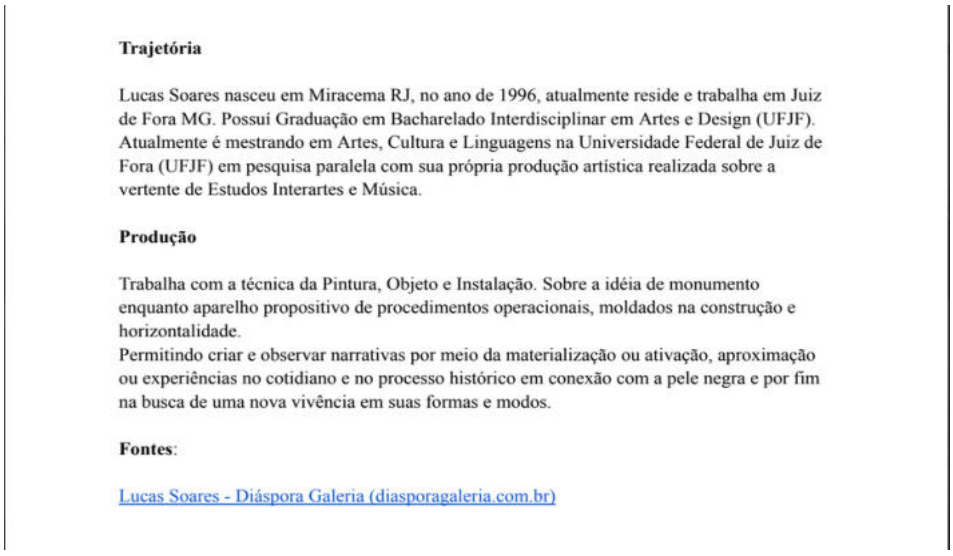
**Fonte:** Afinal de contas, eu não estou no mar n°1, 2023 - Lucas Soares - Diáspora Galeria

**LS_01.jpg**
*Afinal de contas, eu não estou no mar n°1, 2023*
*Fotografia 120mm Kodak Ektar 100*
*100 x 100 cm*

**Descrição:** Um homem negro sem camisa e de calça, se encontra inclinado sobre uma pedra segurando um pedaço de madeira, tudo ocorre por cima de um monumento público quadrilátero elevado por duas escadas que consta com uma placa memorial com inscrições em seu centro, abaixo uma inscrição de '' DUDA'' e ao fundo a paisagem com edifícios, arvores e pessoas transitando.

Fonte: Elaboração Própria

Figura 3 – Exemplo de um documento referência de um artista (continuação)



**Trajetória**

Lucas Soares nasceu em Miracema RJ, no ano de 1996, atualmente reside e trabalha em Juiz de Fora MG. Possui Graduação em Bacharelado Interdisciplinar em Artes e Design (UFJF). Atualmente é mestrando em Artes, Cultura e Linguagens na Universidade Federal de Juiz de Fora (UFJF) em pesquisa paralela com sua própria produção artística realizada sobre a vertente de Estudos Interartes e Música.

**Produção**

Trabalha com a técnica da Pintura, Objeto e Instalação. Sobre a idéia de monumento enquanto aparelho propositivo de procedimentos operacionais, moldados na construção e horizontalidade.
Permitindo criar e observar narrativas por meio da materialização ou ativação, aproximação ou experiências no cotidiano e no processo histórico em conexão com a pele negra e por fim na busca de uma nova vivência em suas formas e modos.

**Fontes**:

Lucas Soares - Diáspora Galeria (diasporagaleria.com.br)

Fonte: Elaboração Própria

Além dessas informações, temos também uma planilha centralizada contendo diversos dados para cada artista. No entanto, nem todos serão usados no mapeamento dos metadados. Os seguintes campos estão presentes na planilha citada:

a) Item;

b) Pesquisante;

c) E-mail pesquisante;

d) Individuo/coletivo/povo;

e) Data início/nasc;

f) Data fim/Falecimento;

g) Linguagem(ns) (tipo de obra);

h) Cidade de origem;

i) Estado de origem;

j) Cidade(s) de atuação;

k) Estado(s) de atuação;

l) País de atuação;

m) Palavras-chave;

n) Gênero;

o) Link p/ site;

p) Data da pesquisa;

q) Revisor/a;

r) Revisor/a 2;

s) Status;

t) Data Revisão.

Portanto, no nível de organização base, os dados estão bem separados e seguem um bom padrão, apesar da não uniformidade para todos os artistas.

# 3 TRABALHOS RELACIONADOS E PADRÕES E FERRAMENTAS UTILIZADAS

Neste capítulo, iremos descrever a maioria das ferramentas principais e auxiliares que compõem o projeto para permitir um maior entendimento da estrutura geral, principalmente a respeito de *containers*. Também serão descritos dois trabalhos com propostas e temas parecidos com este.

## 3.1 TRABALHOS RELACIONADOS

A *string* (cadeia de caracteres) *"'repositórios' AND ('artísticos' OR 'artistas' OR 'pinturas' OR 'trabalhos')"* foi utilizada para busca de trabalhos relacionados em plataformas como *Sol*, *SciELO*, *Google Scholar* e o próprio *Google*. Também foi utilizada uma versão em inglês: *"'repositories' AND ('artistics' OR 'artists' OR 'paintings' OR 'works')"*. Dois trabalhos foram encontrados com relevância nos assuntos descritos no presente trabalho. Inclusive, a similaridade de um é muito próxima justamente pelo tratamento de metadados, uso de uma plataforma existente e hospedagem de um acervo cultural e disponibilização para a comunidade.

Em (SOARES, 2012), o autor apresenta a proposta de um repositório digital com uso do *DSpace* e o padrão de metadados *Dublin Core* para organização deles. Os objetivos e a metodologia são muito parecidos com a deste trabalho. No entanto, o presente trabalho adicionalmente discorre sobre a infraestrutura dos servidores, configuração de ferramentas e uso de protocolos para criar essa infraestrutura mencionada, com o objetivo de garantir maior disponibilidade e consistência dos serviços. O acervo do repositório é composto pelas obras de Odilla Mestriner (PRANDI, 2011). Uma das motivações do trabalho é a maior difusão com a digitalização dos museus pelo encurtamento da distância, disponibilidade quase integral e ausência de limitações físicas. Faz-se uma breve revisão de museus e da ciência que os estuda: museologia; repositórios digitais, a plataforma de repositório *DSpace* e o protocolo *Open Archives Initiative Protocol for Metadata Harvesting* (*OAI-PMH*) que define como deve-se coletar registros de metadados em repositórios. O autor descreve a história de Odilla Mestriner, como são os dados com que trabalha, a organização deles, mapeia os metadados nos formatos digitais conforme os protocolos e a ferramenta de repositório e também descreve todo o aparato que é usufruído. Ele conclui o trabalho com a disponibilização dele sendo um manual e ressaltando a existência da ferramenta que atende a objetivos similares ao deste trabalho. É importante citar que existe um campo em comum com a mesma semântica de um campo do Projeto Presenças, que é o campo "técnica". A decisão tomada por ele não foi adotada em nosso projeto como veremos.

Em (PATRA, 2008), a autora propõe o uso de uma plataforma relativamente antiga chamada "*Greenstone*" também com o uso de *Dublin Core* como padrão de metadados

para catalogação de obras cerâmicas. A UNESCO e a Universidade de Waikato são as desenvolvedoras da plataforma[1]. O trabalho também revisa o *Dublin Core*, mas adicionalmente cita, revisa e utiliza o *Categories for the Description of Works of Arts* (*CDWA*), que é um conjunto de padrões, boas práticas e descrições para descrever trabalhos de arte, cultura e arquitetura (BENOFF; BACA; HARPING, 2024). Também inclui revisão de terminologias gerais a respeito do assunto. Apenas o estudo dos metadados e a temática do repositório foram importantes para a revisão.

## 3.2 A PLATAFORMA DE REPOSITÓRIO *CKAN*

Como já citado, nosso objetivo é construir um repositório que contenha os artistas e obras do Projeto Presenças de forma organizada, catalogável e acessível respeitando os princípios *FAIR* (WILKINSON et al., 2016), isto é, dados acessíveis, localizáveis, interoperáveis e reutilizáveis. O *CKAN* é uma ferramenta que permite criar sites para armazenamento de coleções de dados abertos (Open Knowledge Foundation, 2022) e é desenvolvido pela *Open Knowledge Foundation*. O principal uso do *CKAN* é feito por governos nacionais, como o Brasil, Canadá e muitos outros países, com a intenção de disponibilizar dados governamentais como estatísticas populacionais.

O *CKAN* permite uso de filtros para busca e indexação e possui pré-visualizações para recursos como gráficos e tabelas. Ele é escrito em *Python*, uma linguagem de programação de uso fácil e intuitivo. Utiliza um *framework* chamado *Flask*, que também tem como preceito a simplicidade e facilidade para criar páginas *web* e gerenciar sessões.

Das aplicações que servem ou podem servir como repositórios a exemplo do *Dataverse* e *DSpace*, ele é o que tem construção e uso mais simplificados (ROCHA et al., 2021), principalmente quanto aos metadados, versionamento e integridade histórica e descentralização dos dados. Dessa forma, a personalização dele é mais complicada, mas não impossível como veremos. Ele foi escolhido para uso devido a algumas razões, sendo elas:

a) Pela familiaridade, sua facilidade para instanciação e armazenamento dos dados de forma simplificada, sua utilização foi bem-vinda.

b) O *CKAN* possui uma *API* que permite a criação de um *script* que cria requisições para a plataforma. Ela tem uma documentação de fácil leitura e a biblioteca atende aos procedimentos necessários.

c) A desenvolvedora do *CKAN*, criou uma extensão que armazena e gera representações de metadados. As extensões são uma funcionalidade adicional no *CKAN* que permitem que desenvolvedores mudem o comportamento tanto visual quanto funcional da plataforma. A extensão em questão é para o vocabulário *DCAT*, que será descrito mais à frente.

[1] https://www.greenstone.org/

A aplicação tem uma estrutura simples de uso e construção. Um repositório padrão contém organizações, grupos e conjuntos de dados. As organizações permitem que os conjuntos de dados pertençam a elas. E, dentro das organizações, os usuários criados podem ser alocados com diferentes papéis administrativos (criar, editar ou publicar) para os conjuntos de dados. Os grupos servem para dividir os conjuntos de dados em diferentes categorias, independentemente de organizações ou não. O uso padrão é de maior simplicidade ainda. Após criar uma organização qualquer, basta adicionar um conjunto de dados e novos recursos a esse conjunto. A página de conjunto de dados possui alguns campos como nome, descrição, mantenedor, entre outros que veremos no capítulo seguinte, que trata sobre os dados do Projeto Presenças. E, assim como os conjuntos de dados, os recursos deles (arquivos) possuem campos também para preenchimento.

O *CKAN* possui uma interface administrativa via linha de comando que permite criar/remover usuários, torná-los superadministradores, corrigir indexação de conjuntos de dados, atualizar o banco de dados e outras funcionalidades (Open Knowledge Foundation, 2025b). No entanto, para conteúdo deste trabalho e melhor experiência final ao usuário, temos como objetivo estender o *CKAN* com organização de metadados, filtros de pesquisa e campos adicionais para os conjuntos de dados e recursos. Como mencionado, o *CKAN* permite uso de extensões, que são conjuntos de *plugins* escritos na mesma linguagem de programação da aplicação. O *CKAN* possui interfaces, classes e funções pré-definidas para auxiliar os desenvolvedores (Open Knowledge Foundation, 2025d).

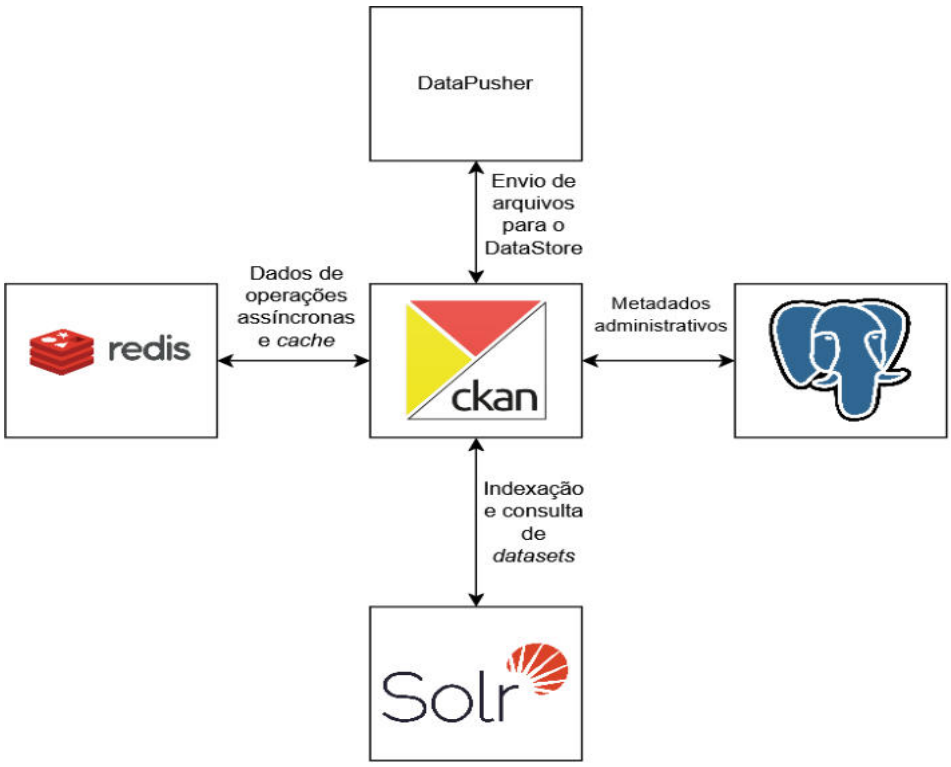
É possível modificar praticamente qualquer funcionalidade como autenticação, busca/indexação/criação de conjuntos de dados, permissões gerais e outras. Não somente modificar, como adicionar. Por padrão, o conceito de metadado é abstraído para "extras", isto é, um mapeamento chave-valor. Para recursos, esses metadados extras só estão disponíveis via inserção/modificação pela *API* (*Application Programming Interface*). Não há uma coleta de metadados nem uma forma de obter esses metadados em um formato padrão, como o *Resource Description Framework* (*RDF*). A extensão *Data Catalog Vocabulary* (*DCAT*), que adiciona a funcionalidade de obter metadados em conformidade com o vocabulário *DCAT*, é escrita pela própria desenvolvedora do *CKAN*, é documentada e também permite que seja modificada para adicionar novos perfis. Iremos falar do *DCAT* e *RDF* mais à frente.

O *CKAN* não possui um automatizador de operações, um verificador de padrões, a capacidade de entender a estrutura de documentos, capacidade direta para criar/inserir metadados de forma nativa e também modificar a indexação/filtragem de datasets e recursos é extremamente dificultada. E, por fim, a adição de novas páginas e a alteração visual do *CKAN* é dependente do conhecimento do *framework Flask* e de programação na linguagem *Python*, isto é, não há uma espécie de *low-code* embutido no *CKAN* para tais objetivos. No entanto, este trabalho irá mostrar que é possível, apesar de todos os empecilhos, criar um repositório que realiza grande parte das modificações descritas neste

parágrafo.

A figura 4 contém o diagrama da estrutura do *CKAN*. Ele depende de um banco de dados *PostgreSQL* para o armazenamento das dados estruturais de indexação, usuários e outros.

Figura 4 – Diagrama dos componentes do *CKAN*




Fonte: Elaboração própria

O *Redis*, que é um banco de dados não relacional, é utilizado para armazenar dados de caches e operações assíncronas de segundo plano (Open Knowledge Foundation, 2025a). No caso das *caches* seriam de sessões, dados de datasets requeridos pelos usuários, renderização das páginas etc. No caso das operações assíncronas, é importante o uso do banco porque, se o *CKAN* realiza uma operação que demanda uma grande leitura/escrita e tempo, outras rotinas seriam comprometidas, tornando inutilizável a plataforma. Logo, de forma paralela e com auxílio do *Redis*, é possível atender a todas as operações.

O *Apache Solr* terá a explicação feita na próxima seção por ser importante para a construção da extensão desenvolvida por nós. Basicamente, ele é utilizado para indexação e consulta dos *datasets*.

Por fim, o *DataPusher* atua na automatização do envio e gestão de grandes arquivos. Ele é a ferramenta que trata o dado e adapta-o para o *DataStore*. O *DataStore* armazena metadados administrativos no banco de dados do *CKAN* e o dado em si é colocado em um sistema de arquivos ou em nuvem. Ele é responsável pelo processo de envio e gestão do arquivo em questão. Com isso, ele é capaz de fornecer uma *URL* para o arquivo e

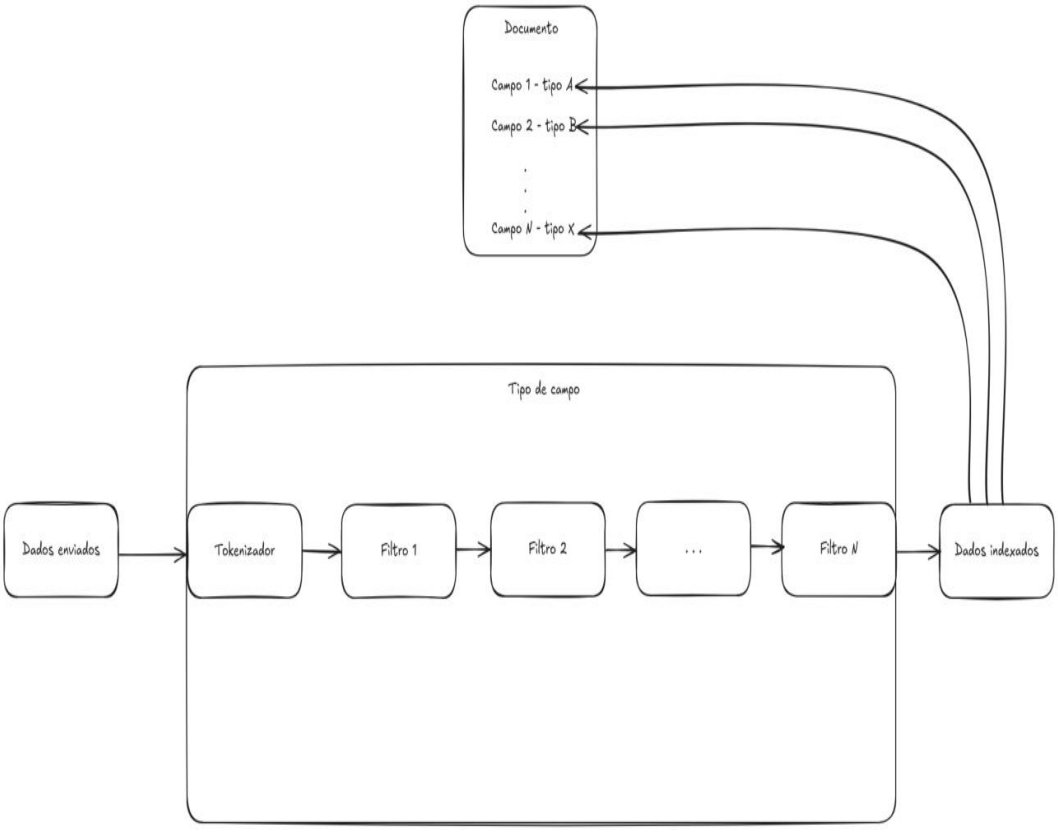
automaticamente enviar e controlar as atualizações deste arquivo também. O controle dos dados a partir do envio é feito através da *API* do *DataStore*. Através dela, é possível filtrar e procurar pelo dado sem precisar baixá-lo, além de permitir que ele seja pré-visualizado e utilizado por outras extensões. O *DataStore* é considerado um banco de dados pela documentação do *CKAN* (Open Knowledge Foundation, 2025c), pois dados que não estão no *DataStore* ficam no banco de dados *PostgreSQL* da plataforma. Para arquivos maiores, essa abordagem não é eficiente e, portanto, o uso do *DataStore* é recomendado.

## 3.3 *APACHE SOLR*

Toda busca e indexação dos datasets e recursos no *CKAN* é organizada pelo *Apache Solr* (Apache Software Foundation, 2022). Ele é uma ferramenta que gere documentos. **Documentos** possuem campos de diversos tipos. Documentos pertencem a **coleções**, e cada coleção possui um esquema que pode ser gerido. Os esquemas possuem configurações de tipos e campos. Cada tipo de campo é dado por uma classe implementada no *Solr*, como *TextField*, *BooleanField*, entre outras. Para *TextFields* ou campos de texto, existem analisadores. **Analisadores** são responsáveis por construir índices com os dados recebidos e para processar as consultas que levam aos índices criados. Eles geram o que chamamos de ***tokens*** para os campos tanto na consulta quanto na indexação. Os *tokens* são partes menores dos dados recebidos em um campo específico. Um analisador possui um pipeline composto por *tokenização* e filtragens. A *tokenização* é responsável por alterar sequências de caracteres como remover espaços em brancos, pontuações etc. Os **filtros** são responsáveis por substituir ou descartar um *token*. Por exemplo, derivações de palavras podem ser mapeadas para apenas a palavra primitiva, como "inconstitucional"para "constituição". Ou juntar duas palavras para uma sigla como "RH"para "recursos humanos". A combinação de *tokenização* e filtros cria um analisador. Como o documento original pode ser alterado na indexação, consultas com palavras que não estavam nele podem remeter a eles. Essa é uma razão para a consulta ser separada da indexação ao construir analisadores. É importante diferenciar dados armazenados e indexados. Isso porque é necessário obter a informação original, porém de forma eficiente se comparado à busca normal, i.e. crua. Os dados indexados são os resultantes do processo de análise feito ao armazenar um documento. Eles são utilizados na comparação durante uma consulta (*query*). A figura 5 contém um diagrama da estrutura básica de um documento e de como funciona o processo de análise de dados recebidos para gerar dados indexados.

O *upload* e a indexação de documentos ocorrem basicamente de três formas: utilizar um *Solr Cell* para arquivos binários como PDF, *Word* e *Excel*, enviar documentos em formato *XML/JSON* via a *API HTTP* ou escrever uma aplicação em *Java* que realize o trabalho de indexação de acordo com a *API* de cliente do *Solr*. A primeira baseia-se em converter o documento para um em formato *XHTML* que depois pode ser tratado na

Figura 5 – Diagrama de estrutura de um documento e análises de campos



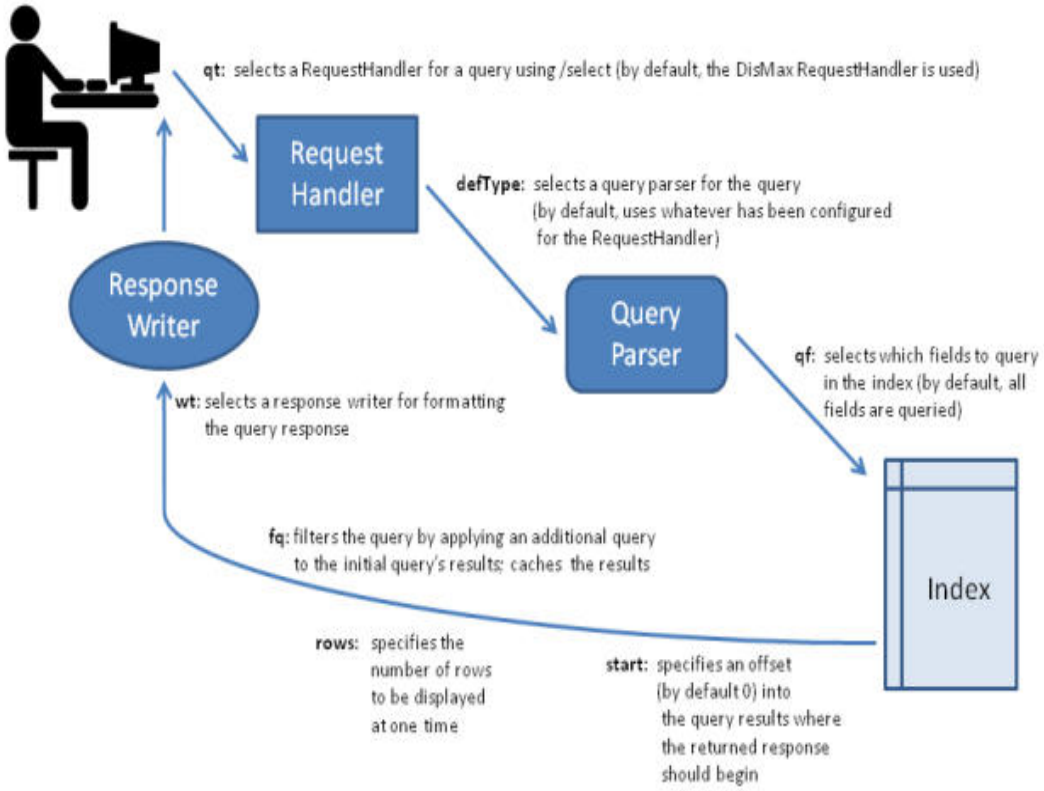

Fonte: Elaboração Própria

indexação. A segunda, que é a mais simples e mais utilizada, inclusive pelo *CKAN* como veremos, baseia-se em simplesmente enviar um *XML* e *JSON* estruturados de acordo com a documentação do *Solr* para criar um novo documento. E a terceira é auto-explicável.

A dinâmica de busca do *Solr* é um grande atrativo da ferramenta. Ela é estruturada com um gerenciador de requisições, que repassa a consulta para um parser (analisador) de texto. O *Solr* oferece três parsers: simplificado (também conhecido como Lucene), que é o padrão, *DisMax*, que permite uma consulta que intensifica e valoriza certos campos de acordo com valores numéricos e é mais tolerante a erros e o *eDisMax* que é uma extensão do *DisMax* com mais tolerância de erros e parâmetros adicionais. Cada *parser* recebe parâmetros específicos que têm suas funcionalidades documentadas. Além disso, como em consultas *SQL*, existem funções que auxiliam as consultas no *Solr*, como médias, mínimo etc. Por exemplo, numa consulta podemos obter documentos com um título $X$ e com data em um certo intervalo de tempo, ou documentos que foram emitidos a partir de uma data, ou o documento com um campo cujo valor é o mínimo de todos os documentos na coleção. E uma grande funcionalidade na busca do *Solr* é a busca facetada, isto é, categorizada. Ela permite que campos específicos sejam escolhidos para dividir as respostas de consultas, e também servem para contar o número de documentos

dentro de uma categoria. Também é possível reduzir ou aumentar os documentos nessas categorias com base em parâmetros como limites e distinção de maiúsculo e minúsculo. A figura 6 mostra como funciona a busca no *Solr*. Mais informações podem ser consultadas na documentação oficial do *Solr* (Apache Software Foundation, 2022).

Figura 6 – Diagrama de busca no Solr




Fonte: https://solr.apache.org/guide/8_9/overview-of-searching-in-solr.html

## 3.4 *RDF*: A BASE PARA O MAPEAMENTO DOS METADADOS

Iremos descrever o *Resource Description Framework* (*RDF*) brevemente. Não é a intenção deste trabalho realizar esta tarefa de forma completa e exaustiva, já que as documentações citadas nesta seção definem todo o modelo. Nossa intenção é dar um entendimento de forma geral para que seja possível compreender que a estrutura do *DCAT* é quase que inteiramente moldada em cima da estrutura do *RDF*, já que ele é um vocabulário *RDF*. O *DCAT*, que será descrito na próxima seção, é um vocabulário *RDF* criado para facilitar a interoperabilidade entre catálogos de dados na *Internet* (ALBERTONI et al., 2024).

O *RDF* é um *framework* para representar informações na *web* (HARTIG et al., 2025). Basicamente, o modelo *RDF* é constituído de grafos e conjuntos de grafos. Um grafo *RDF* é dado por triplas, onde as triplas são chamadas de declarações e têm a estrutura

de *sujeito-predicado-objeto*. Por serem grafos, sujeitos e objetos são **nós** e predicados são **arcos** (um grafo direcionado). A noção de hierarquia e relacionamentos surge dessa natureza da definição de grafos. Cada nó pode ser um *IRI*, literais, nós em branco e outras triplas.

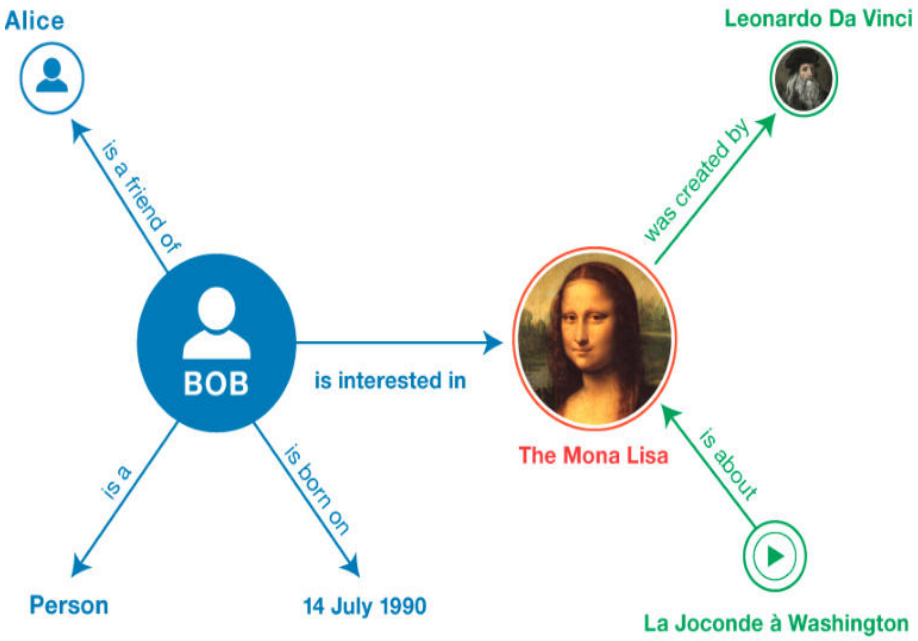
### 3.4.1 Elementos básicos da representação em *RDF*

Um **recurso** pode ser qualquer coisa, como documentos, conceitos abstratos, números etc. Um ***IRI*** (*Internationalized Resource Identifier*) (DüRST; SUIGNARD, 2005) é um identificador que complementa a definição de *URI* (*Uniform Resource Identifier*) (BERNERS-LEE; FIELDING; MASINTER, 2005) por permitir mais tipos de caracteres na *string* e serve para localização e obtenção de recursos, ou seja, uma espécie de identificador do caminho de algo. Um **literal** é definido num "*range*" de valores possíveis, que são tipos de dados como *strings*, números, datas etc. Literais são exatamente o que o nome sugere, valores finais que podem ser atribuídos a outros recursos. Eles têm quatro elementos de acordo com o modelo *RDF*. A forma léxica, que é uma sequência codificada de caracteres ou valores escalares como números, um tipo de dado, uma etiqueta de linguagem padronizada caso o tipo seja *string*, para que saibamos o idioma e também, caso o tipo seja *string*, um indicador padronizado de sentido de leitura, isto é, da esquerda pra direita ou vice-versa.

Uma **tripla *RDF*** define uma relação dada pelo predicado dela entre os recursos dados pelo sujeito e objeto. Um vocabulário *RDF* é um conjunto de *IRIs* para serem utilizados em grafos RDF (HARTIG et al., 2025). Isto é, definindo-se triplas, estas tornam-se recursos que podem ser identificadas e reutilizadas por outros grafos. Um exemplo disso é o próprio *DCAT* que já mencionamos. Em termos de *RDF*, ele é um conjunto de *IRIs* com triplas constituídas definindo tipos de dados, descrições e usos semânticos para outros recursos. Isso é feito utilizando o esquema *RDF*, que falaremos em breve. Com a esquematização é possível criar documentos *RDFs* que podem ser compartilhados entre sistemas, o objetivo principal do *RDF*. Documentos *RDF* nada mais são que grafos ou conjuntos de grafos *RDF* codificados em uma sintaxe. Algumas delas são o *XML* (KELLOGG, 2025) e o *Turtle* (KELLOGG, 2025).

Na figura 7, temos um exemplo de relações de ***sujeito-predicado-objeto*** com reutilização de recursos. Cada nó é um sujeito ou objeto, podendo ser os dois ao mesmo tempo no grafo, dependendo da tripla que participa. Vamos tomar a Mona Lisa nesta figura como exemplo. Na relação *"was created by"*, ela é o sujeito passivo e Leonardo da Vinci é o objeto. Com essa tripla, entendemos que ela foi pintada por ele. Mas, ela também é um objeto nesse grafo. Na tripla *Bob-"is interested in"-Mona Lisa*, o Bob é o sujeito e vemos que ele tem interesse na Mona Lisa.

Figura 7 – Grafo informal de triplas *RDF*




Fonte: (HARTIG et al., 2025)

### 3.4.2 Esquemas em RDF

O esquema RDF (TOMASZUK; HAUDEBOURG, 2025) é uma forma de modelar os dados em *RDF* em uma forma diferente. Tem como objetivo criar novos recursos para descrever outros deles em *RDF*, os quais possuem relacionamentos entre eles e como se dão essas relações. Utiliza um formato de classes e propriedades similar à de orientação ao objeto. No entanto, ao invés de definir uma classe em termos de propriedades, é preferível definir propriedades em termos de classes. Isso facilita a adição de novas propriedades, já que um dos objetivos do *RDF* é o reuso de recursos. A linguagem usada aqui é a de coleção de recursos *RDF* que podem descrever outros recursos. Existe um termo chamado namespace nos conceitos do *RDF*. Não existe uma definição específica, mas podemos dizer que trata-se de um vocabulário *RDF*. Um vocabulário *RDF* possui um nome, por exemplo, o *DCAT*. E, como um recurso, tem uma *IRI* que leva aos grafos *RDF*. Podemos utilizar uma sigla para representar esta *IRI*. Por exemplo, a *IRI* "https://www.w3.org/ns/dcat3.ttl" pode ser mapeada para a sigla *dcat*. E com símbolo de dois pontos (:) seguido de uma palavra, acessamos um termo do vocabulário/esquema. No apêndice A estão descritos dois dos principais componentes do esquema *RDF* e dado um exemplo de uso do *RDF* na construção do *DCAT*. O padrão descrito para o *namespace* do *RDFS* (*RDF Schema*) é utilizado na documentação, no apêndice A e no capítulo 4 quando tratamos os metadados.

## 3.5 *DCAT*

O *DCAT* é um vocabulário que visa organizar recursos em formatos como o *XML* e *Turtle* para catálogos de forma que a publicação e as consultas ficam facilitadas e estruturadas. Ou seja, o objetivo é criar classes *RDF* para descrever catálogos e recursos. A definição oficial dele foi dada na seção anterior, assim como uma definição com base na terminologia do RDF.

O contexto principal do *DCAT*, assim como o do *CKAN*, é para aplicações governamentais (ALBERTONI et al., 2024), mas nada impede que seja utilizado em outros contextos, como mostraremos aqui. A necessidade de utilizar um padrão de metadados existe por dois motivos, basicamente: protocolar a organização dos metadados e diminuir o esforço nesse trabalho de organização, já que o vocabulário estabelece as classes e propriedades específicas para cada metadado existente em um conjunto de dados. Adicionalmente, os esquemas *RDF* criam o conceito de consultas estruturadas aos dados (W3C SPARQL Working Group, 2013). Se os dados estão esquematizados conforme definido nos padrões, a automatização e corretude das tarefas são obtidas. Utilizaremos esse padrão pela razão de ser fornecido pela desenvolvedora do *CKAN* e por ser extensível como veremos no capítulo 4. O *DCAT* possui basicamente as classes:

a) ***Resource***: É a classe pai de outras e não deve ser instanciada diretamente. Representa qualquer coisa que pode ser descrita com metadados.

b) ***Catalog***: Segue o conceito de catálogo, ou seja, a reunião de metadados de datasets e que serve como uma lista deles.

c) ***Dataset***: Representa conjunto de dados criado e mantido por um agente ou grupo. Essa classe funciona como o descritor de acomodadores de dados, onde eles são considerados recursos em diversas formas como texto, imagens etc.

d) ***Distribution***: Representa um recurso de um *dataset* como um arquivo.

e) ***DataService***: Representa, basicamente, funções de uma *API* para coleta e processamento de dados.

f) ***DatasetSeries***: Representa um *dataset* que é a coleção de outros *datasets* publicados em lugares diferentes, mas com características em comum.

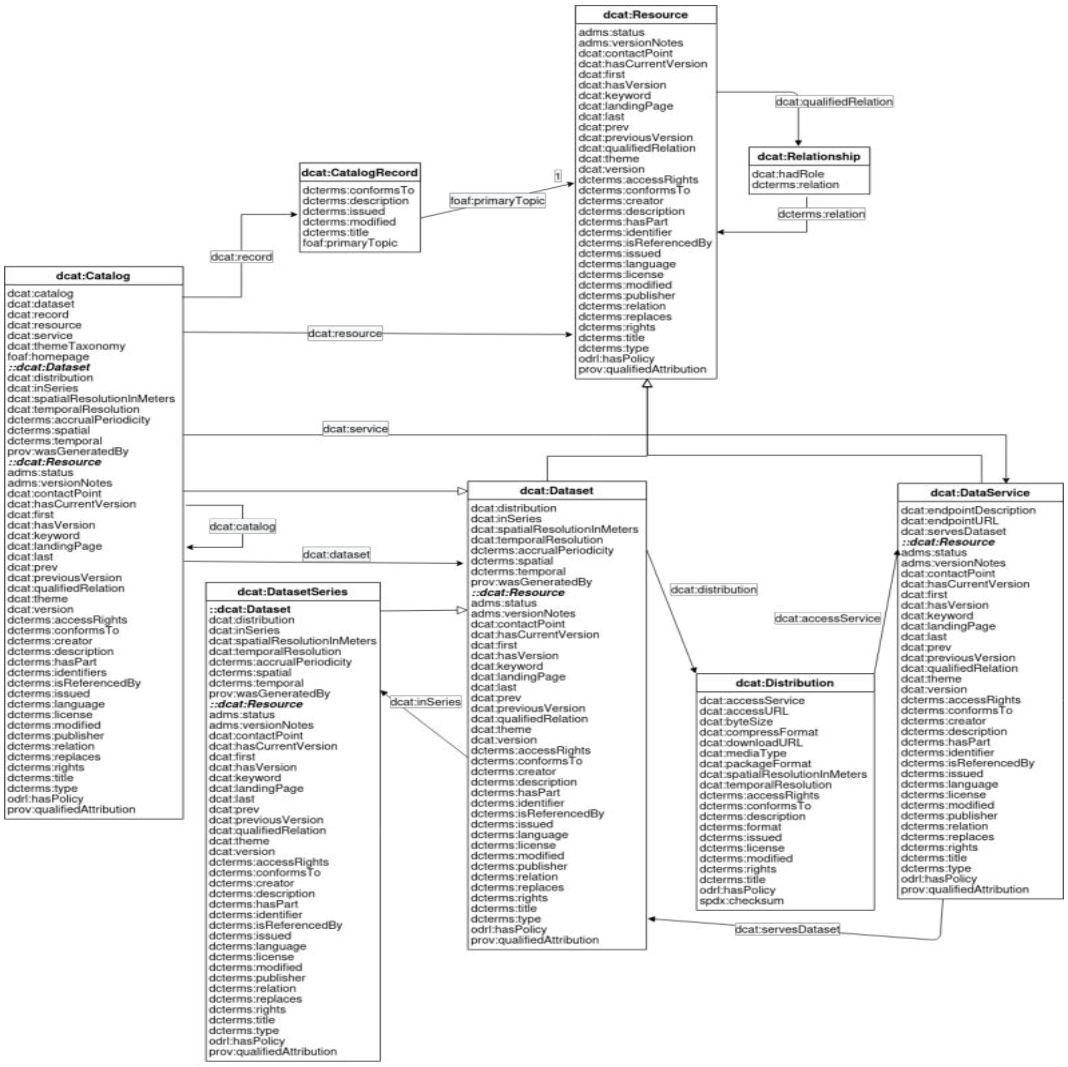
g) ***CatalogRecord***: Representa um registro de um item em um catálogo, como data de adição e quem fez a adição.

Contudo, como estamos atrelados à plataforma do *CKAN*, apenas as classes utilizadas por ele serão nosso objeto de estudo. A figura 8 apresenta o diagrama do *DCAT*. É possível visualizar praticamente todas as propriedades nos domínios das classes citadas.

É importante citar que existe um *framework* para tratar metadados de trabalhos de artes e culturais já citado na seção de trabalhos relacionados, que é o *CDWA*. O *CDWA* juntamente com o *Art & Architecture Thesaurus* (*AAT*) (Getty Research Institute, 2021)

padronizam terminologias relacionadas a trabalhos de cultura e arte, bem como permitem uma melhor catalogação, indexação e descrição desses trabalhos. No entanto, eles não são vocabulários com implementações em esquema *RDF*, isto é, são apenas padrões. Uma implementação em *RDF* que os utiliza é o vocabulário *VRA Core* (Library of Congress, 2022), que veremos no capítulo 4 no mapeamento dos metadados.

Figura 8 – Visão geral do diagrama do modelo *DCAT*




Fonte: (ALBERTONI et al., 2024)

## 3.6 *CONTAINERS* E *DOCKER*

### 3.6.1 Definição de *container*

Para definir o tema central desta seção, *container*, é necessário entender os artifícios necessários para a existência dele. O *kernel Linux*, a partir da versão 2.6.24, implementou o uso de *control groups* (*cgroups*) (Michael Kerrisk, 2025a) e estendeu os *namespaces*

(Michael Kerrisk, 2025b), implementados em 2002. Esses dois recursos são os pilares para a capacidade de criar e manipular um *container*.

A funcionalidade de *namespace* permite que alguns itens do sistema como ID de processos, rede, pontos de montagens e usuários e grupos sejam mapeados ou alterados para novos valores (Michael Kerrisk, 2025b). Ou seja, isso permite que o processo enxergue um cenário diferente de outros processos, uma abstração. É importante citar que *namespaces* são construídos em uma hierarquia. Vários processos podem estar em um mesmo *namespace* e processos de um *namespace* filho não enxergam os dos seus ancestrais. Historicamente, antes dos *namespaces*, existia apenas uma árvore de processos, cujo PID (*Process IDentification*) 1 era o processo *init* de um sistema *Linux* comum. Com o advento de *namespaces*, a segregação de novas árvores de processos e recursos de sistema como citados anteriormente, processos podem ser separados em grupos praticamente isolados uns dos outros.

Para auxiliar nesse objetivo, os *cgroups* (Michael Kerrisk, 2025a) são grupos de controle de recursos (processador, memória, banda de rede, etc.) nos quais os processos ou grupos de processos podem ser alocados, assim como em *namespaces*. De forma granular, é possível limitar a quantidade de recursos que os processos podem utilizar, o que é importante para que os processos respeitem o uso de recursos de outros.

Portanto, *container* é um ambiente em que um ou mais processos executam com alguns ou todos os recursos isolados do sistema hospedeiro. Esse ambiente é construído com o uso dos mecanismos citados acima. De forma comparativa à virtualização utilizando um hipervisor, é praticamente similar a forma que eles visualizam a posse de um ambiente novo. Eles acreditam que estão em uma máquina isolada, com todos os recursos para eles. Todos os processos em um *container* compartilham o mesmo núcleo do sistema hospedeiro, por isso essa abordagem é muito mais leve por não haver uma abstração tão complexa quanto na virtualização. E a visibilidade de recursos e os identificadores diferenciados são o que criam o isolamento e a abstração.

### 3.6.2 *Docker*

O *Docker* é uma ferramenta criada em 2013 que desenvolveu a implementação dos chamados "*containers*". Ele funciona como um gerenciador deles, isto é, aplicações que rodam de forma isolada no sistema no qual o *Docker* está instalado. Esse esquema é completamente baseado no *kernel Linux* e na capacidade de segregar os processos e os recursos do sistema de forma que a comunicação entre os *containers* seja isolada e controlada pelo *containerd*, que é uma ferramenta que os constroi, gerencia *namespace*, *cgroups*, rede, sistemas de arquivos etc. É utilizada para criar uma camada de abstração, evitando a necessidade de lidar com a complexidade dos *containers Linux* nativos. A escolha de utilizar o *Docker* é dada pela questão de portabilidade, já que as imagens da infraestrutura dele contêm todo o necessário para que uma aplicação rode sem necessidade de instalação

de bibliotecas e outros artefatos. Ademais, a segurança criada com os isolamentos é um fator também crucial, pois as vulnerabilidades da aplicação e os ataques causados a ela ficam restritos ao ambiente do *container*.

Ele surgiu com a base inicialmente sendo o *LXC* (*Linux Containers*) (Linux Containers, 2025) para gerenciar *containers*. O maior objetivo do *Docker* era facilitar a portabilidade e a escalabilidade de aplicações e do desenvolvimento delas. Através das imagens *Docker* criadas e padronizadas mais tarde (Docker Inc., 2025g), os serviços que constituem uma aplicação têm suas dependências e arquivos todos empacotados.

As imagens *Docker* são um grande diferencial do uso da ferramenta. Elas possuem camadas. Cada **camada** é dada por uma instrução na criação da respectiva imagem, através dos chamados *Dockerfiles* (Docker Inc., 2025b). As camadas são capturas dos arquivos em um certo momento. Em tempo de execução, o *Docker* une essas camadas para um sistema de arquivo que pode ser utilizado pelas *containers*. A existência desse mecanismo é extremamente útil por vários motivos:

a) Reusabilidade de camadas, evitando redundância (Docker Inc., 2025e);

b) Facilitação na construção e atualização de novas imagens, pois camadas acima de uma podem ser reutilizadas pelo cache, se elas não foram modificadas (Docker Inc., 2025e);

c) Facilitar a criação de novas imagens a partir de containers existentes. É possível salvar modificações de um container para uma nova imagem. Para isso, o Docker cria uma nova camada com essas modificações (Docker Inc., 2025e).

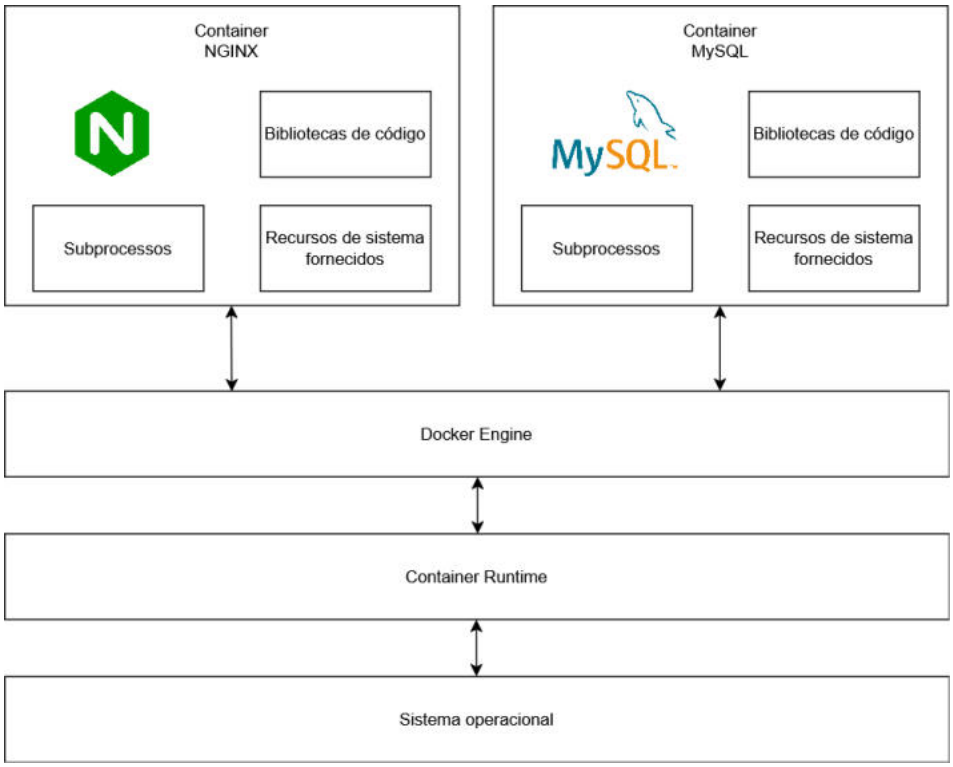
Aproveitando as vantagens desse uso de camadas e uma grande revolução no uso de *containers* para desenvolvimento de aplicações, o uso de repositórios de imagens *Docker* foi desenvolvido no decorrer do tempo. O compartilhamento de imagens e o versionamento delas permite que desenvolvedores criem imagens e um histórico delas. Por exemplo, uma imagem utiliza novos arquivos para um serviço. Através de *tags* (etiquetas) nos nomes das imagens, é possível recriá-la e disponibilizá-la com uma nova etiqueta.

Portanto, como podemos concluir, o isolamento dado por essa abordagem é um grande ponto em quesito de segurança (Docker Inc., 2025a). E também a portabilidade (Docker Inc., 2025f) e facilidade em organizar diversas aplicações em um mesmo servidor, bem como automatizar testes e disponibilidade para produção de novas versões de aplicações (WEISS, 2025).

A figura 9 contém um diagrama de *containers* em *Docker* e a interação com o sistema operacional. Cada *container* roda uma imagem específica contendo as bibliotecas necessárias para execução, juntamente com recursos de sistema fornecidos de forma isolada através de *namespaces* e limites de uso dados por *cgroups*. Outros subprocessos podem ser executados dentro do *container* juntos com os principais (*NGINX* e *MySQL*). O motor do *Docker* abstrai as informações complexas geridas pelo ambiente de execução de *container*

em nível do sistema operacional. Qualquer interação com eles deve ser feita através dos comandos do *Docker*, o que facilita para o usuário/desenvolvedor.

Figura 9 – *Containers* de *NGINX* e *MySQL* geridos pelo *Docker*




Fonte: Elaboração própria

# 4 SOLUÇÃO *CKAN* PARA O PROJETO PRESENÇAS

Neste capítulo, veremos a estrutura da solução para o *CKAN*, isto é, como configuramos os recursos oferecidos por ele em relação ao que necessitamos e também a instanciação do mesmo através do *Docker*. Também veremos como organizamos os metadados no padrão de metadados e as extensões dele.

## 4.1 CONCEPÇÃO DA SOLUÇÃO

O *CKAN* que é utilizado nesse projeto possui algumas modificações em relação ao original. As seguintes extensões estão presentes nesta instalação:

a) ***dcat***: Implementa o vocabulário *DCAT* e todas as operações que o envolvem.

b) ***officedocs***: Permite pré-visualizar, na própria plataforma, documentos do pacote *Office*.

c) ***pdfview***: Assim como o *officedocs*, mas para PDFs.

d) ***scheming***: Permite modificar os esquemas de metadados dos *datasets* e recursos.

e) ***envvars***: Permite utilizar váriaveis de ambiente de um sistema *Linux* para configurar o *CKAN*.

Além delas, existe a extensão de autoria própria (chamada "presencas") para este projeto que será discutida no capítulo 6. Nessa instalação, é utilizado o *DataStore*, descrito no capítulo 3. O uso é necessário para pré-visualização das imagens, já que qualquer extensão que precise acessar recursos só pode fazer tal acesso por meio do *DataStore*.

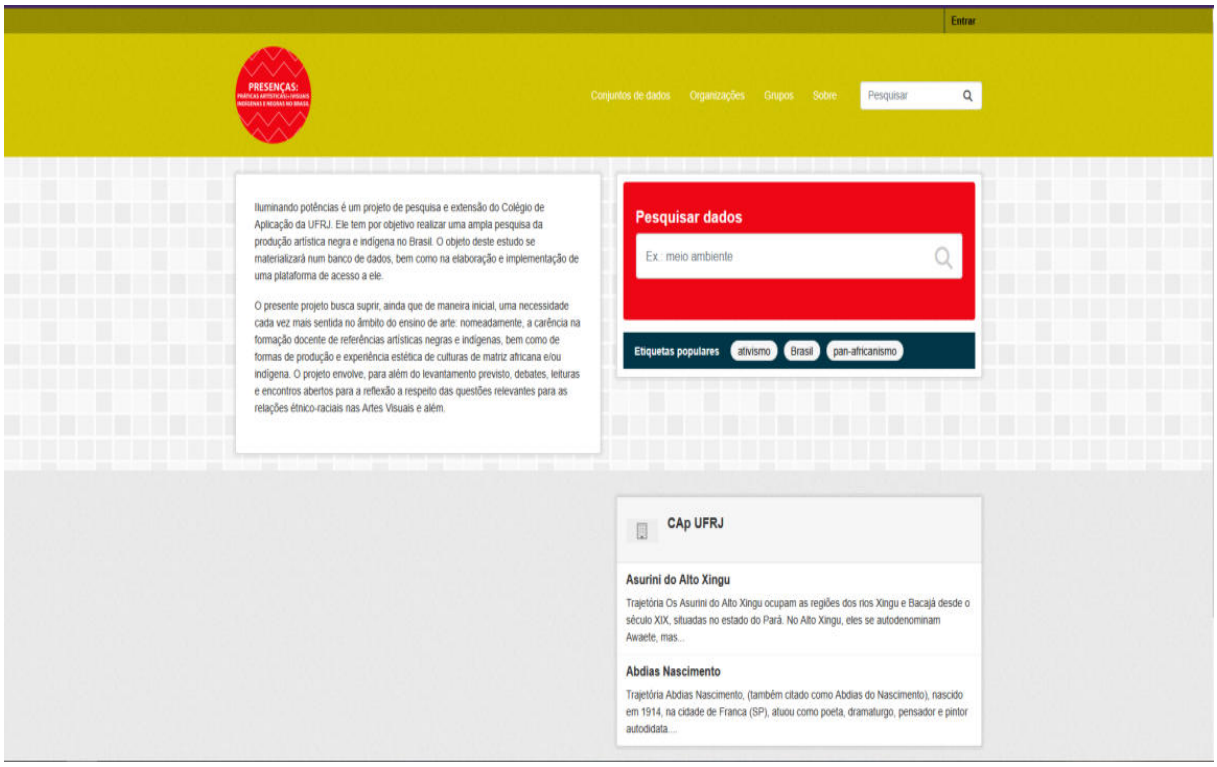
Existe apenas um usuário administrador cujo nome é baseado na coordenadora do Projeto Presenças: "maya_inbar".

Há apenas uma organização, nomeada como "CAp UFRJ". Todos os *datasets* pertencem a ela. A estilização das cores das páginas foi modificada diretamente na opção fornecida no painel administrativo pela plataforma. O código pode ser encontrado no apêndice B. A figura 10 contém a tela inicial do *CKAN* do projeto.

A respeito dos dados na plataforma, cada *dataset* do *CKAN* é um artista e cada recurso em cada *dataset* é uma das imagens dos artistas. Dessa forma, teremos diversos conjuntos de dados nomeados com os artistas e dentro de cada conjunto as obras deles.

Com a extensão "*scheming*", foi possível alterar a esquematização dos datasets e recursos para acomodar os metadados do Projeto Presenças descritos no capítulo 2. Esse processo será melhor explicitado no capítulo 5. Esses campos serão tratados na seção seguinte que fala dos metadados e os mapeamentos deles no *CKAN*.

Figura 10 – Tela inicial do *CKAN* do projeto




Fonte: Elaboração própria

## 4.2 ESTENDENDO O *DCAT* E MAPEANDO OS METADADOS

Nas subseções seguintes será discutida uma extensão do *DCAT* que acomode praticamente todos os metadados do Projeto Presenças semanticamente. Isto é, outros vocabulários e outras ontologias serão adicionadas ao vocabulário *DCAT* para tal objetivo. Também será exibido o mapeamento final dos metadados no esquema dos *datasets* e recursos para este vocabulário estendido.

### 4.2.1 Campos padrões da extensão *DCAT*

A extensão do *DCAT* para o *CKAN* já apresenta alguns mapeamentos por padrão (Open Knowledge Foundation, s.d.). Apenas as classes *Dataset* e *Distribution* do vocabulário *DCAT* foram usadas. Todos os mapeamentos padrões da extensão foram utilizados. Porém, alguns deles foram removidos através do perfil criado para este projeto.

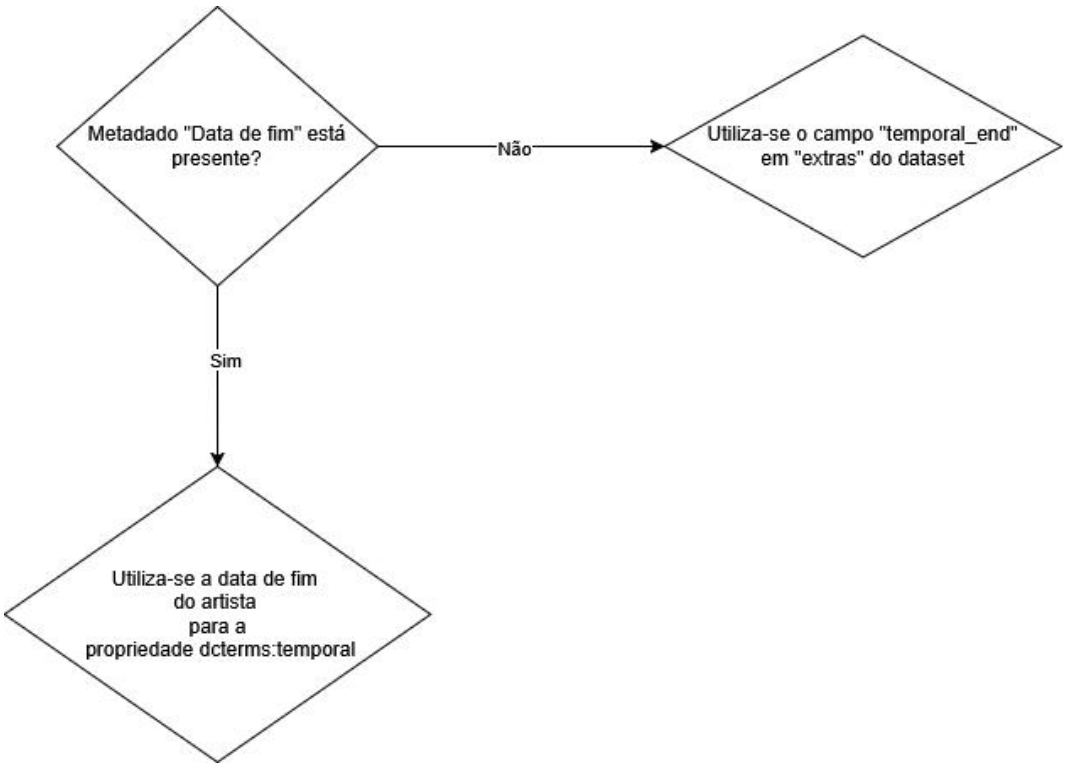
O campo *temporal_end* (se presente, já que também é um valor inserido) na seção *extras* dos *datasets* é ignorado no mapeamento para a propriedade *dcterms:temporal* da classe *Dataset* do *DCAT*. Essa decisão foi tomada porque utilizaremos o metadado "Data de fim" do Projeto Presenças para armazenar tal informação.

De forma análoga, o campo *issued* dos recursos dos *datasets* não é mapeado para *dcterms:issued* da classe *Distribution* do vocabulário, pois a data de criação do recurso será

dada pela data da obra no mundo real, isto é, o metadado "*Ano*" do Projeto Presenças.

No entanto, esses dois campos originais da extensão citados só são ignorados se os metadados estiverem presentes na criação do *dataset*/recurso. Caso contrário, o mapeamento padrão da extensão é assumido. Portanto, daqui em diante, assuma que o padrão da extensão *DCAT* é utilizado. Isto é, todos os campos nomeados no manual são mapeados da forma que são expostos lá. A figura 11 e a figura 12 mostram a ideia descrita acima.

Figura 11 – Mapeamento da propriedade *dcterms:temporal* do *DCAT*



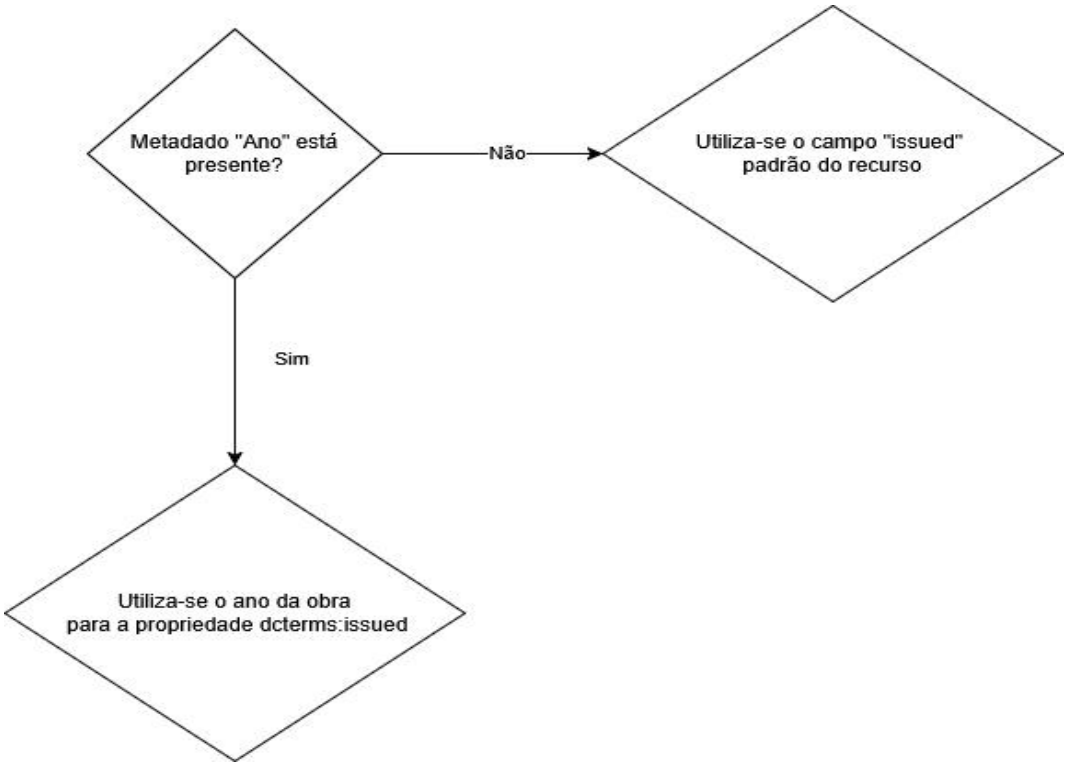

Fonte: Elaboração própria

### 4.2.2 Estendendo o *DCAT* com *VCARD*, *GeoSPARQL*, *QUDT* e *VRA*

O vocabulário *DCAT* não consegue acomodar todos os metadados que necessitamos em nosso cenário. Por exemplo, dado um artista individual com seu gênero, não existe um campo que possa armazenar esse metadado no *DCAT*. Logo, tomando esse campo como exemplo e outros como "comprimento" e "área", faz-se necessário o uso de outras ontologias e outros vocabulários.

Como os *datasets* são os artistas, podemos agregar a ontologia *VCARD* à classe *Dataset*. Essa é uma boa decisão baseada nos fatos de que ela é feita para descrever pessoas, organizações e grupos, possui anos de desenvolvimento e é constantemente atualizada e baseada na RFC 6350 (PERREAULT, 2011) e possui classes e propriedades excelentes

Figura 12 – Mapeamento da propriedade *dcterms:issued* do *DCAT*




Fonte: Elaboração própria

para descrever os artistas do projeto, dentre elas: *Gender* (Gênero), *Locality* (Cidade), *Region* (Estado), *Address* (Cidades, Estados e Países de atuação) e *bday* (Aniversário/Data de início) (ANNELLA; MCKINNEY, 2024). A preferência por utilizar essa ontologia ao invés do vocabulário *FOAF* se justifica porque este tem foco nas relações das pessoas com documentos e outros itens, mais especificamente da *web* (BRICKLEY; MILLER, 2014). Além disso, como temos indivíduos ou grupos/coletivos, podemos utilizar a classe *Individual* para aquele e *Group* para esse. Um mapeamento detalhado será dado na próxima seção.

Já para as obras, temos imagens que as representam no mundo real, sendo elas esculturas, pinturas ou retratos. Esculturas e pinturas possuem áreas e comprimentos, respectivamente. Assim como para os metadados dos artistas, não há uma forma de abrigar dados dimensionais dessas obras. Consequentemente, uma extensão ao perfil padrão do *DCAT* deve ser aplicada. O padrão até apresenta o uso da ontologia que será citada a seguir, mas é aplicada para localizações e coordenadas geográficas (propriedades *locn:geometry*, *dcat:bbox* e *dcat:centroid*) (ALBERTONI et al., 2024). Seguindo o raciocínio dos *datasets*, podemos utilizar a ontologia *GeoSPARQL*. Ela é uma ontologia que permite representar geometrias associadas com características espaciais (CAR et al., 2024). Ou seja, permite que os metadados de área e comprimento possam ser representados. Para tal objetivo, usaremos a classe *SpatialObject*, cuja definição é dada por "Qualquer coisa espacial (tendo

forma, posição ou uma dimensão)." (CAR et al., 2024). Para o comprimento, utilizaremos as propriedades *hasMetricLength* e *hasLength*. Essa decisão foi tomada porque, para fins do código que realiza a geração de metadados, existe a possibilidade de não conseguirmos converter o valor dado nos documentos dos artistas para o range dado em *hasMetricLength*. Quando isso não é possível, a ontologia recomenda o uso de *hasLength*. Já para a área, os documentos estão com um cálculo bidimensional, isto é, no formato base × altura e com unidade geralmente em centímetros. Sendo assim, não há uma representação numérica final em metros como solicita a propriedade *hasMetricArea*. Poderíamos, sim, através do código, calcular o valor conforme a expressão anterior. No entanto, isso é passível de erros, pois temos que checar a entrada do usuário, o que é um trabalho adicional. Logo, é mais fácil utilizar a propriedade *hasArea*. Agora, um adendo importante. A ontologia, em sua seção 10.3, sugere a indicação das unidades de medidas para maior corretude. Para isso, ela recomenda o uso da ontologia *QUDT* (*Quantities, Units, Dimensions and Types*) (MEKONNEN et al., 2024). Portanto, para indicar os valores numéricos para *hasArea* e *hasLength*, utilizaremos a propriedade *qudt:value*, já que a descrição da medida é textual com a unidade dela no próprio literal do nó *RDF*.

Para o campo "*tecnica*" das obras, utilizaremos o já citado vocabulário *VRA* (Library of Congress, 2022). O *VRA*, que possui quatro versões, tem como objetivo descrever, através de um esquema *RDF*, trabalhos culturais e artísticos, bem como as imagens que os representam. Ou seja, exatamente o que possuímos no acervo do Projeto Presenças. Esse vocabulário por si só é um padrão de metadados, isto é, as classes e propriedades nele são autossuficientes para descrever dados. Porém, como já temos o *DCAT* atuando no nível principal da hierarquia para organizar os datasets, iremos incluir as classes e propriedades do *VRA* dentro de instâncias da classe *dcat:Distribution*, que é a classe que representa os recursos dos datasets. O *VRA* possui classes que resolvem problemas anteriores, como o caso das medidas das obras que podem ter os metadados abrigados pela classe *vra:Measurements* e o caso do metadado de "Linguagens", que remete a idiomas e temas trabalhados por um artista e que foi mapeado na seção seguinte para um campo na ontologia VCARD. Não obstante, devido à descoberta tardia do *VRA* para a construção do repositório, preferiu-se manter os mapeamentos e apenas utilizar o *VRA* para o metadado "*tecnica*". Logo, para abrigar esse metadado em nosso repositório, mapeamos "*tecnica*" para um elemento da classe *vra:Technique*. Essa classe representa as técnicas de uma produção ou manufatura ou método na fabricação ou alteração de um trabalho ou imagem (Library of Congress, 2007).

Portanto, condiz com o metadado, já que este descreve o material da obra do artista. Porém, todas as classes do padrão *VRA* devem estar dentro de uma instância de *vra:Work*, *vra:Image* ou *vra:Collection* de acordo com a documentação. Como temos obras de artistas, iremos incorporar a técnica dentro de *vra:Work*, que, por sua vez, estará incorporada na classe *dcat:Distribution*. E, como permite o padrão, iremos armazenar o metadado

de fato dentro da propriedade *vra:display*, que será propriedade de *vra:technique*. Isso é análogo ao uso de *qudt:value* na indicação dos valores numéricos de área e comprimento que descrevemos. Temos imagens representando obras, logo faz sentido o uso da classe *vra:Image*. Mas, iremos considerar que todas as imagens são obras dos artistas, seja esta imagem uma fotografia considerada como obra, no caso de um fotógrafo, ou sendo a fotografia de um objeto como pintura ou escultura. Essa decisão torna o uso da classe *vra:Technique* mais associado à obra em si, e não à técnica da imagem. Por isso, justifica-se o uso da classe *vra:Work*, que representa um objeto criado ou construído.

É importante lembrar que estamos sendo flexíveis com o esquema *RDF* aqui. Sintaticamente e semanticamente, o correto é criar uma nova propriedade e definir o domínio e *range* desta propriedade para as classes que estamos utilizando para a extensão. Porém, não estamos sendo formais a esse ponto. Isto é, estamos incorporando as classes dentro das classes do *DCAT* da forma em que elas estão declaradas nas ontologias/vocabulários de origem, assumindo, que estas classes adicionadas são homônimas ao serem inseridas numa classe do *DCAT*. Como exemplo, temos a classe *vra:Work*. Ao inseri-la no *DCAT*, ela será uma propriedade *vra:work* com domínio *dcat:Distribution* e *range vra:Work*.

O diagrama da classe *Dataset* do padrão *DCAT* com as agregações ficou conforme a figura 13. Já a classe *Distribution* ficou conforme a figura 14. Em ambos os diagramas, os três pontos representam a inclusão dos campos originais do padrão. Os *ranges* das propriedades na figura 13 e na figura 14 seguem os mesmos conforme as ontologias.

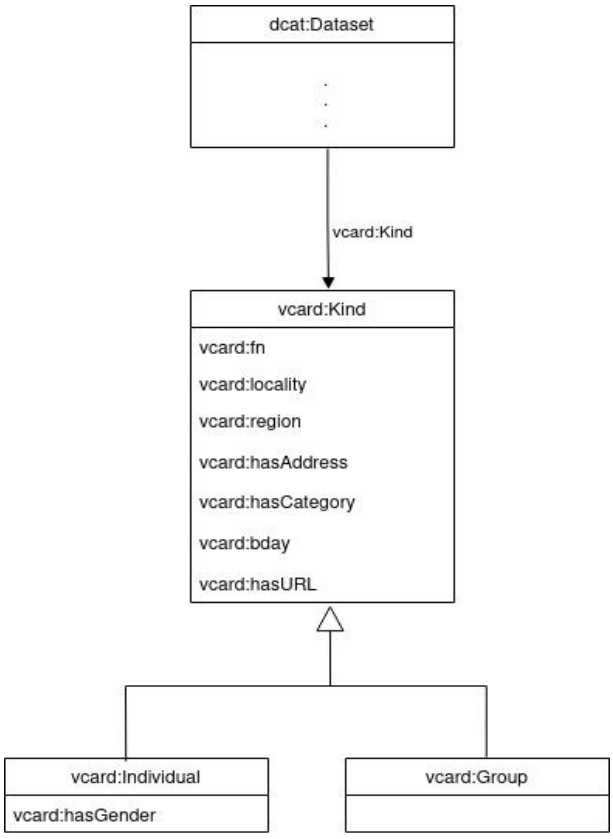
Adicionalmente, devemos citar algumas restrições cuja linguagem *UML* não permite adicionar nesses diagramas. Uma instância de *Distribution* só pode ter uma das propriedades, isto é: ou *hasArea*, ou *hasMetricLength* ou *hasLength*. Ou seja, não representamos área e comprimento ao mesmo tempo e, para o comprimento, ou temos um valor numérico ou um literal com a medida e a unidade de medida. Segundo, a propriedade *hasAddress* para o *vcard:Kind*, para o caso de listas de cidades, estados e países de atuação, terá as propriedades *location*, *region*, *country-name* como *strings* concatenadas com ponto e vírgula. Um exemplo está no código 1.

Código 1 – Propriedade *hasAddress* com propriedades n-árias

```
<vcard:hasAddress>
    <rdf:Description rdf:nodeID="Nf6d9dfc5218f46439161797e003e8b5c">
        <vcard:locality>Rio De Janeiro; Paris</vcard:locality>
        <vcard:region>Rio de Janeiro; Nova Iorque</vcard:region>
        <vcard:country-name>Brasil; EUA</vcard:country-name>
    </rdf:Description>
</vcard:hasAddress>
```

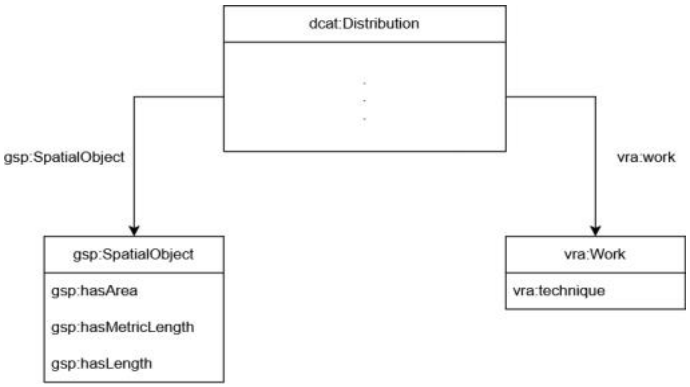
Essa decisão da concatenação dos metadados foi tomada devido à forma que recebemos os dados através da aplicação mencionada no capítulo 5 (e também através do formulário padrão de cadastro do *CKAN* caso ele seja utilizado). Precisaríamos combinar a cidade,

Figura 13 – Classe *Dataset* estendida com *VCard*




Fonte: Elaboração própria

Figura 14 – Classe *Distribution* estendida com *GeoSPARQL* e *VRA*




Fonte: Elaboração própria

estado e país de atuação para criar uma instância de *hasAddress*, permitindo várias instâncias no caso de vários estados, países e cidades de atuação. Porém, considerando um erro do usuário ao fornecer o metadado, poderíamos ter uma dupla como "Rio de Janeiro, Estados Unidos". Como não é possível fazer uma fácil validação, preferiu-se separar os três itens através dessa abordagem.

Com isso, criamos um perfil *DCAT* específico para o Projeto Presenças, capaz de

acomodar quase todos os metadados disponíveis. A seção seguinte possui dois quadros com os mapeamentos dos metadados da seção 2.2. Cabe ratificar que esse perfil criado respeita as regras da seção 4 do documento oficial do DCAT (ALBERTONI et al., 2024).

### 4.2.3 Mapeamento dos campos presentes no projeto

Os campos "trajetória" e "produção" foram unidos em um só para criar a descrição do *dataset*. Os campos "data da pesquisa", "revisor/a", "status" e "data revisão" do Projeto foram ignorados ao carregar os artistas na plataforma, pois não possuem significados úteis para armazenamento no *CKAN* e para fins públicos de exposição dos artistas e obras. Eles devem ser mantidos externamente, pois o *CKAN* não permite que metadados específicos sejam ocultos para o público como no caso do *Dataverse* (The Dataverse Project, 2025a). No *CKAN*, é possível apenas tornar pública ou privada a visibilidade de um *dataset* por completo.

Nos quadros 1, 2 e 3 estão os mapeamentos finais, desde os nomes dos metadados originais no projeto até os mapeamentos na extensão do *DCAT*. A primeira coluna representa a nomenclatura do campo no Projeto Presenças. A segunda coluna é o campo representando o nome dado no *CKAN* para que o usuário o visualize ao procurar um artista ou obra. A terceira coluna é o nome do campo ao nível do *CKAN*, ou seja, como é armazenado no *Solr*. E a quarta coluna representa o mapeamento em nível de vocabulário/ontologia. Para campos que são propriedades de objetos, foi utilizado o símbolo $\rightarrow$ para representar a hierarquia conforme determinada na ontologia/no vocabulário. Lembrando que apenas os metadados do Projeto Presenças estão citados aqui. Os demais seguem o padrão do *CKAN* e da extensão.

## 4.3 CONFIGURAÇÃO DO CKAN E ESTRUTURA GERAL DA SOLUÇÃO

O *CKAN* utilizado neste projeto é baseado no *Github* oficial dos desenvolvedores[1]. Um repositório com o *CKAN* deste projeto com todas as configurações executadas e pronto para instanciação pode ser encontrado aqui[2]. Estamos utilizando a versão em *container* de *Docker* pelo que foi exposto anteriormente no capítulo 3. A versão em questão do *CKAN* é a 2.11.0. Por termos o objetivo de utilizar um *proxy* geral para um *cluster* que será descrito no capítulo 7, não será usado o *NGINX* oferecido na receita de construção de serviços. O *DataPusher* utilizado não é o recomendado também. A imagem que utilizamos pode ser encontrada aqui[3]. A versão utilizada é a 0.15.0. Este trabalho não tem como objetivo configurar o *CKAN* ou mostrar exatamente o que deve ser feito. Apenas descreveremos brevemente como foi feita a configuração para esse projeto, já que

[1] https://github.com/ckan/ckan-docker
[2] https://github.com/thiagoc01/presencas-ckan-tcc
[3] https://github.com/dathere/datapusher-plus/tree/0.15.0

o manual de referência do usuário já faz esse trabalho (Open Knowledge Foundation, 2025e).

### 4.3.1 Estrutura da solução em *Docker*

A figura 15 contém o diagrama da solução *CKAN* descrita neste capítulo em *Docker*. Perceba que o diagrama da figura dos componentes do *CKAN* foi modificado com a adição da rede e das requisições *HTTP*. A rede "*ckan-marilu*" é uma rede virtual gerenciada pelo *Docker* em estilo *Bridge* (Docker Inc., 2025c), ou seja, realiza um *NAT* (*Network Address Translation*) através de um roteador virtual criado para isolar os *containers* da rede do hospedeiro, porém torná-los comunicativos entre eles, ou seja, qualquer um dos serviços consegue acessar o outro na rede. Requisições externas não podem conectar-se a esses *containers*. Somente a porta 5000 do *container* do *CKAN* é exposta para o hospedeiro, já que é preciso que a aplicação seja acessível externamente.

Essa *stack* de serviços em *Docker* irá conter todos os componentes do *CKAN*, inclusive o banco de dados como vemos na figura 15.

### 4.3.2 Instalação *CKAN* em *Docker*

Após clonar o repositório, construímos a imagem Docker do CKAN com o seguinte comando:

docker build –build-arg TZ=America/Sao_Paulo -t ckan:2.11.0 .

Teremos nas imagens locais uma com o nome ***ckan:2.11.0***. Em seguida, criamos a rede de formato *bridge* que será utilizada pelos *containers*:

docker network create ckan-marilu

Para persistir os dados já que os *containers* de *Docker* são efêmeros, criamos a pasta *persistencia*. Dentro desta pasta criada, criamos as pastas:

a) ***banco_dados***: Irá possuir os arquivos do *container* de banco de dados;

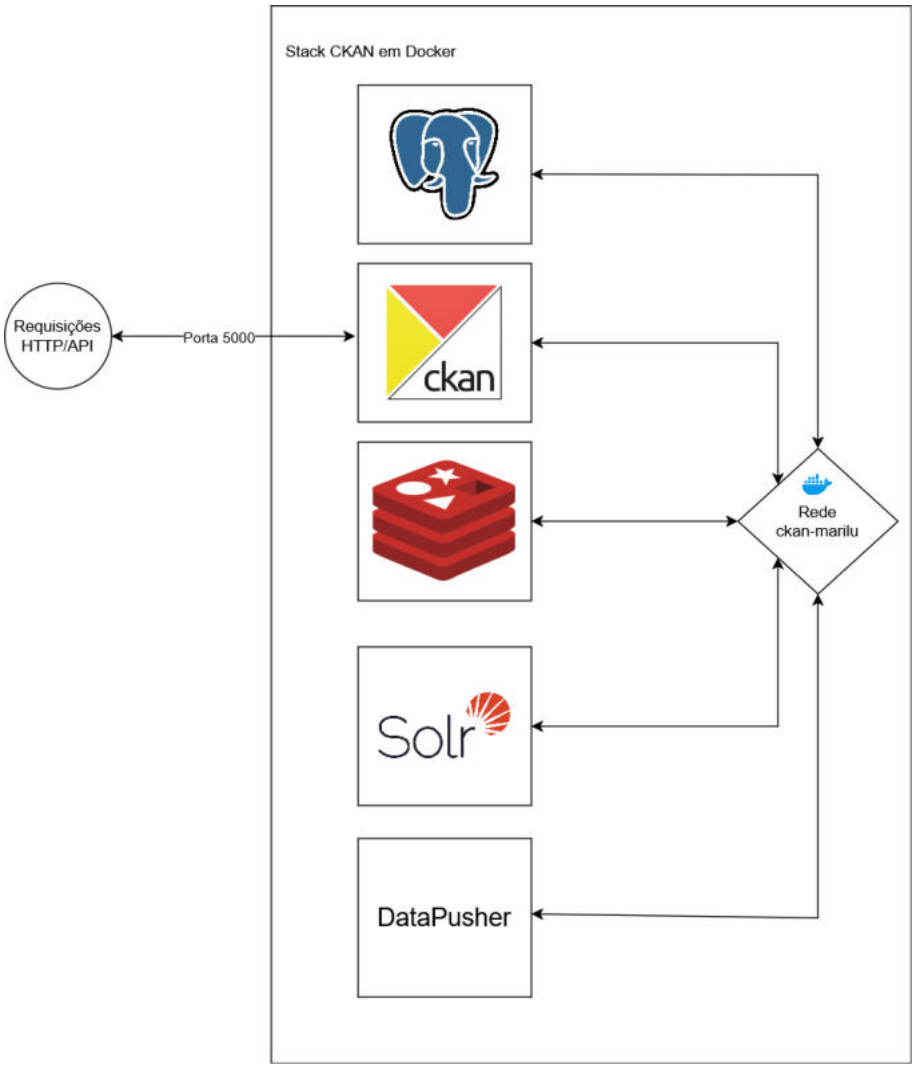
b) ***bibliotecas-python***: Irá possuir os módulos do *Python* e as extensões instaladas para o *CKAN*;

c) ***dados***: Contém todos os arquivos do *CKAN* como as imagens enviadas, *cache* de páginas e outros arquivos enviados para a configuração do *CKAN*;

d) ***datapusher***: Contém apenas o arquivo que possui as variáveis de ambiente do *container* configuradas conforme o *Github* dado anteriormente;

e) ***solr***: Contém os arquivos de configuração do *Apache Solr*, indexador utilizado pelo *CKAN* que falaremos mais à frente.

Figura 15 – Diagrama da solução *CKAN* em *Docker*




Fonte: Elaboração própria

O *compose* utilizado para o projeto pode ser encontrado no apêndice C. Portanto, estaremos supondo que o arquivo *docker-compose.yml* possui o conteúdo dado no apêndice. E o arquivo de variáveis de ambiente para o *CKAN* está no apêndice D. Por fins de segurança, itens sensíveis não tiveram os valores inseridos. Deve-se alterar alguns valores conforme os endereços IPs do hospedeiro em questão. Novamente, não é o objetivo deste trabalho indicar como configurar e instanciar de forma completa a infraestrutura de um repositório *CKAN*.

Antes de iniciarmos completamente a *stack* de serviços, primeiro devemos configurar o banco de dados para o *DataPusher*. Inicializamos apenas o banco de dados.

```
docker compose up -d db
```

Com isso, acessamos o *container*:

```
docker compose exec db bash
```

E executamos os comandos para o banco de dados presentes no repositório do *DataPusher* na seção de configuração.

É de extrema importância que todos os usuários no banco de dados, tanto para o *CKAN* quanto para o *DataPusher* sejam superadministradores, pois algumas funções do *PostgreSQL* que envolvem permissões de escrita de arquivos são usadas, tabelas são criadas etc. Por ser uma permissividade de grande porte, é recomendado que o banco de dados esteja completamente isolado e independente de outros. No nosso caso, isso acontece porque o banco é um dos serviços da *stack*.

Por fim, com as permissões ajustadas é possível inicializar o *CKAN* e os demais serviços.

```
docker compose up -d
```

Ele ficará disponível na porta 5000 da máquina.

Essa instalação já vem com o perfil do *DCAT* configurado para o Projeto Presenças da forma como explicado na seção 4.2. O arquivo com o código de autoria própria do perfil descrito pode ser encontrado na pasta *profiles* da extensão. Esse código é capaz de criar um *dataset* através de um esquema *RDF* e também é capaz de gerar um esquema *RDF* a partir dos *datasets* da plataforma conforme a documentação da extensão (Open Knowledge Foundation, 2025g).

Quadro 1 – Nomes dos metadados dos datasets no *CKAN* e seus mapeamentos para os vocabulários

| **Nome original** | **Nome no CKAN** | **Nome do campo** | **Mapeamento** |
|---|---|---|---|
| Nome | Título | title | dcterms:title |
| Traj./Prod. | Descrição | description | dcterms:description |
| Referências | Links | links | vcard:hasURL |
| Última atualização | Última atualização | modified | dcterms:modified |
| Pesquisante | Mantenedor | maintainer | dcat:contactPoint → vcard:Organization → vcard:fn |
| E-mail pesquisante | E-Mail do Mantenedor | maintainer_email | dcat:contactPoint → vcard:Organization → vcard:hasEmail |
| Indivíduo, coletivo, povo | Indivíduo, coletivo, povo | quantidade | vcard:Individual ou vcard:Group |
| Data início/nasc | Data de início | data_inicio | vcard:Kind → vcard:bday |
| Data fim/Falecimento | Data de fim | data_fim | dcterms:temporal → dcterms:PeriodOfTime → dcat:endDate |
| Linguagens | Linguagens | linguagens | vcard:hasCategory → vcard:value |
| Cidade de origem | Cidade de origem | cidade | vcard:Kind → vcard:locality |
| Estado de origem | Estado de origem | estado | vcard:Kind → vcard:region |
| Cidade de atuação | Cidade de atuação | cidade_atuacao | vcard:Kind → vcard:hasAddress → vcard:locality |
| Estado de atuação | Estado de atuação | estado_atuacao | vcard:Kind → vcard:hasAddress → vcard:region |
| País de atuação | País de atuação | pais_atuacao | vcard:Kind → vcard:hasAddress → vcard:country-name |

Fonte: Elaboração própria

Quadro 2 – Nomes dos metadados dos datasets no CKAN e seus mapeamentos para os vocabulários (continuação)

| **Nome original** | **Nome no CKAN** | **Nome do campo** | **Mapeamento** |
|---|---|---|---|
| País de atuação | País de atuação | pais_atuacao | vcard:Kind → vcard:hasAddress → vcard:country-name |
| Palavras-chave | Etiquetas | tags | dcat:keyword |
| Gênero | Gênero | genero | vcard:Kind → vcard:hasGender → vcard:value |
| Link p/site | Fonte | url | dcat:landingPage → foaf:Document |

Fonte: Elaboração própria

Quadro 3 – Nomes dos metadados dos recursos no CKAN e seus mapeamentos para os vocabulários

| **Nome original** | **Nome no CKAN** | **Nome do campo** | **Mapeamento** |
|---|---|---|---|
| Título | Nome | name | dcterms:title |
| Fonte | Fonte | fonte | foaf:page |
| Descrição | Descrição | description | dcterms:description |
| Comprimento | Comprimento | comprimento | gsp:SpatialObject → gsp:hasMetricLength ou gsp:SpatialObject → gsp:hasLength → qudt:value |
| Área | Área | area | gsp:SpatialObject → gsp:hasArea → qudt:value |
| Ano | Data da criação da obra | data_criacao | dcterms:issued |
| Técnica | Técnica | tecnica | vra:Work → vra:technique → vra:display |

Fonte: Elaboração própria

# 5 UMA APLICAÇÃO EXTERNA QUE INTERAGE COM A *API* DO *CKAN* PARA CADASTROS/ATUALIZAÇÕES

Neste capítulo iremos abordar o desenvolvimento de uma aplicação simples, externa ao *CKAN*, que servirá para cadastro de artistas e obras. Ela foi criada por duas razões basicamente: o controle de novos campos e criações de formulários mais simplificados e uma revisão de alguns conteúdos estudados no curso para o qual este trabalho é apresentado como conclusão. A aplicação desenvolvida está em um repositório *Github*[1].

## 5.1 MOTIVAÇÃO DE EXISTÊNCIA E CONDIÇÕES

O *CKAN* possui uma extensão chamada *scheming*[2] . Essa extensão permite modificar a estrutura de metadados de *datasets* e recursos. Existem alguns padrões de tipos como textos, múltiplos textos, listas de escolha, datas e outros tipos que já existem no próprio *CKAN*, como tipos que conectam-se com dados de um *dataset*. Caso necessário, é possível criar novos tipos primários, bem como alterar e criar formulários de cadastros destes tipos de metadados.

A existência dessa extensão não era conhecida até um certo ponto do início deste projeto. Acreditava-se que era possível apenas adicionar metadados para *datasets* via a seção "*extras*" e para recursos através do uso da *API* do *CKAN*, que permite enviar campos via um objeto *JSON*. Ademais, como já mencionado, este trabalho é uma extensão de outro de uma disciplina do curso. Uma ideia sugerida por uma professora com base em um *script* automatizador que interagia com a *API* já havia sido apresentada à orientadora. Com a aprovação desta ideia, foi dado início ao desenvolvimento da aplicação externa. Ao fim do desenvolvimento, ocorreu a descoberta da extensão. No entanto, ela foi utilizada em conjunto com a aplicação externa. Ela permite que os campos de metadados não sejam mais "*extras*". Isso facilita na manipulação dos metadados para a extensão que será escrita por nós que será abordada no capítulo 6. Além de, claro, ter uma melhor exibição para o usuário final nas páginas de exibição. Logo, a necessidade de cadastrar artistas e obras de forma acessível e simples para a equipe gestora do Projeto Presenças motivou a criação de uma aplicação externa.

Esta aplicação é uma aplicação *web* escrita em *Flask*, o mesmo *framework* do *CKAN*. O desenvolvimento dela auxiliou no entendimento das extensões do *CKAN* e de como funciona seu código-fonte, já que o próprio é uma aplicação *Flask* como já mencionado. Também, trabalhar com esse *framework* é muito simples e ele possui uma documentação bem acessível, bem como outros componentes que auxiliaram como módulos de sessão, senhas e formulários. Comparando-se a outro *framework*, o *Django*, a sua complexidade

[1] https://github.com/thiagoc01/presencas-tcc
[2] https://github.com/ckan/ckanext-scheming

é muito menor tanto na curva de aprendizado quanto de criação e manutenção de código. O uso da linguagem *Python* foi dado também pela facilidade, assim como pelo fato de também ser a linguagem em que o *script* automatizador do trabalho da disciplina anterior ter sido escrito em *Python*. Novamente, a escrita da aplicação em *Python* facilitou o entendimento de como extensões são criadas no *CKAN*, o que auxiliou para a extensão criada para o projeto por nós.

Adicionalmente, esta aplicação é uma excelente reunião para estudo dos seguintes assuntos do curso:

- **Protocolo *HTTP***: Por ser uma aplicação *web*, a interação navegador e servidor é a interação cliente e servidor da *Internet*. No código-fonte, as respostas enviadas ao usuário e as respostas recebidas da *API* do *CKAN* são inteiramente baseadas no código *HTTP* obtido, isto é, sucesso ou erros.

- **Banco de dados**: Como esta aplicação estará para a *Internet*, ela é acessível apenas via autenticação. Portanto, foi preciso criar uma tabela de usuários com seus respectivos atributos de administrador, identificação e senha. Além disso, como será visto mais à frente, uma tabela chamada "*solicitacoes*" foi criada para que o usuário obtenha o real progresso da criação de artistas e envio de obras, já que este progresso é uma conexão entre o servidor da aplicação de cadastro e o servidor do *CKAN*, o qual não existe um acesso direto para obtenção de informação.

- **Aplicações *web***: Aplicações modernas são construídas com uma *stack* conhecida como *HTML*, *CSS*, *JavaScript* e alguns frameworks como *JQuery* e *Bootstrap*, além do próprio *Flask* para atuar nos bastidores (*backend*). No caso do banco de dados, foi utilizado o *PostgreSQL*, que é uma implementação de *SQL*.

Outro motivo que impulsionou o uso da aplicação mesmo após a descoberta da existência da extensão *scheming* foi a dificuldade de alterar o comportamento padrão para a criação dos formulários de cadastro. O código-fonte possui uma certa complexidade. E, ainda, existe um *bug* na exibição e armazenamento de listas de metadados. Em resumo, criar novos campos de metadados, controlar os tipos deles e a validação de entrada é consideravelmente mais fácil se comparado ao estender o código-fonte da extensão.

## 5.2 ESTRUTURA DA APLICAÇÃO

### 5.2.1 Como funciona?

A aplicação segue uma estrutura *MVC*. No quesito *Model*, temos os arquivos que modelam os campos dos formulários e as tabelas de solicitações e usuários. Também temos a classe que modela um artista no contexto do Projeto Presenças. No quesito *View*, temos os *templates HTML* e os arquivos *CSS* que controlam a disposição e estilização

das informações nas páginas. No quesito *Controller*, temos como exemplo o arquivo *criador.py* que carrega os dados do usuário no modelo de artista, verifica a validade deles e conecta com o *CKAN* para criá-lo. Todas as funções e métodos que recebem os dados dos formulários também entram nesse quesito, pois ocorrem validações e carregamentos no banco de dados, como ao criar um usuário por exemplo.

Ela consiste de quatros pontos principais. A autenticação para o menu principal, o cadastro de artistas e obras, o cadastro de novos usuários e a alteração de senha. A figura 16 contém a tela de autenticação. Apenas usuários cadastrados podem acessar o *menu* principal (figura 17), que consiste de botões com as opções de cadastrar artistas e, se desejado imediatamente, suas obras. Também é possível inserir apenas obras de artistas já cadastrados no *CKAN* e adicionar ou remover um usuário. Na barra superior existe um botão que retorna para esse *menu* principal e outro que realiza o *logout*. Podemos ver que é uma aplicação simples, porém efetiva.

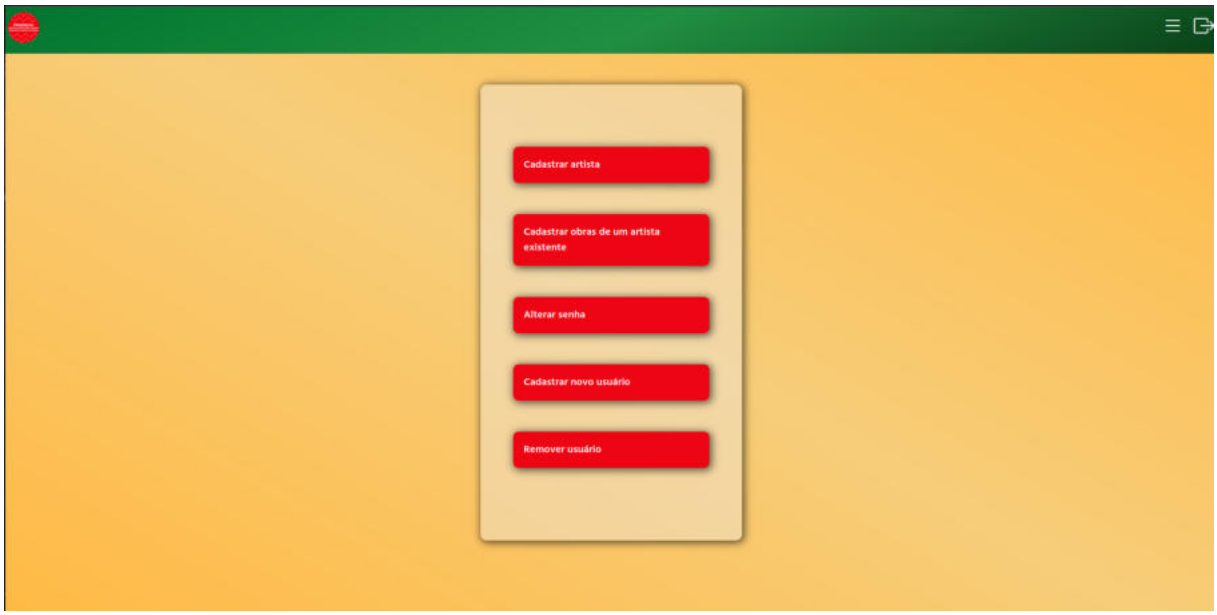
Figura 16 – Tela de autenticação


Fonte: Elaboração Própria

#### 5.2.1.1 Cadastro artistas e obras

Na opção de cadastrar artistas (figuras 18, 19 e 20), existe um formulário com todos os metadados mencionados no capítulo 4. Obrigatoriamente, é necessário apenas inserir o nome, trajetória e produção do artista. É possível adicionar vários *links*, imagens, linguagens, cidades/estados/países de atuação e palavras-chave. Para todo campo com múltiplas inserções, a presença de um deles solicita, obrigatoriamente, um valor a ser inserido.

Figura 17 – Menu principal




Fonte: Elaboração Própria

Figura 18 – Formulário de cadastro de artista


Fonte: Elaboração Própria

O fluxo máximo de um usuário nesta página consiste em preencher todos os campos, solicitar ao menos o envio de uma imagem (somente os formatos JPG, JPEG, PNG e JFIF são aceitos), preencher ao menos uma linguagem, cidade/estado/país de atuação, um *link*, uma palavra-chave e preencher a data da última atualização do artista, que deve ser anterior ou igual ao dia do presente cadastro. Ao clicar em "Criar Artista", uma tela de carregamento será exibida (figura 21). Como dito, esse progresso diz respeito ao

Figura 19 – Formulário de cadastro de artista (continuação)


Fonte: Elaboração Própria

Figura 20 – Formulário de cadastro de artista (continuação)

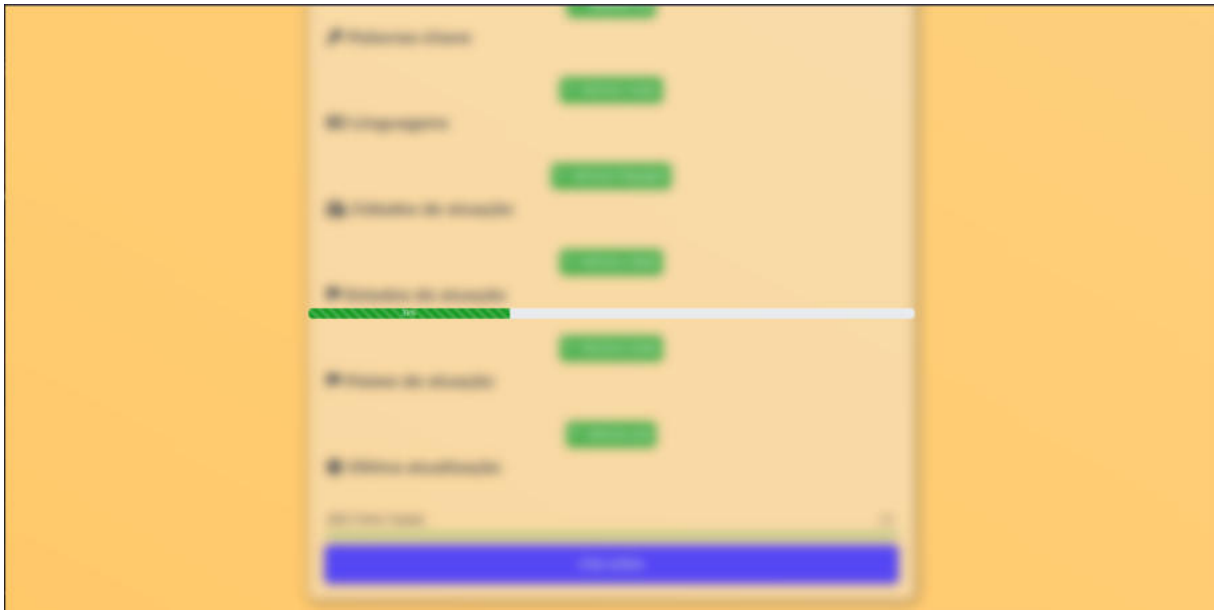

Fonte: Elaboração Própria

envio das informações para o *CKAN*, ou seja, a comunicação da aplicação de cadastro com o *CKAN*, e não ao envio das informações do navegador do cliente ao servidor da aplicação. Se tudo ocorrer bem, será retornada uma página de sucesso com um *JSON* cru de resposta do *CKAN* conforme a figura 22. Caso contrário, se houver um problema entre o servidor e o *CKAN*, uma página de erro será exibida também com um *JSON* cru de resposta. Se o usuário inserir um artista que já existe, não carregar uma imagem

ou existir qualquer dado inválido de formulário (em nível estrutural com manipulação da página já que existem validações no código *HTML* e *JavaScript*), a mesma página é exibida com avisos de erros para cada campo incorreto. No caso da verificação da existência do artista, tanto nessa página quanto na página de adicionar somente obras de um artista já existente, a checagem é feita pelo nome somente, já que o *dataset* é criado no *CKAN* na forma "nome_sobrenome". Para sobrenomes com mais de 2 itens, todos eles são concatenados. Por exemplo, "Thiago Figueiredo Lopes de Castro" gera um nome de *dataset* "thiago_figueiredolopesdecastro".

Para adicionar novas obras para artistas existentes, o fluxo é semelhante. Apenas o nome do artista e ao menos uma imagem são solicitados. A validação do nome e da imagem são iguais ao formulário anterior. Nesse caso, se o artista não está já no *CKAN*, haverá um erro de formulário.

Figura 21 – *Status* da solicitação



Fonte: Elaboração Própria

#### 5.2.1.2 Cadastro e remoção de usuários e alteração de senha

Existem três tipos de usuários na aplicação:

a) **Usuário comum**: Apenas tem a possibilidade de cadastrar artistas e obras e alterar a senha dele. Ou seja, o botão de cadastrar ou remover usuários não é exibido.

b) **Administrador**: Possui a capacidade de adicionar e remover usuários. Apenas administradores ou superadministradores podem criar outros dos seus tipos.

Figura 22 – Página de sucesso ao cadastrar artista


Fonte: Elaboração Própria

c) **Superadministrador**: É como um administrador, porém ele é criado via linha de comando e não pode ser removido pela interface *web*, isto é, pela página de remoção de usuários.

A figura 23 mostra a página de criação de usuário. Um nome deve ser inserido e uma senha respeitando os critérios exibidos. É possível tornar ou não o novo usuário como administrador. Enquanto os critérios da senha não forem atendidos, não é possível criar o usuário. Após inserir os dados corretamente, se o usuário já existir é retornado um erro na própria página. Se tudo estiver correto, será exibida uma página de sucesso no mesmo formato da criação de artistas. Caso haja erro na aplicação, será retornado um erro específico de *SQL* ou um erro *HTTP*.

A remoção de usuário consiste numa página apenas com a inserção do nome de usuário. Se ele existir, será removido. Lembrando que superadministradores não podem ser removidos por esse meio.

Para criar e remover superadministradores, é necessário remover pelo *container* utilizando a interface do *Flask* de linha de comando conforme a figura 24.

Para criar:

*python3 manage.py criar_super_administrador*

Para remover:

*python3 manage.py remover_super_administrador*

Figura 23 – Formulário de criação de usuário


Fonte: Elaboração Própria

Na criação, será solicitado o nome de usuário e a senha. Na remoção, apenas o nome. A senha deve seguir os mesmos critérios do formulário *web*.

Figura 24 – Comandos para criar e remover superadministradores

```
(env) thiago@thiagoVB /presencas-tcc (main)> docker compose exec gunicorn ash
~ $ python3 manage.py criar_super_administrador
Insira o nome de usuário: maya_inbar
Insira a senha:
Confirme a senha:
Usuário criado com sucesso
~ $ python3 manage.py remover_super_administrador
Insira o nome de usuário: maya_inbar
Usuário removido com sucesso
~ $
```

Fonte: Elaboração Própria

A alteração de senha segue o mesmo critério da senha ao criar um usuário. O formulário contém apenas a seção de senha da página de cadastro de usuários.

### 5.2.2 Diretórios e explicação geral

Os diretórios da aplicação externa estão na figura 25. A aplicação consiste em uma estrutura *Flask* (Pallets Organization, s.d.) com o uso de *Gunicorn* (CHESNEAU, 2024) para a produção. A princípio, ela deve ser utilizada com *Docker*, já que com uso de *containers*, o *NGINX* e o *Fail2Ban* são serviços que compõem a *stack*. Iremos discorrer sobre eles no capítulo 7. Cada diretório/arquivo tem uma função.

a) ***env***: Contém os módulos *Python* que a aplicação requer para executar.

b) ***Fail2Ban***: Contém o arquivo de *log* que é verificado para aplicar as ações e contém os arquivos de configuração do serviço como *jails*, filtros e ações.

c) ***instance***: Bancos de dados *SQLite* para ambiente de desenvolvimento e depuração de erros, não é utilizado em produção.

d) ***nginx***: Contém o arquivo de configuração para o *proxy* da aplicação.

e) ***presencas***: Contém todo o código da aplicação *Flask* como os *templates HTML*, códigos *CSS*, lógica de execução etc.

f) ***psql***: Contém a estrutura para o banco de dados de produção da aplicação.

g) ***.env*** **e** ***.env.template***: Variáveis de ambiente para a aplicação.

h) ***.dockerignore*** e ***.gitignore***: Arquivos *ignore* para o *Dockerfile* e *Git*, respectivamente;

i) ***config.py***: Contém as classes de constantes de configuração para a aplicação como, por exemplo, *URL* de banco de dados, configurações de sessões, *URL* do *CKAN* etc.

j) ***docker-compose.yml***: Receita *compose* para iniciar os serviços da aplicação.

k) ***Dockerfile***: Receita para construir a imagem da aplicação *Flask*.

l) ***manage.py***: Ponto de entrada para gerenciar a aplicação *Flask* via linha de comando.

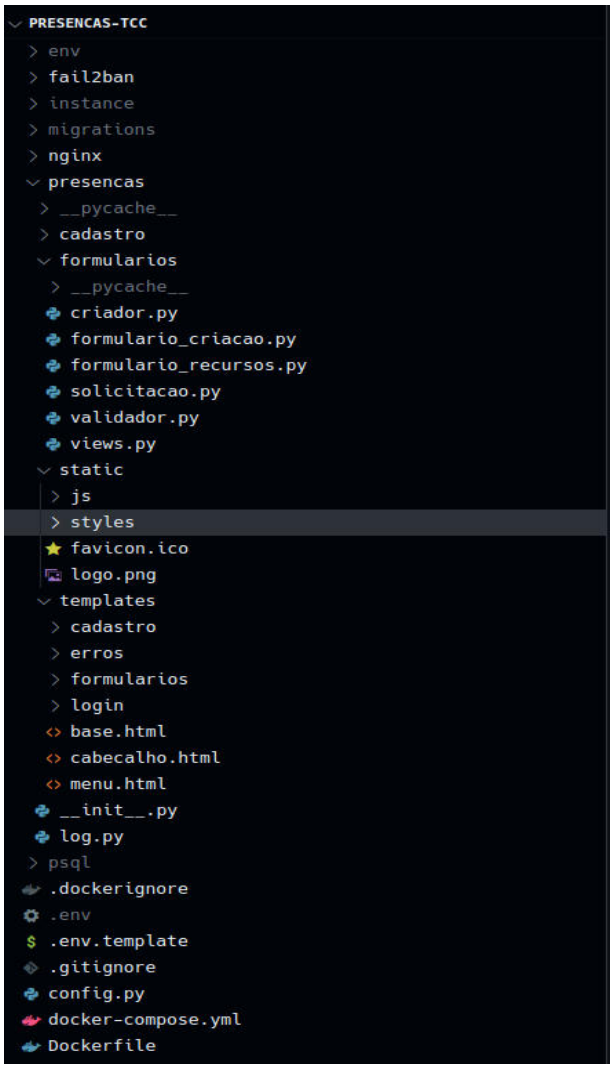
m) ***requirements.txt***: Arquivo típico do *Python* que contém a lista de módulos necessários para a aplicação executar.

Como o diretório *presencas* é o centro da aplicação, é necessário descrever os subdiretórios e arquivos.

a) ***cadastro***:

- ***controle_usuario.py***: Contém as classes que modelam os campos de formulários para as páginas de cadastro e remoção de usuários.
- ***login.py***: Contém a classe que modela os campos de formulário para autenticação na aplicação.
- ***usuario.py***: Contém a classe que modela as colunas no banco de dados de um usuário cadastro na aplicação.
- ***views.py***: Contém as funções que redirecionam para os caminhos na aplicação *web* para *login*, *logout*, criar e remover usuário e alterar senha. Para o método POST do *HTTP*, executa a lógica para conectar no banco, validar os dados do usuário e aplicar a respectiva ação. Para o método GET, retorna a página *HTML* respectiva.

Figura 25 – Diretórios da aplicação externa




Fonte: Elaboração Própria

b) ***formularios***:

- ***criador.py***: Código-fonte que contém toda a lógica para cadastrar os artistas e suas obras. Possui a classe que modelam um artista e seus dados e informações de imagens, a classe que realiza as requisições para o *CKAN* e as funções que iteram nos dados recebidos dos formulários de criação dos usuários.
- ***formulario_criacao.py***: Contém a modelagem dos formulários de criação de artistas e obras.
- ***formulario_recursos.py***: Análogo ao anterior, mas para o formulário de adicionar obras para artistas já cadastrados.
- ***solicitacao.py***: Modela a tabela "solicitacoes" no banco de dados.
- ***validador.py***: Contém as classes para validar todas as entradas do usuário para os formulários de cadastro de artista e obras.

  - ***views.py***: Análogo ao *views.py* do diretório *cadastro*.

c) ***static***:

  - ***js***: Contém os códigos originais e minimizados de *JavaScript* para ações de botões nas páginas, criação de requisição *AJAX*, verificação dos critérios de senha do lado do cliente, adição ou remoção dos avisos de erro. Também contém as bibliotecas *highlight* e *JQuery*.
  - ***styles***: Contém os arquivos *CSS* que estilizam toda a aplicação.

d) ***templates***: Contém todos as páginas HTML em formato *Jinja*. Este formato possui uma documentação específica e é utilizado pelo *Flask* para gerar uma página com base no estado de variáveis do código *Python*, ao invés de páginas estáticas;

e) ***__init.py__***: Contém todo o ponto de entrada da aplicação *Flask*, carregamento de configuração da aplicação e as variáveis que gerem o banco de dados;

f) ***log.py***: Contém a configuração de *log* da aplicação. Configura os formatos de mensagens, o arquivo de *log* e o tamanho dele. Faz a divisão da configuração do *log* de depuração pro de produção.

## 5.3 FLUXO DE EXECUÇÃO DE CÓDIGO E DIAGRAMA DA APLICAÇÃO

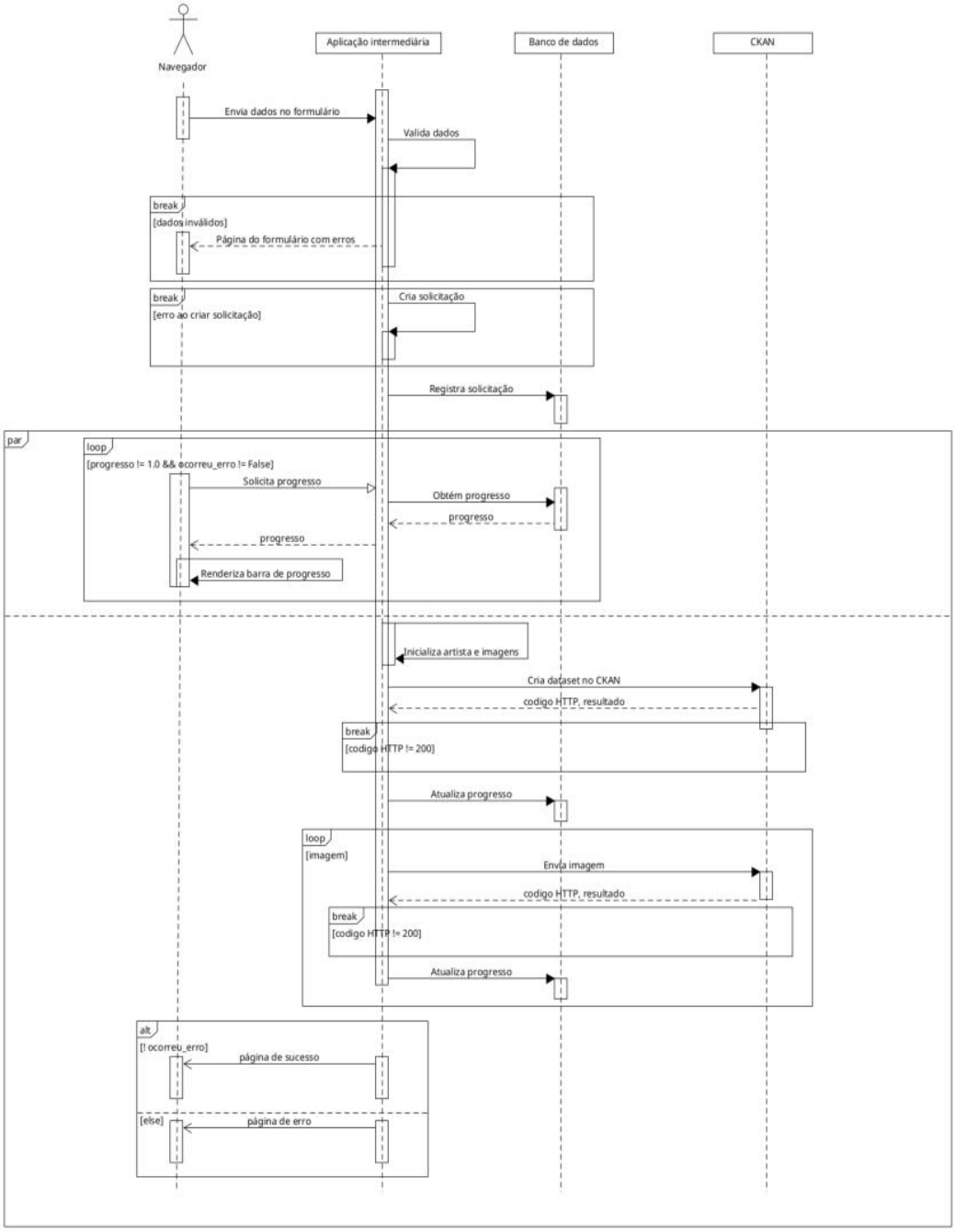
Ao iniciar a aplicação, seja pelo comando *Gunicorn* do ambiente virtual do *Python* (*venv*), seja pelo *Docker*, o arquivo *__init__.py* no diretório *presencas* é iniciado. A aplicação *Flask* inicia conforme as configurações de produção ou desenvolvimento. As *blueprints*, que são formas de modularizar a aplicação, são carregadas. Existem dois módulos: *cadastro* e *formularios*. Os *templates* de páginas de erro são configurados para não serem os padrões do *Flask*. O *log* é configurado para produção ou desenvolvimento conforme uso.

Com a aplicação carregada, ela fica disponível na porta especificada para desenvolvimento ou via comando. No caso de rodar via *container*, ela fica disponível na porta 80. Ao acessar qualquer página, a função com o decorador cujo respectivo endereço *URI* seja igual ao solicitado retornará o que foi solicitado. Após preencher os dados, eles são recebidos e validados para o caso de campos múltiplos e para nome, trajetória e produção. Se tudo estiver correto uma solicitação é criada. Em caso de sucesso, para cada imagem inserida, uma solicitação via *API* também é criada para ela. A imagem enviada via formulário é adicionada no repositório do artista. A adição de novas obras para artistas já existentes é similar, porém a etapa de criar o *dataset* do artista é desconsiderada. Caso o artista e/ou suas obras sejam criadas com sucesso, é retornado 201 de *Created*. Os detalhes completos deste fluxo estão no apêndice E.

A figura 26 mostra um diagrama UML de sequência com uma simplificação de como ocorre o fluxo. As variáveis, funções e classes foram abstraídas, focando apenas na ativi-

dade em si. Podemos ver que a solicitação do navegador pelo progresso via *AJAX* e todo o processo de criação de artistas e obras ocorre paralelamente. Ao final, se não ocorreu erro conforme mostra a variável, a página de sucesso é exibida.

Figura 26 – Diagrama simples UML de sequência para o cadastro de artistas e obras




Fonte: Elaboração Própria

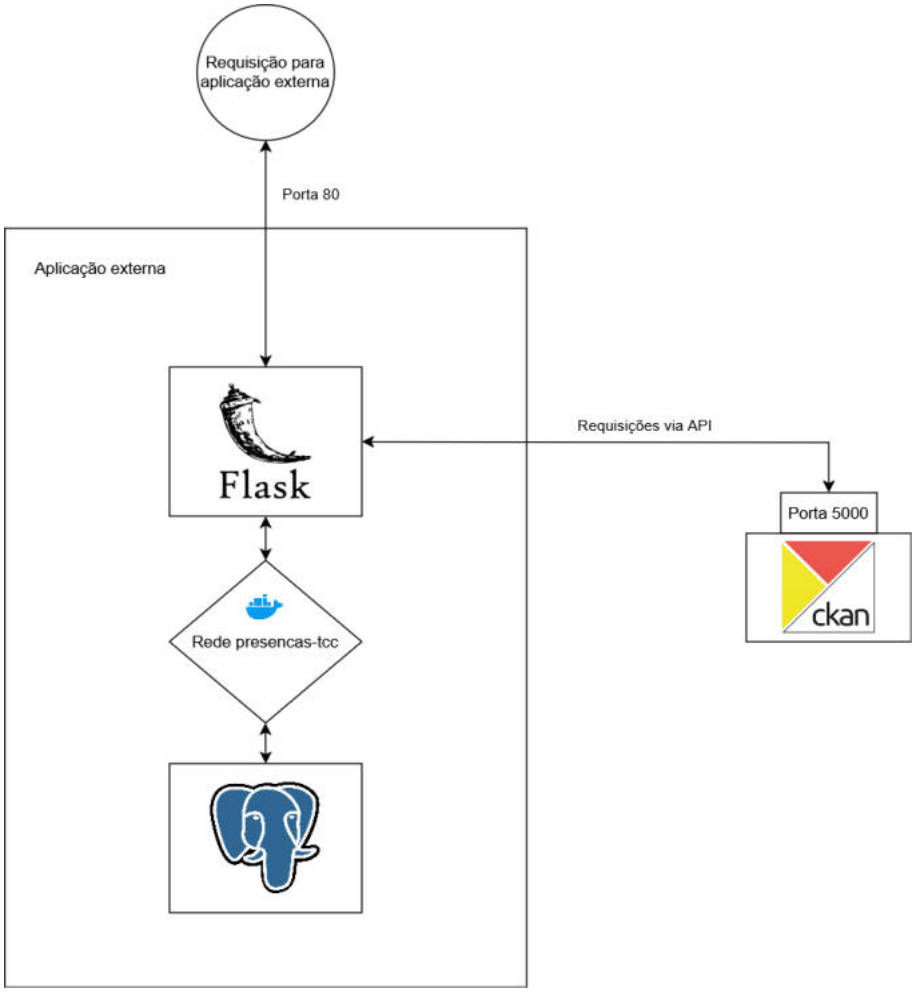
A criação e remoção de usuário são controladas pelas funções *criar_usuario* e *remover_usuario* do arquivo *views.py* no diretório *cadastro*. Ao acessar as *URIs* correspondentes para cada ação, a página é retornada no método GET. Para o método POST, os

dados enviados via formulário são validados pelos métodos *validate* de cada classe que modela os campos dos formulários. Para a criação, é verificado se o usuário existe e se a senha atende aos critérios desejados como números, letras maiúsculas e minúsculas e tamanho máximo de 128 caracteres. E também se a senha e a confirmação estão iguais. Se a validação está correta, o usuário é criado. No caso de remoção, apenas é verificado se o usuário existe, se ele não é superadministrador (já que não pode ser removido via *web*) e se o usuário digitado não é igual ao atual, o qual realiza a ação. Se a validação é correta, o usuário é removido. A alteração de senha tem a validação similar à da criação de usuário, porém é verificado se a nova senha desejada não é igual à atual. Com a validação efetuada, o *hash* da senha é gerado e guardado no banco.

Em todos os casos, qualquer exceção oriunda do banco de dados, seja problema de conexão ou operacional ou qualquer outro, um código *HTTP* 500 é retornado ao usuário junto com a exceção SQL gerada pela aplicação.

A figura 27 contém o diagrama da aplicação externa até este capítulo e como ela se conecta com o *CKAN*. Assim como a solução *CKAN* em *Docker*, temos uma rede do mesmo tipo chamada "*presencas-tcc*", que conecta o *container* do banco de dados e o *container* da aplicação. Esta tem a porta 80 exposta para as requisições. A aplicação externa interage com a *API* do *CKAN* pela porta *5000*. No capítulo 7, será fornecido um diagrama geral após a introdução dos serviços auxiliares, do *proxy* geral e da criação do *cluster swarm*.

Figura 27 – Diagrama da aplicação externa e a interação com o CKAN




Fonte: Elaboração Própria

# 6 CRIANDO UMA EXTENSÃO COM NOVOS FILTROS PARA O *CKAN*

Com o conhecimento geral adquirido do Solr e visando melhorar a experiência do usuário, uma extensão foi escrita para adicionar filtros de pesquisa. Essa extensão implementa interfaces disponibilizadas pelo *CKAN* com comportamentos específicos (Open Knowledge Foundation, 2025f). Por exemplo, como o objetivo é adicionar filtros, iremos utilizar as seguintes interfaces:

a) ***IFacets***: Possui métodos para adicionar novas categorias para filtrar a busca para *datasets*, grupos e organizações;

b) ***IPackageController***: Possui métodos que alteram aspectos gerais dos *datasets*, como a indexação, exibição, atualização e procura;

c) ***ITemplateHelpers***: Como haverá um filtro que obtém informação dos *datasets*, é preciso registrar uma função que auxilia na geração da página de procura, para que o dado seja retornado ao *template HTML*;

d) ***IConfigurer***: Para tornar as páginas e outros recursos da extensão visíveis ao *CKAN*, é preciso implementar essa interface.

O código da extensão pode ser encontrado aqui[1]. O comportamento geral dela é simples. Ela registra os campos de cidade, estado, cidades/estados/países de atuação, linguagens e data de início como filtros para o usuário. Esse último não é um filtro de seleção, e sim um botão deslizante com um intervalo do *dataset* com menor ano de data de início para o maior valor encontrado em outro *dataset*. Caso não exista algum *dataset* ou nenhum apresente esse dado, o filtro não fica disponível. Além disso, a extensão *scheming*, até a versão existente neste trabalho, possui um *bug* onde campos que são listas de *strings* exibem caractere por caractere caso exista apenas um valor. Por exemplo, se o campo possui apenas um item com valor "Rio de Janeiro", a página que exibe o *dataset* irá mostrar cada letra de "Rio de Janeiro". A extensão também faz essa correção colocando o valor único numa lista.

Agora, aqui está a razão do uso da extensão *scheming* mesmo com o desenvolvimento da aplicação externa para cadastro. Ela adiciona todos os campos configurados nela como verdadeiros metadados de um *dataset*, isto é, eles não decaem na seção "extras". Em termos de manipulação, ocorre uma maior facilidade, já que os "extras" são um campo na representação de um *dataset* no *Solr*, o que dificulta tratar na extensão e adicionar como filtro. Ademais, a exibição deles nas páginas dos *datasets* e recursos é melhorada, pois podemos mudar os nomes dos campos na exibição, já que originalmente os nomes do mapeamento chave-valor é que são mostrados na seção de "extras", o que deixa a experiência do usuário piorada.

[1] https://github.com/thiagoc01/ckanext-presencas

A *stack* do *CKAN* acompanha uma imagem de *Apache Solr*. Esta imagem é especial e criada pela desenvolvedora. Ela já contém um esquema pré-pronto para o *CKAN*. Existem diversas delas, e o esquema e a versão do *Solr* dependem do versionamento da imagem dado pela etiqueta. Todos os campos dos *datasets* são campos de um documento no *Solr*. Os recursos dos *datasets* também possuem campos, que estão presentes com o prefixo de res_. Os campos com títulos em português são campos do Projeto Presenças que foram adicionados para os novos filtros da extensão. Todos os tipos e campos configurados para a coleção do *CKAN* estão no apêndice F.

A criação do *template* da extensão foi seguida conforme a documentação (Open Knowledge Foundation, 2025d). A extensão tem o nome de *presencas*. O código *Python* dela é descomplicado. A figura 28 mostra o método que adiciona os filtros na página de busca. E também mostra o método que corrige a indexação de listas. Este é necessário porque, para campos de listas de *strings*, o scheming gera um *JSON* em *string*, isto é, a representação em texto de uma lista. Logo, o valor retornado na busca do *Solr* será um texto de uma lista. Assim, ao gerar os filtros, não veremos cada valor na lista disponível para filtro, e sim um filtro com esse texto que indexa para um *dataset* que possui essa lista de valores. Por exemplo, se um *dataset* apresenta cidades de atuação com valores Rio de Janeiro e São Paulo, a lista seria indexada como "["Rio de Janeiro", "São Paulo"]". Na busca, teríamos um filtro em cidades de atuação com exatamente essa representação: "["Rio de Janeiro", "São Paulo"]". Quando na verdade o correto seriam dois filtros: "Rio de Janeiro" e "São Paulo". Essa é também uma das correções feitas pela extensão. O código que faz essa alteração e também a alteração já descrita na exibição de campos múltiplos quando apresentam um valor está na figura 29. No código, as representações textuais são convertidas em listas do *Python*. E para o caso de valores únicos, eles são transformados em uma lista com valor único. Portanto, o *CKAN* irá indexar e consultar apenas listas, não representações textuais nem valores individuais.

Iremos, agora, entender como isso é renderizado ao usuário. As extensões do *CKAN* criam uma pasta chamada "*assets*". Qualquer recurso carregado na interface *web* deve estar nesta pasta. E, para alterar o código *HTML* original do *CKAN*, um arquivo com o mesmo nome do *template* deve estar na pasta *templates*. Como explicado anteriormente, o *CKAN* utiliza o *Jinja*, que possui templates *HTML* para gerar páginas de forma dinâmica. Ao criar um *template* com o mesmo caminho e mesmo nome do original, ele será substituído.

Os códigos 2 e 3 mostram como ocorre a mudança. Primeiro, obtemos todo o escopo do *template* original com a primeira linha. A seguir, na linha 2, sobrescrevemos o bloco original que trata a exibição dos filtros. Nas duas linhas seguintes, importamos os códigos *CSS* e *JavaScript* escritos para esse objetivo. Exceto pelo filtro de data de início, todos os demais devem ser exibidos conforme o padrão do *CKAN*, ou seja, opções para escolha com o respectivo número de *datasets* contendo aquele valor para o campo. No entanto,

Figura 28 – Código para adicionar filtros e solicitar a alteração da indexação de listas

```
def dataset_facets(self, facets_dict, package_type):

    facets_dict['cidade'] = toolkit._('Cidades')
    facets_dict['estado'] = toolkit._('Estados')
    facets_dict['cidade_atuacao'] = toolkit._('Cidades de atuação')
    facets_dict['estado_atuacao'] = toolkit._('Estados de atuação')
    facets_dict['pais_atuacao'] = toolkit._('Países de atuação')
    facets_dict['linguagens'] = toolkit._('Linguagens')
    facets_dict['data_inicio'] = toolkit._('Data de início')
    return facets_dict

def organization_facets(self, facets_dict, organization_type, package_type):

    return facets_dict

def group_facets(self, facets_dict, group_type, package_type):

    return facets_dict

# IPackageController

def before_dataset_index(self, pkg_dict):

    converte_string_lista_indexacao(pkg_dict, 'cidade_atuacao')
    converte_string_lista_indexacao(pkg_dict, 'estado_atuacao')
    converte_string_lista_indexacao(pkg_dict, 'pais_atuacao')
    converte_string_lista_indexacao(pkg_dict, 'linguagens')

    adiciona_indexacao_listas(pkg_dict, 'cidade_atuacao')
    adiciona_indexacao_listas(pkg_dict, 'estado_atuacao')
    adiciona_indexacao_listas(pkg_dict, 'pais_atuacao')
    adiciona_indexacao_listas(pkg_dict, 'linguagens')
    return pkg_dict
```

Fonte: Elaboração própria

para a data de início, como mencionado, queremos um botão deslizante de intervalo com o mínimo sendo o menor ano de início e o máximo o maior. Por isso, temos os blocos *if* e *else*. Caso o filtro tenha nome "data_inicio", iremos exibir o botão deslizante. Note o uso da função que criamos na extensão (*presencas_retorna_minimo_maximo_data_inicio*). Essa função realiza exatamente o que menciona o nome. Ela retorna uma dupla com o menor e maior valor. Caso haja apenas um *dataset*, a dupla tem os mesmos valores. Caso não haja *dataset*, retorna *None* e a mensagem de não existência é exibida, conforme o padrão do *CKAN*. Caso contrário, o botão é gerado. Para todos os outros filtros, o padrão de exibição é utilizado, ou seja, invocamos a implementação original do bloco com *super()*.

Figura 29 – Código que indexa e exibe valores únicos em uma lista e as indexa, não representações textuais delas

```python
def converte_string_lista_exibicao(dataset, nome):

    # O ckanext-scheming possui um bug para exibir multiple_text se apenas um foi inserido
    # O código abaixo transforma string para uma lista com um item antes de exibir

    if dataset.get(nome) and type(dataset.get(nome)) != list:

        dataset[nome] = json.loads(json.dumps([dataset.get(nome)]))

def converte_string_lista_indexacao(dataset, nome):

    if dataset.get(nome):

        try:
            json.loads(dataset.get(nome))

        except Exception:
            dataset[nome] = json.dumps([dataset.get(nome)])

def adiciona_indexacao_listas(dataset, nome_lista):

    if dataset.get(nome_lista):

        lista = ast.literal_eval(dataset[nome_lista])

        dataset[nome_lista] = []

        for item in lista:

            dataset[nome_lista].append(item)
```

Fonte: Elaboração própria

Código 2 – Código HTML do *template* que gera os filtros da busca

```
{% ckan_extends %}
{% block facet_list_items %}

{% asset 'presencas/presencas-css' %}
{% asset 'presencas/presencas-js' %}

{% if name == "data_inicio" %}
    {% set datas = h.presencas_retorna_minimo_maximo_data_inicio() %}
    {% if datas != None %}
    <div data-module="presencas-data-inicio">
        <div class="sliders pt-4 pb-4 mt-4">
            <div class="intervalo-slide"></div>
            <span class="slide1-val"></span>
            <input class="slide" type="range" min="{{ datas[0] }}" max="
                {{ datas[1] }}" value="{{ datas[0] }}"/>
            <span class="slide2-val"></span>
            <input class="slide" type="range" min="{{ datas[0] }}" max="
                {{ datas[1] }}" value="{{ datas[1] }}"/>
        </div>
        <div class="botoes-data-inicio d-flex align-content-center
            justify-content-around gap-1 mb-3">
```

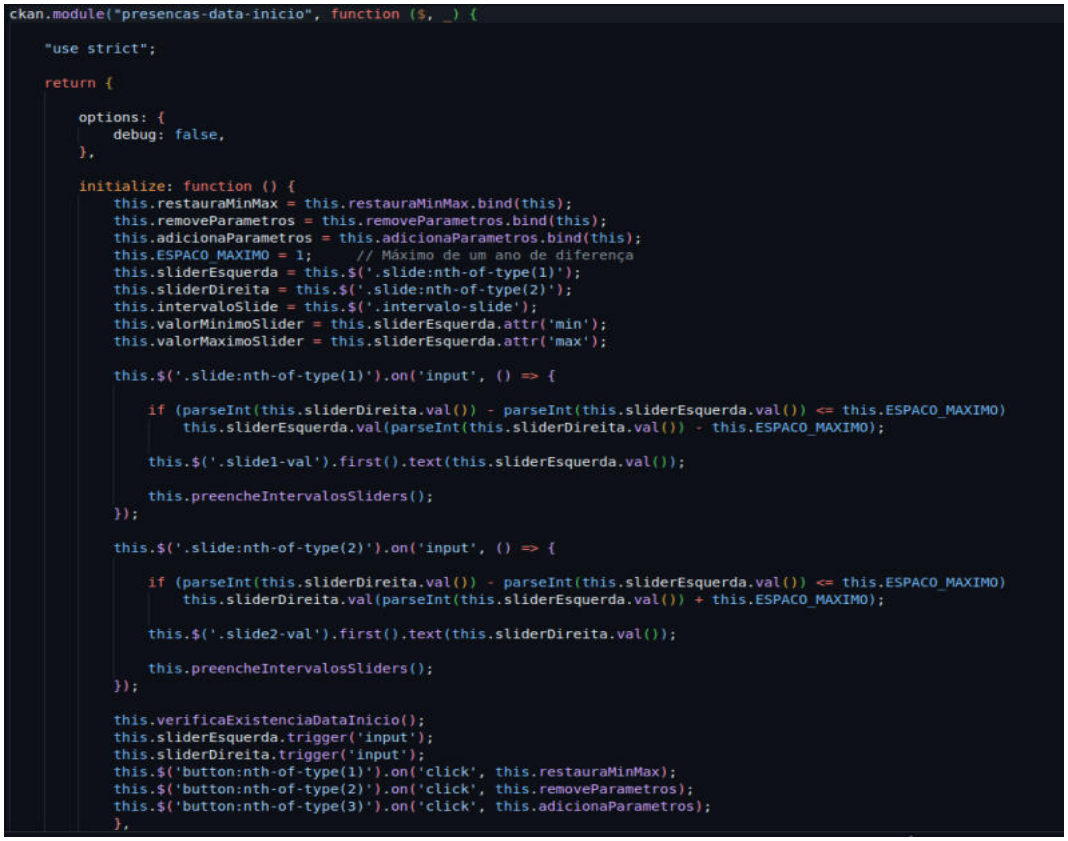
Código 3 – Código HTML do *template* que gera os filtros da busca (continuação)

```
            <button class="botao-data-inicio p-2 text-center">Restaurar
                botoes</button>
            <button class="botao-data-inicio p-2 text-center">Remover
                intervalo</button>
            <button class="botao-data-inicio p-2 text-center">Aplicar
                intervalo</button>
        </div>
    </div>
    {% else %}
    <p class="module-content empty">{{ _("Nao ha datasets com data de
        inicio configurada") }}</p>
    {% endif %}
{% else %}
    {{ super() }}
{% endif %}
{% endblock %}
```

Por fim, veremos rapidamente o código *JavaScript* para o botão deslizante. A figura 30 mostra a inicialização do módulo. Conforme a documentação do *CKAN*, a forma correta de implementar novos códigos *JavaScript* é através do uso de módulos, onde cada elemento *HTML* novo deve referenciar através do atributo *HTML* "data-module". Dessa forma, o código fica mais modularizado para cada extensão e também menos indisponível em caso de falhas ou problemas com a execução do *JavaScript*.

A função *initialize* trata da inicialização das variáveis, adições de eventos aos botões e ajustes para que o botão da esquerda não sobreponha o da direita e vice-versa. Conforme a figura 31, vemos que existem três botões: um que remove a data de início dos filtros, um que a adiciona nos filtros e o outro que restaura os botões para o mínimo e máximo do intervalo. Cada funcionalidade está implementada nas funções *removeParametros*, *adicionaParametros* e *restauraMinMax*, respectivamente. A distância dos botões não pode ser menor que uma unidade (um ano). Ao carregar a página, essa verificação é efetuada. Caso não haja parâmetro, os valores dos botões são de mínimo e máximo ou conforme navegação anterior do usuário. Caso o filtro tenha sido aplicado na consulta atual, os botões são configurados conforme os valores do parâmetro. O resto do código é apenas a lógica de preenchimento da barra conforme os valores de mínimo e máximo e ações *JQuery* para adicionar os eventos aos botões e dispará-los no carregamento da página.

Figura 30 – Código *JavaScript* do botão deslizante



```
ckan.module("presencas-data-inicio", function ($, _) {

    "use strict";

    return {

        options: {
            debug: false,
        },

        initialize: function () {
            this.restauraMinMax = this.restauraMinMax.bind(this);
            this.removeParametros = this.removeParametros.bind(this);
            this.adicionaParametros = this.adicionaParametros.bind(this);
            this.ESPACO_MAXIMO = 1;      // Máximo de um ano de diferença
            this.sliderEsquerda = this.$('.slide:nth-of-type(1)');
            this.sliderDireita = this.$('.slide:nth-of-type(2)');
            this.intervaloSlide = this.$('.intervalo-slide');
            this.valorMinimoSlider = this.sliderEsquerda.attr('min');
            this.valorMaximoSlider = this.sliderEsquerda.attr('max');

            this.$('.slide:nth-of-type(1)').on('input', () => {

                if (parseInt(this.sliderDireita.val()) - parseInt(this.sliderEsquerda.val()) <= this.ESPACO_MAXIMO)
                    this.sliderEsquerda.val(parseInt(this.sliderDireita.val()) - this.ESPACO_MAXIMO);

                this.$('.slide1-val').first().text(this.sliderEsquerda.val());

                this.preencheIntervalosSliders();
            });

            this.$('.slide:nth-of-type(2)').on('input', () => {

                if (parseInt(this.sliderDireita.val()) - parseInt(this.sliderEsquerda.val()) <= this.ESPACO_MAXIMO)
                    this.sliderDireita.val(parseInt(this.sliderEsquerda.val()) + this.ESPACO_MAXIMO);

                this.$('.slide2-val').first().text(this.sliderDireita.val());

                this.preencheIntervalosSliders();
            });

            this.verificaExistenciaDataInicio();
            this.sliderEsquerda.trigger('input');
            this.sliderDireita.trigger('input');
            this.$('button:nth-of-type(1)').on('click', this.restauraMinMax);
            this.$('button:nth-of-type(2)').on('click', this.removeParametros);
            this.$('button:nth-of-type(3)').on('click', this.adicionaParametros);
            },
```

Fonte: Elaboração própria

Figura 31 – Barra de filtros do *CKAN* após a extensão


Fonte: Elaboração própria

# 7 CRIAÇÃO DO *CLUSTER* PARA OS SERVIÇOS

Neste último capítulo de conteúdo, iremos criar um conjunto de máquinas utilizando um recurso do *Docker*. Também iremos configurar *softwares* que auxiliam na disponibilidade, redundância e gestão dos servidores e serviços.

## 7.1 MOTIVAÇÃO

Como já mencionado, serviços digitais devem ser oferecidos com a maior integralidade possível. Também já foi mencionado que tanto o *CKAN* quanto a aplicação auxiliar estão em *containers* baseados em *Docker*. Além da portabilidade, segurança e facilidade oferecidas por essa abordagem, o *Docker* possui um recurso em que um conjunto de máquinas podem ser conectadas entre si para gerar redundância e maior disponibilidade dos serviços. Esse recurso é chamado de *Docker Swarm* (Docker Inc., 2025d).

O modo de *Docker Swarm* oferece a capacidade de serviços estarem em diferentes máquinas para melhor uso de recursos e também para o caso de falha de algum hospedeiro, todos os serviços neste irem para outro disponível. Portanto, é criada uma automatização na gestão de falhas de hospedeiros e indisponibilidade de rede, já que todos devem comunicar-se entre si para se considerarem "saudáveis". Logo, já que essa capacidade é oferecida e servidores nos dias atuais trabalham em conjunto, em muitos casos com a virtualização sendo utilizada, optou-se por esse procedimento.

Teremos, portanto, um *cluster* de máquinas, isto é, um conjunto de máquinas interconectadas que juntas oferecem os serviços descritos nos capítulos anteriores. Dessa forma, com o modo *Swarm* do *Docker*, obteremos maior redundância, disponibilidade e melhor uso de recursos. Para isso, é necessário configurar:

a) Um servidor de arquivos para que todos as máquinas enxerguem os mesmos;

b) Um endereço IP único para que usuários o acessem e esse IP será controlado de forma automática em caso de indisponibilidade;

c) Um *proxy web* na máquina com o endereço IP único que receba as requisições e repasse para o container que as trata do serviço específico solicitado.

Além disso, é necessário configurar um *firewall* simples nas máquinas para bloquear quaisquer requisições que não sejam para os serviços ofertados. Quanto aos itens a) e b) mencionados, o apêndice G explica sobre o protocolo *VRRP*, que contribui com essa disponibilidade, e o protocolo *NFS* para acesso comum aos arquivos pelos servidores. Usaremos ambos os protocolos na implementação do *cluster*.

## 7.2 ESTRUTURA DOS SERVIDORES E APLICAÇÕES

Para a *clusterização*, iremos utilizar quatro máquinas neste trabalho. Três delas formarão o *Docker Swarm*. O *Docker Swarm* é uma ferramenta avançada do *Docker* que permite justamente a construção de conjuntos de máquinas com *Docker Daemons* e que interagem entre elas. No formato individual (*Docker Standalone*), uma máquina contém *containers*. Em *Docker Swarm*, existe o conceito de stacks e serviços. E também, análogo ao *VRRP*, *masters* e *workers*. Os *masters* controlam os nós e os *workers* apenas servem como hospedeiros de serviços. No caso de "controlar os nós", isto quer dizer que somente a partir dessas máquinas é que pode-se iniciar um serviço, adicionar novos nós ao *cluster* etc. Uma *stack* possui configurações de vários serviços, redes e volumes. Ao realizar o lançamento de uma *stack* no modo *Swarm*, os *masters* distribuem o serviço de forma que os recursos sejam bem utilizados em cada máquina. A forma de trabalho é similar a uma máquina com *Docker* individual, porém, a orquestração dos *masters* é o que permite a redundância de serviços e o reajuste de um *container* para outra máquina em caso de indisponibilidade. Já foram exibidos *Docker Composes* neste trabalho. Eles atuam como receitas para criação de serviços. Toda a configuração de rede, volume, verificação de estado, mapeamento de portas e outras possibilidades podem ser feitas ali. É possível, também, escolher quais nós podem receber o serviço, dependências, alterar comandos da imagem. Basicamente, as opções do comando *docker run* são disponibilizadas de forma declarativa neste arquivo.

Neste cluster de *Docker Swarm*, teremos os serviços em *container* do *CKAN*, da aplicação externa para cadastro, do *HAProxy* e do *Portainer*. O *HAProxy* será mencionado mais à frente por efetuar a função do gerenciador das requisições *HTTP*. O *Portainer* nos auxiliará na tarefa de gerenciar os serviços e *stacks*, bem como outros itens de um *Docker Swarm*. Ao invés de utilizarmos os comandos do *Docker Daemon* em cada máquina, o *Portainer* fornece uma interface visual numa aplicação *web* que permite gerenciar as imagens dos nós, verificar o estado deles, criar receitas, iniciar *stacks*, entre outras funcionalidades. Portanto, as três máquinas do *cluster* receberão esses serviços. Em nível de sistema e aplicações dele, iremos configurar:

- ***Keepalived***: *Software* que implementa o protocolo *VRRP*. Com ele, uma máquina do *cluster* terá um IP virtual que servirá para receber as requisições dos usuários para os serviços descritos. As requisições serão tratadas pelo *HAProxy*. Existirá apenas um *MASTER* e, como já mencionado, ele será o que possuir o *container* do *HAProxy*.

- ***IPTables***: *Firewall* praticamente nativo de sistemas *Linux*. Com ele, é possível criar regras de filtragem de pacotes, alteração de cabeçalhos de pacotes e uso de *NAT* para alterar endereços e portas. Criaremos regras de entrada de pacotes para

restringir que apenas dos serviços de *NFS*, do *Docker Swarm* e de *SSH* possam ser enviados. O *IPTables* também é pré-requisito padrão do *Docker* (podendo não ser utilizado).

- ***NFS***: Para compartilhar os arquivos persistentes dos serviços/aplicações, é necessário que todos os nós (máquinas) do *cluster* consigam enxergá-los. Portanto, o protocolo *NFS* será utilizado para criar um diretório comum às três máquinas.
- ***Fail2Ban***: Será discutido na próxima seção. Ele servirá para bloquear atacantes que realizem força bruta à autenticação da aplicação externa. O uso dele é desenvolvido para *Docker Standalone*. Para *Docker Swarm*, é preciso fazer uma configuração para que ele esteja sempre na mesma máquina do *HAProxy*, já que as regras de filtragem devem ser aplicadas para ele.

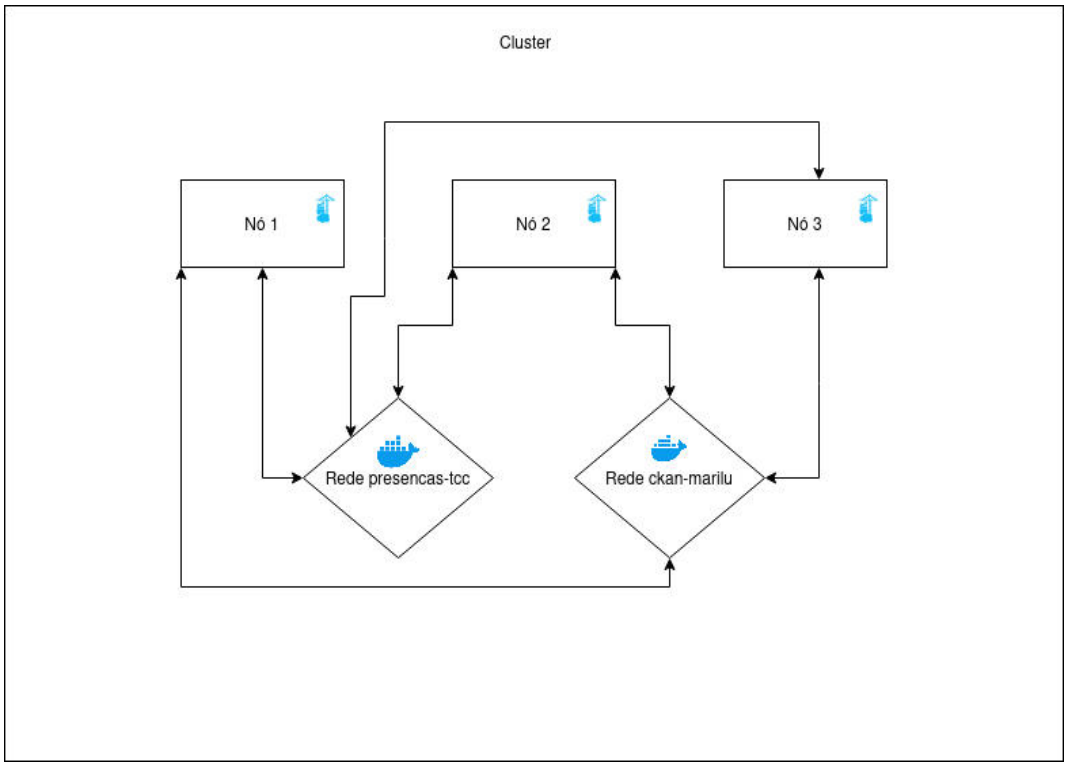
As quatro máquinas virtuais estão configuradas com o seguinte esquema:

- *Debian Bookworm* 12;
- 8 GB de memória *RAM*;
- 2 núcleos de processador;
- Um disco de 50 GB;
- Uma interface de rede (*enX0*).

A figura 32 contém apenas o diagrama em nível *Docker*. Isto é, apenas como as redes *Docker* conectam-se. Podemos ver também que em cada nó existe um serviço do *Portainer Agent*. Este serviço é responsável por reunir os dados do nó no *Swarm* e fornecer ao *Portainer Server*, que organiza-os em informações para a aplicação *web*. A rede *presencas-tcc*, já mencionada no capítulo 5, conecta os serviços da aplicação externa (*NGINX*, *Gunicorn* e banco de dados) e o HAProxy. Já a rede *ckan-marilu*, já mencionada no capítulo 4, conecta os serviços do *CKAN* (*CKAN*, banco de dados, *Redis*, *Solr*, *DataPusher*) e o *HAProxy*. O ideal era criar apenas a rede do *HAProxy* para que somente os *containers* que recebem o redirecionamento do *HAProxy* estejam conectados com ele. Mas, esse formato não será problema. O *cluster* foi criado conforme as instruções padrões do *Docker Swarm* (Docker Inc., 2025d).

Já a figura 33 exibe o diagrama em nível de sistema. Como mencionado, os serviços descritos acima estão em cada nó. Todos estão conectados a um servidor *NFS* central. Cada nó está com as cores conforme a figura do diagrama *VRRP* no apêndice G. Por definição e sem perda de efeito geral, o nó 1 começa como *MASTER* com o IP virtual 10.199.8.30. Caso o *HAProxy* suba em uma das outras máquinas, o *Keepalived* irá tratar e redirecionar o IP virtual para a nó que tiver o serviço. Todos estão conectados na rede 10.199.0.0/16. Isso é importante quando discutirmos as regras do *IPTables*.

Figura 32 – Diagrama do *cluster* em nível de Docker




Fonte: Elaboração própria

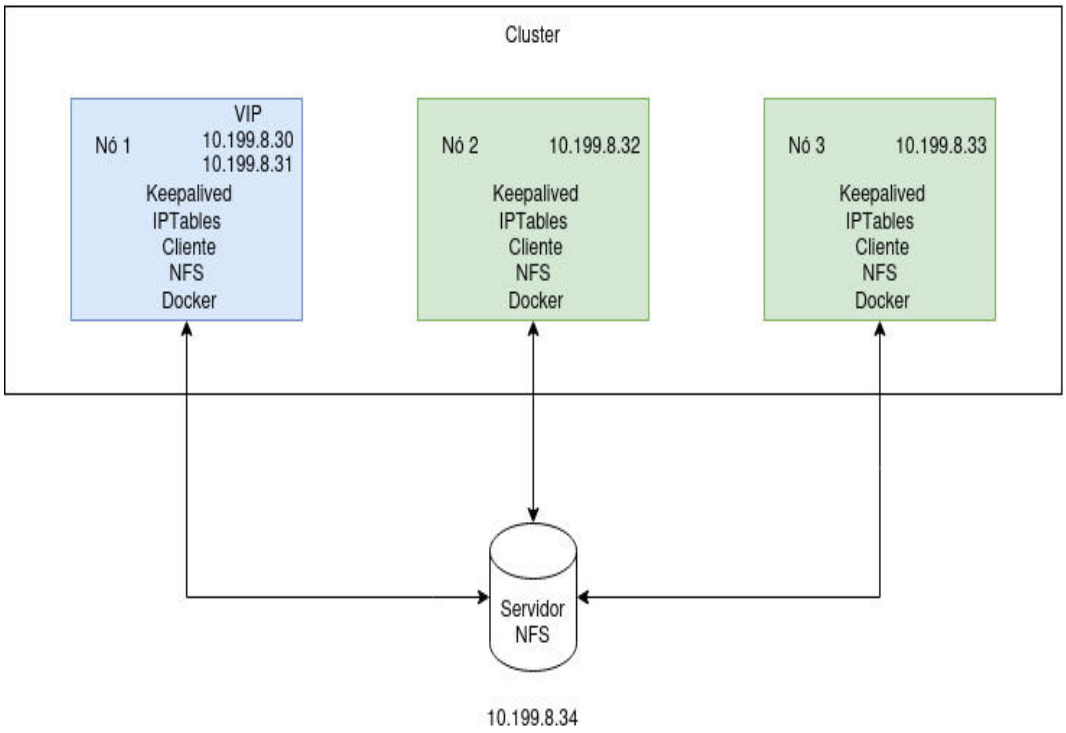
## 7.3 CONFIGURAÇÕES DO *NFS*, *KEEPALIVED*, *FAIL2BAN*, *HAPROXY* E *IPTABLES*

### 7.3.1 *IPTables*

Por haver poucos serviços nas máquinas e por serem servidores de borda, o simples uso do *IPTables* já é de grande auxílio para bloquear requisições que não sejam para serviços declarados e disponíveis. É assumido que o leitor entenda sobre as configurações básicas da ferramenta.

Como são servidores de borda, precisamos apenas trabalhar na cadeia de *INPUT*. Os únicos pacotes que passam pelas interfaces nativas da máquina são para redirecionamentos internos para os *containers*. Como o *Docker* gere a cadeia de *FORWARD*, que trata de pacotes encaminhados, não precisamos nos atentar a ela. No entanto, máquinas com ele precisam ter o *IPtables* configurado de forma diferente. Isto porque, ao iniciar, o *Docker* limpa todas as cadeias se houver ao menos um *container* executando, o que faz com que as regras inseridas por nós em algum momento sejam apagadas. Para isso, tomamos o cuidado de criar um serviço de sistema no *systemd*, utilitário de distribuições *Debian* que controla serviços e recursos deles e de processos no sistema. O serviço criado por nós é orientado a iniciar/reiniciar após o início ou reinício do serviço do *Docker*. Após tais eventos, as regras são inseridas.

Figura 33 – Diagrama do *cluster* em nível de sistema




Fonte: Elaboração própria

Pela baixa quantidade de serviços, as máquinas terão poucas regras na cadeia de *INPUT*. A regra geral da cadeia é de *DROP*. Qualquer pacote que não case com as regras abaixo deve ser **descartado**. A seguir, está a análise dos pacotes de entrada de cada máquina.

a) **Nós com *Docker* que participam do *Swarm***: Por estarem sob o protocolo *VRRP*, esse protocolo deve ser autorizado e somente na rede das máquinas, a 10.199.0.0/16. Por estarem em um *cluster* de *Docker Swarm*, as portas solicitadas (2377/*TCP*, 7946/*TCP*, 4789/*UDP* e 7946/*UDP*) devem ser permitidas.

b) **Nó do servidor *NFS***: Apenas as portas de RPC e *NFS* devem ser disponibilizadas, que são 111 e 2049 tanto para *TCP* quanto para *UDP*.

c) **Comum a todas as máquinas**: O serviço de *SSH* deve ser permitido. A porta escolhida para diferenciar da comum é a 22022. Somente solicitações da rede interna devem ser permitidas. Logo, pacotes para a porta 22022 da rede 10.199.0.0/16 devem ser liberados. E, claro, a interface de *loopback* da máquina pode receber conexões de qualquer rede de qualquer interface. Por fim, para pacotes de entrada de conexões já estabelecidas não serem descartados, uma regra que permite a entrada deles é adicionada.

O apêndice H contém a configuração do serviço para o *systemd* e os *scripts* que adi-

cionam as regras nos nós do *cluster swarm* e no servidor *NFS*.

### 7.3.2 NFS

A configuração do servidor *NFS* é basicamente a padrão ao instalá-lo. No arquivo */etc/exports*, o que configura os diretórios a serem compartilhados pelo protocolo, adicionamos a linha no código 4:

Código 4 – Permissão para as máquinas acessarem o diretório comum

```
/docker-volumes    10.199.8.31(rw,sync,no_subtree_check,no_root_squash)
    10.199.8.32(rw,sync,no_subtree_check,no_root_squash) 10.199.8.33(rw,
    sync,no_subtree_check,no_root_squash)
```

Este diretório é o que servirá para abrigar outros diretórios e arquivos para os serviços que rodarão no *cluster swarm*. Os endereços IPs das três máquinas que estão permitidas a acessar esse diretório estão presentes, e somente elas tem essa concessão. Entre os parênteses, os parâmetros de exportação presentes na documentação do pacote.

Do lado do cliente, basta montarmos o diretório pelo arquivo */etc/fstab* com o sistema de arquivo sendo NFS, apontando para *10.199.8.34:/docker-volumes*.

### 7.3.3 HAProxy

O *HAProxy* é uma ferramenta gratuita para *proxy* reverso e balanceamento de carga, tanto para *HTTP* e *TCP*. A sigla *HA*, do inglês: alta disponibilidade, define que o objetivo principal é permitir que as aplicações tenham essa disponibilidade acima do normal, ou seja, o chamado *downtime* (tempo de queda) deve ser praticamente imperceptível. Existem outros conceitos por trás desse termo, mas não iremos analisá-los.

Para esse projeto, usaremos as funcionalidades mais básicas. Para nossa estrutura, ele será um *proxy* reverso que recebe as requisições *HTTP* e redireciona para os *containers* de *frontend* dos serviços. De antemão, não iremos expor a construção de uma configuração para o protocolo *HTTPS*, haja vista que o ambiente construído não está exposto ao mundo, o que torna impossível realizar desafios de prova de posse de domínios e subdomínios. Consequentemente, não há como gerar certificados confiáveis.

Como mencionado anteriormente, apenas um nó irá conter o *container* do *HAProxy*. A porta 80 do hospedeiro ficará exposta para receber as requisições *HTTP*. Como esse hospedeiro possui o *container* do *HAProxy*, ele terá o IP virtual do roteador virtual já descrito. A depender do *Host* no cabeçalho *HTTP* da requisição, o tráfego será redirecionado ou para a aplicação externa de cadastro ou para o *CKAN*. Nesse cenário, há um adendo. Como as *URLs* do *CKAN* estão configuradas para o IP virtual e não para um nome de domínio, se a solicitação for para o IP virtual, o tráfego é redirecionado para o *CKAN*.

O apêndice I mostra a configuração do *HAProxy*. O modo padrão é *HTTP* porque trabalharemos apenas com ele. Os tempos de expiração de cliente, conexão ou servidor são de dez segundos, o que fecha a conexão do ponto de vista do *HAProxy*. Declaramos um bloco *resolvers*. De acordo com a documentação, podemos declarar servidores *DNS* para que nomes sejam resolvidos. Como trabalharemos com os nomes dos serviços no *Docker*, precisamos resolvê-los com o *DNS* dele. O servidor *DNS* dele é no endereço 127.0.0.1 na porta padrão (é possível verificar isso acessando o arquivo *resolv.conf* dentro de um container). Os parâmetros *hold* definem a mudança de estado de *UP* para *DOWN*, ou seja, o estado de disponibilidade ou não. Dado o tempo em um dos tipos de resposta, aguarda-se para receber uma nova resposta válida. Caso contrário, é considerado que o serviço está indisponível (TECHNOLOGIES, 2025).

Criamos um *frontend*, responsável por escutar na porta 80 e tratar as requisições. Com as diretivas *use_backend*, redirecionamos o tráfego dada a resposta da análise feita no bloco entre chaves. Se o *Host* no cabeçalho *HTTP* tem o endereço "*ckan.tcc.ufrj.br*" ou tem o endereço do IP virtual, redireciona-se para o serviço do *CKAN*. Se o *Host* tem o endereço "*presencas-cadastros.tcc.ufrj.br*", redireciona-se para a aplicação de cadastro.

Para cada um dos serviços, existe um *backend*. Declaramos em cada um deles um servidor apontando para o ponto de entrada de cada aplicação. No caso do *CKAN*, o próprio *container* dele na porta 5000. No caso da aplicação de cadastro, o *NGINX* que trata as requisições. Como mencionado, esses nomes declarados são dos **serviços**. Precisamos utilizar a resolução de nomes do *Docker*. Fazemos isso adicionando *check resolvers docker* para que o *HAProxy* consulte os nomes. E com *init-addr none*, dizemos que não existe um endereço atribuído a princípio, forçando uma consulta logo ao iniciar o *HAProxy*. Por fim, a diretiva *option forwardfor* no *backend* da aplicação de cadastro tem a função de repassar o endereço do cliente original (usuário) que iniciou a requisição *HTTP*, já que do ponto de vista do *NGINX* ele receberá uma nova requisição com o endereço do *HAProxy* sendo o cliente. Essa opção é importante para gerar um *log* correto do endereço e para a verificação do *Fail2Ban*.

### 7.3.4 *Keepalived* e *Fail2Ban*

A configuração do nó *MASTER* do *Keepalived* (MACK; CASSEN; ARMITAGE, 2022) está no código 27 no apêndice J. Configuramos apenas uma instância *VRRP* (*vrrp_instance*) com um roteador virtual pela simplicidade do *cluster*. Atribuímos o ID de 220 aleatoriamente. Como dito, os hospedeiros estão na rede 10.199.0.0/16 e o IP virtual é o 10.199.8.30. Logo, para o hospedeiro com estado *MASTER*, ele receberá este IP conforme o bloco *virtual_ipaddress*. O bloco *vrrp_track_process* é onde configuramos que o processo do *HAProxy* deve ser investigado a cada criação ou remoção de processos na máquina. Caso uma dessas ações ocorra, o *Keepalived* deve adicionar ou remover 20 na prioridade do hospedeiro e retornar essa informação para as demais máquinas. As outras

duas máquinas do *cluster Docker Swarm* entram em estado *BACKUP*. A prioridade das outras duas começa com 99 e 98, respectivamente. Como a prioridade do *MASTER* é 100 nativamente, é garantido que, se o *HAProxy* "flutuar" para outra máquina, o outro nó será *MASTER*. Com o *track_process*, referenciamos na instância *VRRP* que ela deve ser orientada ao processo definido.

Adicionalmente, temos duas diretivas: *notify_master* e *notify_backup*. Elas estão presentes para subir o *Fail2Ban*. O *Fail2Ban* é uma aplicação que permite criar filtros, ações e cadeias. Com filtros, é possível obter pedaços específicos de arquivos de *log*. Esses trechos são considerados comportamentos suspeitos a depender do desejo de quem configura o *Fail2Ban*. As **ações** constituem medidas a serem tomadas conforme limites específicos ou comportamentos repetidos. Ao unir ações e filtros, criamos uma **cadeia**. Nas cadeias, determina-se o que fazer baseados nesses comportamentos e métricas obtidas. Por exemplo, se um usuário solicita muitas páginas em um tempo extremamente curto, isso provavelmente é um ataque de negação de serviço. Ou, se um usuário erra uma senha 50 vezes, provavelmente está tentando realizar um ataque de força bruta. Logo, unindo essas informações, aplicam-se **ações**. Portanto, dado esse poder do *Fail2Ban*, podemos proteger a aplicação externa. Como apenas a parte de autenticação é exposta à *Internet* inicialmente, a principal proteção que podemos oferecer é bloquear um usuário após um número de tentativas. Dessa forma, o *Fail2ban* faz-se necessário.

O grande problema é que o formato dele não é o ideal para *Docker Swarm*. Precisamos que ele esteja junto com o *HAProxy* sempre para que as regras de bloqueio sejam aplicadas na mesma máquina. Através das configurações de *stack*, esse controle é bem precário. Para controlar que os dois subam juntos, precisamos criar um subconjuntos de máquinas com uma etiqueta específica. Porém, existem apenas três. Teríamos quase nenhuma opção de redundância caso um nó caia. Logo, essa não é a melhor abordagem. Com isso, entram as duas diretivas citadas acima. Como ele deve acompanhar o *HAProxy* e o *Keepalived* está orientado ao *HAProxy* basta que, ao subir o *HAProxy*, o *Keepalived* execute um *script* que também execute a criação do serviço do *Fail2Ban*. De forma contrária, se o *HAProxy* não estiver mais na máquina, o *Fail2Ban* também deve ser removido. A diretiva *notify_backup* realiza esse papel, já que uma máquina sem o *HAProxy* está em estado de *BACKUP*. Os códigos 25 e 26 no apêndice J possuem, respectivamente, os scripts em *Bash* da criação e remoção do *Fail2Ban*. Verifica-se se o *HAProxy* está de fato ali para subir o serviço. Assim como também verifica-se se não existe o *HAProxy* na máquina para poder remover o serviço do *Fail2Ban*. Com isso, apenas a máquina com *HAProxy* irá conter o *Fail2Ban* para aplicar regras à porta *HTTP*, já que com *Docker Swarm* cada um poderia estar em máquinas diferentes, o que torna o *Fail2Ban* inútil.

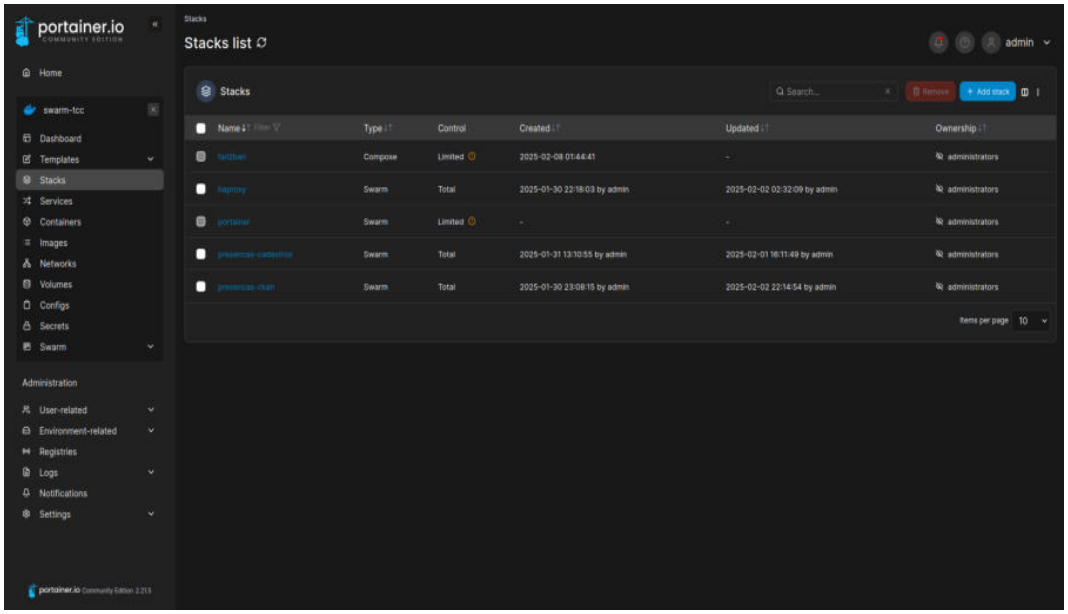
O código 24 no apêndice J contém a receita *Compose* do *Fail2Ban*. Fornecemos a capacidade de sistema de administrador de rede a ele, para que possa utilizar o *IPTables*. Colocamos o modo de rede em **modo de hospedeiro** para que ele possa enxergar todas

as interfaces da máquina hospedeira, já que o *namespace*, como já mencionado no capítulo 3, isola do *namespace* do hospedeiro. Mapeamos o *log* da aplicação externa para que o *container* do *Fail2Ban* possa ler para aplicar os filtros configurados. Caso o usuário erre a senha cinco vezes conforme a linha **"*Nome de usuário ou senha incorretos*"**, ele terá o IP banido com tempo que cresce conforme uma fórmula padrão do *Fail2Ban*. Os códigos 22 e 23 no apêndice J contêm, respectivamente, as configurações de filtro e cadeia feita para a aplicação externa do Projeto Presencas.

## 7.4 ESQUEMA FINAL DAS APLICAÇÕES NO DOCKER SWARM

Para finalizar, veremos a organização dos diretórios e *stacks* no *Docker Swarm* e um diagrama do *cluster* por completo. A figura 34 mostra o ambiente *swarm-tcc* criado no *Docker Swarm*. Temos a *stack* do *CKAN* e a *stack* da aplicação de cadastro, ambas reconhecíveis pelos nomes. Temos a *stack* do *HAProxy* e a do próprio *Portainer*. Observe que o *Fail2Ban* está criado no ambiente, mas não está em modo *Swarm*, pois a instanciação está apenas na máquina com o serviço do *HAProxy*.

Figura 34 – Ambiente *swarm-tcc* com as *stacks* no *Portainer*




Fonte: Elaboração própria

Não iremos exibir os *composes* do *CKAN* e da aplicação *web* porque são praticamente iguais às já apresentadas anteriormente. Apenas algumas modificações foram feitas conforme descrito nos repositórios *Github* já divulgados nos capítulos 3 e 4. Os volumes do lado do hospedeiro estão no diretório */docker-volumes/* que é compartilhado pelo *NFS*.

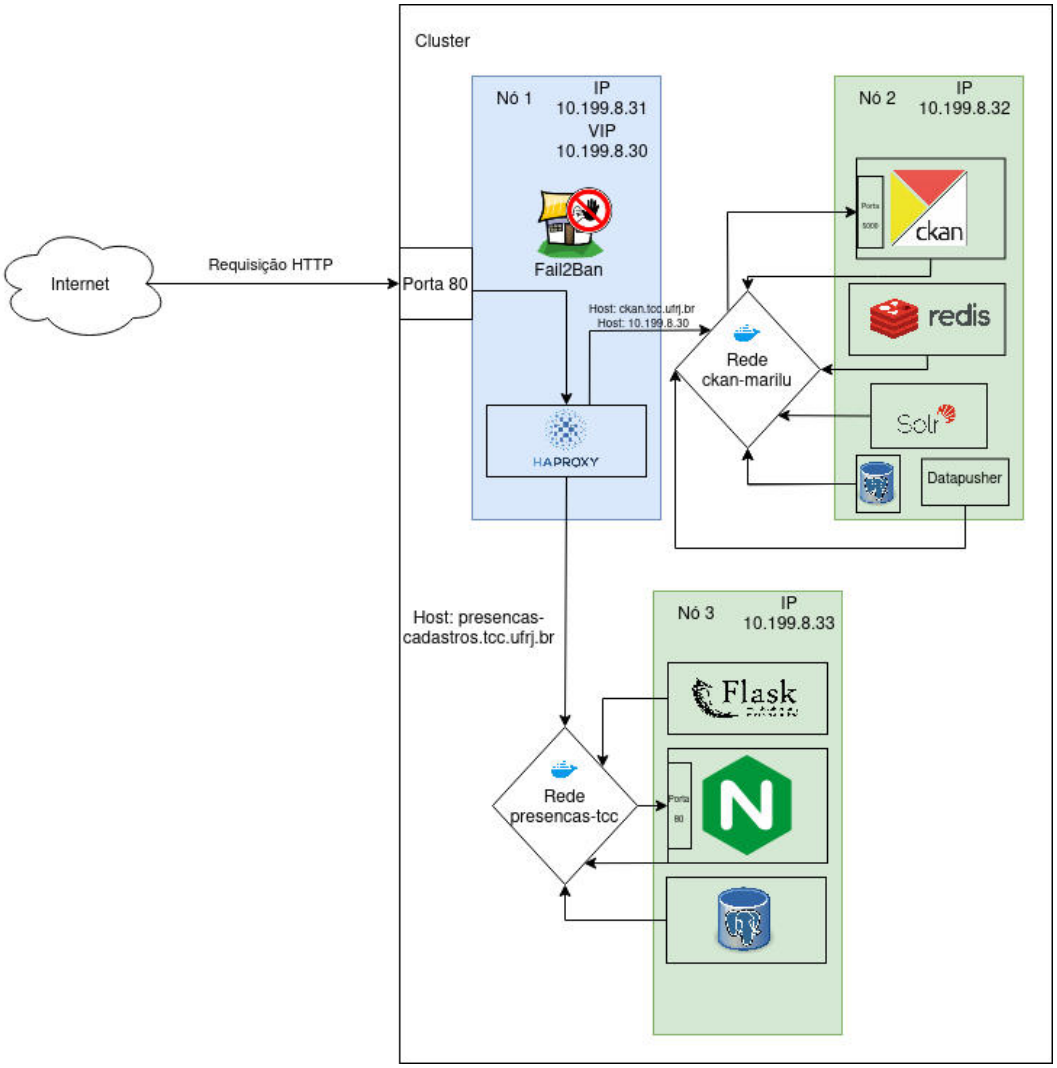
A figura 35 contém o diagrama final do *cluster* com serviços, portas e requisições. O roteador virtual fica em posse do nó com o *HAProxy*. Para melhor entendimento, foi

feita uma divisão dos serviços de como ocorre o fluxo de uma requisição. Essa decisão foi tomada porque a distribuição de serviços é aleatória, não é possível saber como será a configuração de distribuição. Logo, para facilitar a compreensão, assumiu-se um cenário em que a *stack* do *CKAN* foi colocada no nó 2, a da aplicação externa no nó 3 e o *HAProxy* e o *Fail2Ban* no nó 1.

A requisição chega pela porta 80 do nó com o *HAProxy*, que redireciona para a porta 80 do *container*. Dependendo do cabeçalho, redireciona ou pra porta 80 do *NGINX* da aplicação externa ou para o *container* do *CKAN* na porta 5000. O *NGINX* da aplicação externa é importante tanto em eficiência quanto proteção. Na configuração dele, existem dois blocos de regras dependendo da *URI* solicitada. Caso o usuário solicite somente arquivos *JavaScript*, *CSS* ou de imagem, o *NGINX* os serve sem redirecionar para a aplicação. Qualquer outra solicitação é enviada a ela. Em questão de eficiência, o serviço de arquivos do *NGINX*, que é excelente, é utilizado. E em questão de segurança, requisições maliciosas que visam realizar ataques ao sistema de arquivos e comprometimento deles serão menos suscetíveis ao sucesso.

As redes *Docker* foram representadas. Apesar das setas utilizadas, a comunicação ocorre nos dois sentidos. Embora não tenha sido representada, a leitura dos *logs* da aplicação externa é feita pelo *Fail2Ban*. Como já mencionado, após cinco tentativas, o IP do usuário é banido. O servidor *NFS* também não foi representado para simplificação. Como já explicado, os três nós atuam como clientes e acessam o diretório */docker-volumes/*.

Figura 35 – Diagrama final do *cluster swarm* e de todas as aplicações




Fonte: Elaboração própria

# 8 CONCLUSÃO

Este trabalho apresentou uma solução de *cluster* de servidores com *Docker Swarm* de um repositório de dados que é catalogável e com padronização de metadados com grande capacidade de busca para um usuário comum ou avançado. Foram introduzidas as ferramentas utilizadas e explicado de forma breve como funciona o esquema de *containers* em um sistema *Linux*. Foi motivado que o uso de *Docker* no gerenciamento de *containers* geram segurança, portabilidade e facilidade na implementação de serviços. E também foi explicado o que é a aplicação que utilizamos para repositório de dados e o padrão de metadados *DCAT* disponibilizado por ela. Foi apresentado o Projeto Presenças, a estrutura de dados dele, como mapeamos os dados para o padrão de metadados *DCAT* e extensões necessárias para abrigar o máximo possível de metadados.

Em seguida, foi apresentada uma aplicação externa para cadastro de artistas e obras no *CKAN*, haja vista que para uma maior manipulação e interação da plataforma, faz-se necessário o conhecimento da linguagem de programação que construiu o *CKAN*. A construção da aplicação foi uma grande revisão de alguns assuntos como banco de dados, protocolo *HTTP* e desenvolvimento *web*. Além disso, para uma melhor interação do usuário com o *CKAN*, apresentou-se uma extensão que adiciona filtros com uma *API* dada pela própria desenvolvedora da aplicação. Para entender como o ele indexa e consulta os documentos, a ferramenta que faz tal trabalho, *Apache Solr*, teve o funcionamento explicado. Por fim, para construir um ambiente de *containers* com redundância, confiabilidade e alta disponibilidade, exibimos a configuração de um *cluster* em *Docker Swarm* com uso do protocolo *VRRP* com um endereço geral para acesso único e independente de qual máquina possui o *proxy* reverso e de ferramentas auxiliares como um servidor *NFS* para acesso comum a um diretório de arquivos, um *firewall* para proteção das máquinas e uma aplicação que introduz ações baseadas em comportamentos considerados suspeitos.

Dentre as limitações, este trabalho não contemplou o desenvolvimento de uma interface visual agradável e simples para o usuário final na aplicação externa. A aplicação não pôde ser avaliada e testada pela equipe do Projeto Presenças durante o desenvolvimento, somente tendo o funcionamento exibido. Por não haver esse *feedback*, possíveis necessidades existentes não puderam ser verificadas. Também houve o desafio de adaptar a ferramenta para gerar *logs* simples e eficientes para o *Fail2Ban*, bem como seria a implementação em *containers* e como configurar os *logs* de acordo com ambiente de desenvolvimento ou produção, já que para este último utiliza-se o *Gunicorn*, que tem as próprias configurações de *logs*. O uso de requisições assíncronas em um ambiente síncrono para criar uma tela de carregamento que informa ao usuário o progresso entre a aplicação e o *CKAN* na comunicação com *API* deste foi outro desafio adicional. Já na extensão desenvolvida para o *CKAN*, a leitura da documentação fornecida das interfaces a serem

implementadas para estender o comportamento da plataforma demandou tempo e uma necessidade de entendimento maior, já que alguns pontos são incompletos e faltam serem documentados, além de ser necessário adaptar para o objetivo desejado. Contudo, foi mostrada a possibilidade de criar um repositório consistente com filtros baseados nos metadados, adjunto a uma aplicação auxiliar extremamente simples para os membros do Projeto cadastrarem os artistas, dividirem o trabalho entre eles com a criação de usuários comuns ou administradores e com alta disponibilidade dado pelo conjunto de máquinas conectadas e com redundância.

Para trabalhos futuros, alguns pontos podem ser revistos. O *CKAN* pode ser reformulado visualmente como foi mencionado neste trabalho. De acordo com as ideias da coordenadora do projeto, Maya Inbar, a página inicial poderia ter um mapa do Brasil com o filtro de cidades e estados aplicados a ele, e abaixo a exibição de artistas dessas cidades e estados. Quanto ao código da aplicação externa, melhorias podem ser feitas como remoção de código repetido, simplificação no uso do *JQuery* para o *frontend* das páginas e dos *templates HTML* e melhores tratamentos para cenários de concorrência de acesso ou erros gerados. Também, pode-se fazer uma análise da complexidade de tempo de algumas funções/métodos que potencialmente podem ter uma maior carga de processamento, criação de testes, o que é necessário em ambientes de desenvolvimento, e também um melhor tratamento e disponibilidade de informação de artistas já existentes ao usuário no momento de preencher os formulários de criação de artistas/adição de obras. Por exemplo, sugere-se, para uma melhor experiência, que seja retornada uma lista de artistas existentes no *CKAN* para que o usuário tenha certeza que o novo artista a ser cadastrado não esteja na plataforma, pois no momento de preenchimento pode haver um pequeno erro de digitação do nome, o que irá fazer com que o artista seja criado novamente, apesar da incorretude do nome. Dessa forma, irá haver redundância de dados e uma má utilização do repositório tanto para os gestores quanto para o usuários que irão visualizar dados repetidos. Outro ponto extremamente interessante e que avançaria assim como na evolução da *containerização* é a conversão de *Docker Swarm* para *Kubernetes*. *Kubernetes* é um esquema de *containers* que possui o conceito de *pods*, que são aglomerados de *containers* que dividem os mesmos recursos. Isso, por exemplo, já resolveria o problema da união do *HAProxy* com o *Fail2Ban*. Ademais, o formato de instanciação de serviços é declarativo e com uma escalabilidade muito maior que de *Docker Swarm*. Não obstante, a curva de aprendizado é maior que a de *Docker Swarm* e a recomendação de uso é para aplicações complexas e de dimensões superiores em relação às apresentadas neste trabalho.

# REFERÊNCIAS


ALASEM, A. An overview of e-government metadata standards and initiatives based on dublin core. **EJEG**, v. 7, n. 1, 2009.

ALBERTONI, R. et al. **Data Catalog Vocabulary (DCAT) - Version 3**. 2024. Disponível em: https://www.w3.org/TR/vocab-dcat-3/. Acesso em: 28 dez.2024.

ANNELLA, R.; MCKINNEY, J. **vCard Ontology - for describing People and Organizations**. 2024. Disponível em: https://www.w3.org/TR/vcard-rdf/. Acesso em: 28 dez.2024.

Apache Software Foundation. **Apache Solr Reference Guide**. [S.l.], 2022. Disponível em: https://solr.apache.org/guide/solr/9_0/.

BENOFF, E.; BACA, M.; HARPING, P. **Categories for the Description of Works of Art**. Los Angeles: J. Paul Getty Trust, 2024. Revised edition; licensed under a Creative Commons Attribution 4.0 International License. Disponível em: https://www.getty.edu/publications/categories-description-works-art/.

BERNERS-LEE, T.; FIELDING, R. T.; MASINTER, L. M. **Uniform Resource Identifier (URI): Generic Syntax**. RFC Editor, 2005. RFC 3986. (Request for Comments, 3986). Disponível em: https://www.rfc-editor.org/info/rfc3986.

BRICKLEY, D.; MILLER, L. **FOAF Vocabulary Specification 0.99 (Paddington Edition).** 2014. Disponível em: http://xmlns.com/foaf/spec. Acesso em: 25 dez.2024.

CAMPêLO, L. R. R. R.; NETO, V. C. B. Comparando softwares gratuitos para criação de repositórios de dados abertos. **Ciência da Informação**, Instituto Brasileiro de Informação em Ciência e Tecnologia (IBICT), v. 48, n. 3, p. —, 2020. Disponível em: https://doi.org/10.18225/ci.inf.v48i3.5004.

CAR, N. J. et al. **OGC GeoSPARQL - A Geographic Query Language for RDF Data.** 2024. Disponível em: https://docs.ogc.org/is/22-047r1/22-047r1.html. Acesso em: 25 dez.2024.

CHESNEAU, B. **Gunicorn - WSGI serve**. [S.l.], 2024. Disponível em: https://docs.gunicorn.org/en/latest/index.html.

Docker Inc. **Docker Engine security**. Docker Inc., 2025. https://docs.docker.com/engine/security/. Acesso em: 25 abr. 2025. Disponível em: https://docs.docker.com/engine/security/.

Docker Inc. **Dockerfile Reference**. Docker Inc., 2025. https://docs.docker.com/reference/dockerfile/. Acesso em: 25 abr. 2025. Disponível em: https://docs.docker.com/reference/dockerfile/.

Docker Inc. **Networking overview**. Docker Inc., 2025. https://docs.docker.com/engine/network/. Disponível em: https://docs.docker.com/engine/network/.

Docker Inc. **Swarm mode**. [S.l.], 2025. Disponível em: https://docs.docker.com/engine/swarm/.

Docker Inc. **Understanding image layers**. Docker Inc., 2025. https://docs.docker.com/get-started/docker-concepts/building-images/understanding-image-layers/. Acesso em: 25 abr. 2025. Disponível em: https://docs.docker.com/get-started/docker-concepts/building-images/understanding-image-layers/.

Docker Inc. **What is a Container?** Docker Inc., 2025. https://www.docker.com/resources/what-container/. Acesso em: 25 abr. 2025. Disponível em: https://www.docker.com/resources/what-container/.

Docker Inc. **What is an image?** Docker Inc., 2025. https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-an-image/. Acesso em: 25 abr. 2025. Disponível em: https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-an-image/.

DüRST, M. J.; SUIGNARD, M. **Internationalized Resource Identifiers (IRIs)**. RFC Editor, 2005. RFC 3987. (Request for Comments, 3987). Disponível em: https://www.rfc-editor.org/info/rfc3987.

FACHIN, J.; BLATMANN, U.; CALDIN, C. TendÊncias e uso de repositÓrios de acesso aberto. **PontodeAcesso**, v. 13, p. 86, 11 2019.

Getty Research Institute. **Art & Architecture Thesaurus (AAT)**. J. Paul Getty Trust, 2021. https://www.getty.edu/research/tools/vocabularies/aat/about.html. Acesso em: 25 abr. 2025. Disponível em: https://www.getty.edu/research/tools/vocabularies/aat/about.html.

GUARINO, N. Formal ontologies and information systems. In: . [S.l.: s.n.], 1998.

HARTIG, O. et al. **RDF 1.2 Concepts and Abstract Syntax**. 2025. Disponível em: https://www.w3.org/TR/rdf12-concepts. Acesso em: 05 fev.2025.

HIGGINSON, P. L. et al. **Virtual Router Redundancy Protocol**. RFC Editor, 1998. RFC 2338. (Request for Comments, 2338). Disponível em: https://www.rfc-editor.org/info/rfc2338.

KELLOGG, G. **RDF 1.2 XML Syntax**. 2025. Disponível em: https://www.w3.org/TR/rdf12-xml/. Acesso em: 07 fev.2025.

Library of Congress. **VRA Core 4.0 Element Description and Tagging Examples**. Library of Congress, 2007. https://www.loc.gov/standards/vracore/VRA_Core4_Element_Description.pdf. Disponível em: https://www.loc.gov/standards/vracore/VRA_Core4_Element_Description.pdf.

Library of Congress. **VRA Core: A Data Standard for the Description of Works of Visual Culture**. Library of Congress, 2022. Acesso em: 25 abr. 2025. Disponível em: https://www.loc.gov/standards/vracore/.

Linux Containers. **Introdução ao LinuX Containers (LXC)**. Linux Containers, 2025. https://linuxcontainers.org/lxc/introduction/. Acesso em: 25 abr. 2025. Disponível em: https://linuxcontainers.org/lxc/introduction/.

MACK, J.; CASSEN, A.; ARMITAGE, Q. **keepalived.conf - configuration file for Keepalived**. [S.l.], 2022. Disponível em: https://www.keepalived.org/manpage.html.

MEKONNEN, D. et al. **Quantities, Units, Dimensions and Types (QUDT) Schema - Version 2.1.46**. 2024. Disponível em: https://www.qudt.org/doc/DOC_SCHEMA-QUDT.html. Acesso em: 25 dez.2024.

Michael Kerrisk. **cgroups(7) - Linux manual page**. 2025. https://man7.org/linux/man-pages/man7/cgroups.7.html. Acesso em: 26 abr. 2025.

Michael Kerrisk. **namespaces(7) - Linux manual page**. 2025. https://man7.org/linux/man-pages/man7/namespaces.7.html. Acesso em: 26 abr. 2025.

NFS: Network File System Protocol specification. RFC Editor, 1989. RFC 1094. (Request for Comments, 1094). Disponível em: https://www.rfc-editor.org/info/rfc1094.

Open Knowledge Foundation. **User Guide**. 2.11.2. ed. [S.l.], 2022. Acesso em: 25 abr. 2025. Disponível em: https://docs.ckan.org/en/2.11/user-guide.html.

Open Knowledge Foundation. **Background jobs**. CKAN, 2025. Acesso em: 25 abr. 2025. Disponível em: https://docs.ckan.org/en/2.11/maintaining/background-tasks.html.

Open Knowledge Foundation. **Command Line Interface (CLI)**. [S.l.], 2025. Disponível em: https://docs.ckan.org/en/2.11/maintaining/cli.html.

Open Knowledge Foundation. **DataStore extension**. CKAN, 2025. https://docs.ckan.org/en/2.11/maintaining/datastore.html. Acesso em: 25 abr. 2025. Disponível em: https://docs.ckan.org/en/2.11/maintaining/datastore.html.

Open Knowledge Foundation. **Extending guide**. [S.l.], 2025. Disponível em: https://docs.ckan.org/en/2.11/extensions/index.html.

Open Knowledge Foundation. **Installing CKAN**. [S.l.], 2025. Disponível em: https://docs.ckan.org/en/2.11/maintaining/installing/index.html.

Open Knowledge Foundation. **Plugin interfaces reference**. [S.l.], 2025. Disponível em: https://docs.ckan.org/en/2.11/extensions/plugin-interfaces.html.

Open Knowledge Foundation. **Writing Profiles**. Open Knowledge Foundation, 2025. https://docs.ckan.org/projects/ckanext-dcat/en/latest/writing-profiles/. Disponível em: https://docs.ckan.org/projects/ckanext-dcat/en/latest/writing-profiles/.

Open Knowledge Foundation. **DCAT <-> CKAN Mapping**. s.d. Disponível em: https://docs.ckan.org/projects/ckanext-dcat/en/latest/mapping/. Acesso em: 25 dez.2024.

Pallets Organization. **Flask Documentation**. [S.l.], s.d. Disponível em: https://flask.palletsprojects.com/en/stable/.

PATRA, C. Digital repository in ceramics: a metadata study. **The Electronic Library**, Emerald Group Publishing Limited, v. 26, n. 4, p. 561–581, 2008. ISSN 0264-0473. Disponível em: https://doi.org/10.1108/02640470810893792.

PERREAULT, S. **vCard Format Specification**. RFC Editor, 2011. RFC 6350. (Request for Comments, 6350). Disponível em: https://www.rfc-editor.org/info/rfc6350.

PRANDI, M. B. R. **Organização do acervo da artista plástica Odilla Mestriner: um estudo de caso**. Monografia (Graduação em Ciências da Informação e da Documentação) — Universidade de São Paulo, Faculdade de Filosofia, Ciências e Letras de Ribeirão Preto, Ribeirão Preto, 2011.

Public Knowledge Project. **Open Journal Systems**. [S.l.], 2025. Disponível em: https://pkp.sfu.ca/software/ojs/.

ROCHA, R. P. d. et al. Análise dos sistemas dspace e dataverse para repositórios de dados de pesquisa com acesso aberto. **Revista Brasileira de Biblioteconomia e Documentação**, v. 17, p. 1–25, jun 2021. Disponível em: https://rbbd.febab.org.br/rbbd/article/view/1572.

SOARES, H. R. **O uso do padrão Dublin Core para a catalogação de obras de arte: Uma proposta para repositórios e museus digitais**. Monografia (Graduação em Ciências da Informação e da Documentação) — Universidade de São Paulo, Faculdade de Filosofia, Ciências e Letras de Ribeirão Preto, Ribeirão Preto, 2012.

TECHNOLOGIES, H. **HAProxy Configuration Manual**. [S.l.], 2025. Acesso em: 26 abr. 2025. Disponível em: https://www.haproxy.com/documentation/haproxy-configuration-manual/latest/.

The Dataverse Project. **Metadata Customization**. 2025. https://guides.dataverse.org/en/6.6/admin/metadatacustomization.html. Acessado em: 10 jul. 2025. Disponível em: https://guides.dataverse.org/en/5.0/admin/metadatacustomization.html.

The Dataverse Project. **Open source research data repository software**. [S.l.], 2025. Disponível em: https://dataverse.org/.

TOMASZUK, D.; HAUDEBOURG, T. **RDF 1.2 Schema**. 2025. Disponível em: https://www.w3.org/TR/rdf12-schema/. Acesso em: 07 fev.2025.

W3C SPARQL Working Group. **SPARQL 1.1 Query Language**. World Wide Web Consortium (W3C), 2013. https://www.w3.org/TR/sparql11-query/. Acesso em: 25 abr. 2025. Disponível em: https://www.w3.org/TR/sparql11-query/.

WEISS, T. R. **How to Improve Your DevOps Automation**. Docker Inc., 2025. https://www.docker.com/blog/how-to-improve-your-devops-automation/. Acesso em: 25 abr. 2025. Disponível em: https://www.docker.com/blog/how-to-improve-your-devops-automation/.

WILKINSON, M. D. et al. The fair guiding principles for scientific data management and stewardship. **Scientific data**, v. 3, n. 160018, 2016.

# APÊNDICE A – CLASSES E PROPRIEDADES NO *RDFS* (*ESQUEMA RDF*)

## A.1 CLASSES

Recursos podem ser classificados em **classes**. E membros de classe são **instâncias**. Novamente, por serem recursos, classes são identificadas por *IRIs*. Aqui, tem-se o mesmo conceito da orientação ao objeto em questão de heranças e instâncias. O que diferencia duas classes são as propriedades já que, como dito, o *RDF* define propriedades baseadas nas classes.

Todos os recursos são instâncias de *rdfs:Resource*. E *rdfs:Resource* é instância de *rdfs:Class*. Ela, por sua vez, é a classe de todas as classes e é instância dela mesma. A classe *rdfs:Literal* é a classe de valores literais. Para entender melhor esta questão de instância e classe, a string "olá" é uma instância de *rdfs:Literal*. A classe *rdfs:Datatype* é a classe dos tipos de dados. E, a última principal, *rdfs:Property* é a classe das propriedades.

## A.2 PROPRIEDADES

As **propriedades** nada mais são que os predicados do *RDF*. São utilizadas para descrever classes. A propriedade *rdfs:range* é definida como é dada a instanciação dos valores de uma propriedade. Ou seja, um valor de uma propriedade deve ser instância de uma classe especificada no *range*. O *rdfs:range* pode ter uma aplicação recursiva. O *range* de *rdfs:range* é a classe *rdfs:Class*. Logo, apenas classes podem ser valores de propriedades.

No outro sentido da hierarquia, existe a propriedade *rdfs:domain*. Ela define que uma propriedade de um recurso deve pertencer a recursos que sejam instância de uma ou mais classes. Também pode ser aplicado de forma recursiva. O domain de *rdfs:domain* é a classe *Property*, já que ela só pode pertencer/ser propriedade de propriedades. E o range de *rdfs:domain* é *rdfs:Class*, isto é, os recursos de domain são classes.

Por fim, pela importância em grafos *RDFs*, *rdf:type*. Ela denota que um recurso é instância de uma classe. Em orientação ao objeto, ela é análoga à resposta ao perguntar o tipo de uma variável. Adicionalmente, mas sem um contexto semântico definido, temos a propriedade *rdf:value*. Ele funciona para determinar o valor de um recurso quando nenhuma outra propriedade atende à semântica daquele valor.

Aplicando o que descrevemos na figura 7, podemos considerar que na tripla *Bob-"is interested in"-Mona Lisa*, Bob é instância de *rdfs:Resource*, *"is interested in"* é uma propriedade, isto é, instância de *rdf:Property* e a Mona Lisa um *rdfs:Resource*. Veja que não temos um significado formal aqui. Daí que parte a motivação de criar vocabulários que acrescentem esse valor. Baseado no *RDF* Esquema, novas subclasses que são instâncias de *rdfs:Class* são adicionadas com novas propriedades que são instâncias de *rdf:Property*. Por

exemplo, vamos verificar, através da sintaxe *Turtle*, um recurso do *namespace* do *DCAT*. O código 5 define uma instância da classe *rdf:Property*. Com a propriedade *rdfs:comment* (que não comentamos, mas pode ser encontrada na referência do *RDFS*), adiciona-se um comentário para que o usuário desse vocabulário *RDF* entenda para que serve aquela propriedade, ou seja, adiciona a semântica. O *rdfs:domain* é *dcat:Catalog*. Portanto, essa propriedade só pode estar presente na classe *dcat:Catalog*. E a propriedade *rdfs:range* informa que somente instâncias da classe *dcat:Resource*, isto é, recursos de fato em um *dataset*, podem ser atribuídas a essa propriedade como objetos na tripla *RDF*. Somente com essas propriedades dado o objetivo do *DCAT*, que já descrevemos, de organizar metadados de *datasets*, percebemos que a propriedade *dcat:resource* é utilizada para receber a *IRI* de uma instância da classe *dcat:Resource* contendo as informações de um recurso de um *dataset* para que o catálogo contendo essa propriedade tenha uma espécie de ponteiro, justamente por se tratar de um catálogo. Mais definições de triplas do vocabulário *RDF* podem ser encontradas no *namespace* do *DCAT*[1]. No capítulo 4, é visto o *DCAT* e o mapeamento dos metadados.

Código 5 – Propriedade *dcat:resource*

```
dcat:resource
  a rdf:Property ;
  a owl:ObjectProperty ;
  rdfs:comment "A resource that is listed in the catalog."@en ;
  rdfs:comment "Una risorsa elencata nel catalogo."@it ;
  rdfs:domain dcat:Catalog ;
  rdfs:isDefinedBy <https://www.w3.org/TR/vocab-dcat-3/> ;
  rdfs:label "resource"@en ;
  rdfs:label "risorsa"@it ;
  rdfs:range dcat:Resource ;
  rdfs:subPropertyOf dcterms:hasPart ;
  skos:changeNote "New property added in DCAT 3."@en ;
  skos:definition "A resource that is listed in the catalog."@en ;
  skos:definition "Una risorsa elencata nel catalogo."@it ;
  skos:editorialNote "Status: English Definition text modified by DCAT 3
      revision team, translations pending."@en ;
  skos:scopeNote "This is the most general predicate for membership of a
      catalog. Use of a more specific sub-property is recommended when
     available."@en ;
  skos:scopeNote "See also:     Sub-properties of dcat:resource in
     particular dcat:dataset, dcat:catalog, dcat:service."@en ;
.
```

[1] www.w3.org/ns/dcat3.ttl

# APÊNDICE B – CÓDIGO *CSS* QUE MODIFICA AS CORES E PÁGINA INICIAL DO *CKAN*

Código 6 – Código CSS que modifica as cores e página inicial do CKAN

```
.masthead {
        background: #d5c700 url("../../base/images/bg.png");
}
.homepage .module-search .search-form {
        border-radius: 0.25rem;
        background-color: #ef0614;
        padding: 30px 20px;
}
.masthead .main-navbar ul li a {
        padding: 0.6rem 0.9rem;
        text-decoration: none;
}
.masthead .navbar .logo img {
        max-height: 120px;
}
.account-masthead .account ul li a {
        display: block;
        text-decoration: none;
        color: #000000;
        font-size: 13px;
        font-weight: bold;
        padding: 0 10px;
        line-height: 31px;
}
.account-masthead {
        min-height: 30px;
        color: #fff;
        background: #998f00 url("../../base/images/bg.png");
}
.site-footer {
        background: #d5c700 url("../../base/images/bg.png");
        padding: 20px 0;
}
.site-footer * {
        color: #000000 !important;
}
.media-overlay {
        position: relative;
        min-height: 35px;
        display: none;
}
```

# APÊNDICE C – ARQUIVO *COMPOSE* DO *DOCKER* PARA A *STACK* DO *CKAN* E SEUS SERVIÇOS AUXILIARES

Código 7 – Arquivo *compose*

```
services:

  ckan:
    image: ckan:2.11.0
    container_name: ckan_ckan
    networks:
      - ckan-marilu
    ports:
      - 5000:5000
    env_file:
      - .env
    depends_on:
      - db
      - solr
      - redis
      - datapusher
    volumes:
      - ./persistencia/dados:/var/lib/ckan
      - /etc/localtime:/etc/localtime:ro
      - ./persistencia/bibliotecas-python/site-packages:/usr/local/lib/
        python3.10/site-packages
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "wget", "-q0", "/dev/null", "http://localhost:5000/
        api/action/status_show"]
      interval: 60s
      timeout: 10s
      retries: 3
```

Código 8 – Arquivo *compose* (continuação)

```
  datapusher:
      container_name: ckan_datapusher
      environment:
        - TZ=America/Sao_Paulo
      depends_on:
        - db
      networks:
        - ckan-marilu
      image: datapusher-plus-final
      restart: unless-stopped
      volumes:
        - ./persistencia/datapusher/.env:/srv/app/src/datapusher-plus/
            datapusher/.env
        - /etc/localtime:/etc/localtime:ro
      healthcheck:
        test: ["CMD", "wget", "-qO", "/dev/null", "http://localhost:8800"]
        interval: 60s
        timeout: 10s
        retries: 3

  db:
      container_name: ckan_postgresql
      build:
        context: postgresql/
        tags:
          - "ckan-docker-db"
      networks:
        - ckan-marilu
      environment:
        - POSTGRES_USER
        - POSTGRES_PASSWORD
        - POSTGRES_DB
        - CKAN_DB_USER
        - CKAN_DB_PASSWORD
        - CKAN_DB
        - DATASTORE_READONLY_USER
        - DATASTORE_READONLY_PASSWORD
        - DATASTORE_DB
      volumes:
        - ./persistencia/banco_dados:/var/lib/postgresql/data
        - /etc/localtime:/etc/localtime:ro
      restart: unless-stopped
      healthcheck:
        test: ["CMD", "pg_isready", "-U", "${POSTGRES_USER}", "-d", "${
            POSTGRES_DB}"]
```

Código 9 – Arquivo *compose* (continuação)

```
  solr:
      container_name: ckan_solr
      networks:
        - ckan-marilu
      image: ckan/ckan-solr:${SOLR_IMAGE_VERSION}
      volumes:
        - ./persistencia/solr/:/var/solr
        - /etc/localtime:/etc/localtime:ro
      restart: unless-stopped
      healthcheck:
        test: ["CMD", "wget", "-q0", "/dev/null", "http://localhost:8983/
            solr/"]

    redis:
      container_name: ckan_redis
      image: redis:${REDIS_VERSION}
      networks:
        - ckan-marilu
      restart: unless-stopped
      volumes:
        - /etc/localtime:/etc/localtime:ro
      healthcheck:
        test: ["CMD", "redis-cli", "-e", "QUIT"]

  networks:
    ckan-marilu:
      external: true
```

# APÊNDICE D – ARQUIVO .ENV COM AS VARIÁVEIS DE AMBIENTE DO *CKAN*

Código 10 – Arquivo .env

```
CKAN_PORT_HOST=5000
POSTGRES_USER=postgres
POSTGRES_PASSWORD=
POSTGRES_DB=postgres
POSTGRES_HOST=db
CKAN_DB_USER=ckandbuser
CKAN_DB_PASSWORD=
CKAN_DB=ckandb
DATASTORE_READONLY_USER=datastore_ro
DATASTORE_READONLY_PASSWORD=
DATASTORE_DB=datastore
CKAN_SQLALCHEMY_URL=postgresql://ckandbuser:@ckan_postgresql:5432/ckandb
CKAN_DATASTORE_WRITE_URL=postgresql://ckandbuser:@ckan_postgresql:5432/
    datastore
CKAN_DATASTORE_READ_URL=postgresql://datastore_ro:@ckan_postgresql:5432/
    datastore
TEST_CKAN_SQLALCHEMY_URL=postgres://ckandbuser:@ckan_postgresql/
    datastore_test
TEST_CKAN_DATASTORE_WRITE_URL=postgresql://ckandbuser:@ckan_postgresql
    :5432/datastore_test
TEST_CKAN_DATASTORE_READ_URL=postgresql://datastore_ro:@ckan_postgresql/
    datastore_test
CKAN_VERSION=2.11.0
CKAN_SITE_ID=default
CKAN_SITE_URL=http://10.0.2.15:5000
CKAN__LOCALE_DEFAULT=pt_BR
CKAN__AUTH__CREATE_USER_VIA_WEB=false
CKAN__HIDE_ACTIVITY_FROM_USERS=sysadmin
CKAN__BEAKER__SESSION_TIMEOUT=3600
CKAN___BEAKER__SESSION__SECRET=
CKAN___API_TOKEN__JWT__ENCODE__SECRET=
CKAN___API_TOKEN__JWT__DECODE__SECRET=
CKAN___API_TOKEN__NBYTES=342
CKAN_SYSADMIN_NAME=admin
CKAN_SYSADMIN_PASSWORD=
CKAN_SYSADMIN_EMAIL=your_email@example.com
```

Código 11 – Arquivo .env (continuação)

```
CKAN_STORAGE_PATH=/var/lib/ckan
CKAN_SMTP_SERVER=smtp.corporateict.domain:25
CKAN_SMTP_STARTTLS=True
CKAN_SMTP_USER=user
CKAN_SMTP_PASSWORD=
CKAN_SMTP_MAIL_FROM=ckan@localhost
CKAN_MAX_UPLOAD_SIZE_MB=1024
CKAN_VIEWS__DEFAULT__VIEWS=image_view datatables_view pdf_view
   officedocs_view
TZ=America/Sao_Paulo
SOLR_IMAGE_VERSION=2.10-solr9
CKAN_SOLR_URL=http://ckan_solr:8983/solr/ckan
TEST_CKAN_SOLR_URL=http://ckan_solr:8983/solr/ckan
REDIS_VERSION=6
CKAN_REDIS_URL=redis://ckan_redis:6379/1
TEST_CKAN_REDIS_URL=redis://ckan_redis:6379/1
DATAPUSHER_VERSION=0.0.20
CKAN_DATAPUSHER_URL=http://ckan_datapusher:8800
CKAN__DATAPUSHER__CALLBACK_URL_BASE=http://10.0.2.15:5000
CKAN__DATAPUSHER__FORMATS=csv xls xlsx xlsm xlsb tsv tab application/csv
    application/vnd.ms-excel application/vnd.openxmlformats-
   officedocument.spreadsheetml.sheet ods application/vnd.oasis.
   opendocument.spreadsheet jpeg jpg png jfif pdf
CKAN__DATAPUSHER__API_TOKEN=
CKANEXT__SCHEMING__DATASET_SCHEMAS=ckanext.scheming:presencas.yaml
CKAN__SCHEMING__DATASET_SCHEMAS=ckanext.scheming:presencas.yaml
CKANEXT___SCHEMING__DATASET_SCHEMAS=ckanext.scheming:presencas.yaml
CKAN___SCHEMING__DATASET_SCHEMAS=ckanext.scheming:presencas.yaml
CKANEXT__DCAT__RDF__PROFILES=presencas_dcat_3
CKAN__PLUGINS="image_view text_view datatables_view datastore datapusher
    officedocs_view pdf_view dcat structured_data scheming_datasets
   presencas envvars"
CKAN__HARVEST__MQ__TYPE=redis
CKAN__HARVEST__MQ__HOSTNAME=redis
CKAN__HARVEST__MQ__PORT=6379
CKAN__HARVEST__MQ__REDIS_DB=1
CKAN__SECRET_KEY=
```

# APÊNDICE E – FLUXO DE CADASTRO/ATUALIZAÇÃO DE ARTISTAS E OBRAS

Com a aplicação carregada, ela fica disponível na porta especificada para desenvolvimento ou via comando. No caso de rodar via *container*, ela fica disponível na porta 80. Ao acessar qualquer página, a função com o decorador cujo respectivo endereço *URI* seja igual ao solicitado retornará o que foi solicitado. Por exemplo, ao realizar o fluxo de cadastro descrito na seção 5.2.1.1, a função index_formulario_criacao_artista no arquivo *views.py* do diretório "*formularios*" é executada. Após preencher os dados, eles são recebidos e validados para o caso de campos múltiplos e para nome, trajetória e produção. Se tudo estiver correto uma solicitação é criada. Uma entrada na tabela "*solicitacoes*" é criada com o ID aleatório dado pelo cliente e seu ID de sessão sendo chaves primárias. Essa solicitação marca o progresso, o que será exibido na barra de carregamento. A cada 0.5 segundo, o cliente solicita via *AJAX* o progresso. O valor é utilizado pelo *CSS* para gerar a barra. Os dados inseridos pelo usuário inicializam uma instância da classe *Artista*, que acomoda todos os dados de um deles. Com o artista carregado, uma solicitação via *API* do *CKAN* é enviada para criar o *dataset* dele. Em caso de sucesso, para cada imagem inserida, uma solicitação via *API* também é criada para ela. A imagem enviada via formulário é adicionada no repositório do artista. Todo esse fluxo de criação começa na função *cria_artista_recursos_ckan* do arquivo *criador.py* do diretório formularios. A adição de novas obras para artistas já existentes é similar, porém a etapa de criar o *dataset* do artista é desconsiderada. A validação tem um passo adicional, que é verificar se o artista existe. Em todos os casos, se ocorrer algum erro, a resposta do *CKAN* é retornada com o respectivo código. Se o *CKAN* estiver indisponível, o código 502 de *Bad Gateway* é retornado. Se a solicitação não puder ser criada ou qualquer erro ocorrer no fluxo da nossa aplicação, é retornado 500. Caso o artista e/ou suas obras sejam criadas com sucesso, é retornado 201 de *Created*.

# APÊNDICE F – TIPOS E CAMPOS DO ESQUEMA DO *CKAN* NO *APACHE SOLR*

Código 12 – Tipos do esquema do *CKAN* no *Apache Solr*

```
<types>
    <fieldType name="data_presencas" class="solr.DateRangeField"/>
    <fieldType name="string" class="solr.StrField" sortMissingLast="true
       " omitNorms="true"/>
    <fieldType name="boolean" class="solr.BoolField" sortMissingLast="
       true" omitNorms="true"/>
    <fieldtype name="binary" class="solr.BinaryField"/>
    <fieldType name="int" class="solr.IntPointField" omitNorms="true"
       positionIncrementGap="0"/>
    <fieldType name="float" class="solr.FloatPointField" omitNorms="true
       " positionIncrementGap="0"/>
    <fieldType name="long" class="solr.LongPointField" omitNorms="true"
       positionIncrementGap="0"/>
    <fieldType name="double" class="solr.DoublePointField" omitNorms="
       true" positionIncrementGap="0"/>
    <fieldType name="pint" class="solr.IntPointField" omitNorms="true"
       positionIncrementGap="0"/>
    <fieldType name="pfloat" class="solr.FloatPointField" omitNorms="
       true" positionIncrementGap="0"/>
    <fieldType name="plong" class="solr.LongPointField" omitNorms="true"
        positionIncrementGap="0"/>
    <fieldType name="pdouble" class="solr.DoublePointField" omitNorms="
       true" positionIncrementGap="0"/>
    <fieldType name="date" class="solr.DatePointField" omitNorms="true"
       positionIncrementGap="0"/>
    <fieldType name="pdate" class="solr.DatePointField" omitNorms="true"
        positionIncrementGap="0"/>

    <fieldType name="pdates" class="solr.DatePointField"
       positionIncrementGap="0" multiValued="true"/>
    <fieldType name="booleans" class="solr.BoolField" sortMissingLast="
       true" multiValued="true"/>
    <fieldType name="pints" class="solr.IntPointField"
       positionIncrementGap="0" multiValued="true"/>
    <fieldType name="pfloats" class="solr.FloatPointField"
       positionIncrementGap="0" multiValued="true"/>
    <fieldType name="plongs" class="solr.LongPointField"
       positionIncrementGap="0" multiValued="true"/>
    <fieldType name="pdoubles" class="solr.DoublePointField"
       positionIncrementGap="0" multiValued="true"/>
```

Código 13 – Tipos do esquema do *CKAN* no *Apache Solr* (continuação)

```
    <fieldType name="text" class="solr.TextField" positionIncrementGap="
        100">
        <analyzer type="index">
            <tokenizer class="solr.WhitespaceTokenizerFactory"/>
            <filter class="solr.WordDelimiterGraphFilterFactory"
                generateWordParts="1" generateNumberParts="1"
                catenateWords="1" catenateNumbers="1" catenateAll="0"
                splitOnCaseChange="1"/>
            <filter class="solr.FlattenGraphFilterFactory"/>
            <filter class="solr.LowerCaseFilterFactory"/>
            <filter class="solr.SnowballPorterFilterFactory" language="
                English" protected="protwords.txt"/>
            <filter class="solr.ASCIIFoldingFilterFactory"/>
        </analyzer>
        <analyzer type="query">
            <tokenizer class="solr.WhitespaceTokenizerFactory"/>
            <filter class="solr.SynonymGraphFilterFactory" synonyms="
                synonyms.txt" ignoreCase="true" expand="true"/>
            <filter class="solr.WordDelimiterGraphFilterFactory"
                generateWordParts="1" generateNumberParts="1"
                catenateWords="0" catenateNumbers="0" catenateAll="0"
                splitOnCaseChange="1"/>
            <filter class="solr.LowerCaseFilterFactory"/>
            <filter class="solr.SnowballPorterFilterFactory" language="
                English" protected="protwords.txt"/>
            <filter class="solr.ASCIIFoldingFilterFactory"/>
        </analyzer>
    </fieldType>
```

Código 14 – Tipos do esquema do *CKAN* no *Apache Solr* (continuação)

```
    <fieldType name="text_general" class="solr.TextField"
       positionIncrementGap="100">
        <analyzer type="index">
            <tokenizer class="solr.WhitespaceTokenizerFactory"/>
            <filter class="solr.WordDelimiterGraphFilterFactory"
               generateWordParts="1" generateNumberParts="1"
               catenateWords="1" catenateNumbers="1" catenateAll="0"
               splitOnCaseChange="0"/>
            <filter class="solr.FlattenGraphFilterFactory"/> <!--
               required on index analyzers after graph filters -->
            <filter class="solr.LowerCaseFilterFactory"/>
        </analyzer>
        <analyzer type="query">
            <tokenizer class="solr.WhitespaceTokenizerFactory"/>
            <filter class="solr.SynonymGraphFilterFactory" synonyms="
               synonyms.txt" ignoreCase="true" expand="true"/>
            <filter class="solr.WordDelimiterGraphFilterFactory"
               generateWordParts="1" generateNumberParts="1"
               catenateWords="0" catenateNumbers="0" catenateAll="0"
               splitOnCaseChange="0"/>
            <filter class="solr.LowerCaseFilterFactory"/>
        </analyzer>
    </fieldType>

    <fieldType name="text_ngram" class="solr.TextField"
       positionIncrementGap="100">
      <analyzer type="index">
        <tokenizer class="solr.NGramTokenizerFactory" minGramSize="2"
           maxGramSize="10"/>
        <filter class="solr.LowerCaseFilterFactory"/>
      </analyzer>
      <analyzer type="query">
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <filter class="solr.LowerCaseFilterFactory"/>
      </analyzer>
    </fieldType>

</types>
```

Código 15 – Campos do esquema do *CKAN* no *Apache Solr*

```
<fields>
    <field name="index_id" type="string" indexed="true" stored="true"
       required="true" />
    <field name="id" type="string" indexed="true" stored="true" required
       ="true" />
    <field name="site_id" type="string" indexed="true" stored="true"
       required="true" />
    <field name="title" type="text" indexed="true" stored="true" />
    <field name="title_ngram" type="text_ngram" indexed="true" stored="
       true" />
    <field name="entity_type" type="string" indexed="true" stored="true"
        omitNorms="true" />
    <field name="dataset_type" type="string" indexed="true" stored="true
       " />
    <field name="state" type="string" indexed="true" stored="true"
       omitNorms="true" />
    <field name="name" type="string" indexed="true" stored="true"
       omitNorms="true" />
    <field name="name_ngram" type="text_ngram" indexed="true" stored="
       true" />
    <field name="revision_id" type="string" indexed="true" stored="true"
        omitNorms="true" />
    <field name="version" type="string" indexed="true" stored="true" />
    <field name="url" type="string" indexed="true" stored="true"
       omitNorms="true" />
    <field name="ckan_url" type="string" indexed="true" stored="true"
       omitNorms="true" />
    <field name="download_url" type="string" indexed="true" stored="true
       " omitNorms="true" />
    <field name="notes" type="text" indexed="true" stored="true"/>
    <field name="author" type="text_general" indexed="true" stored="true
       " />
    <field name="author_email" type="text_general" indexed="true" stored
       ="true" />
    <field name="maintainer" type="text_general" indexed="true" stored="
       true" />
    <field name="maintainer_email" type="text_general" indexed="true"
       stored="true" />
    <field name="license" type="string" indexed="true" stored="true" />
    <field name="license_id" type="string" indexed="true" stored="true"
       />
    <field name="tags" type="string" indexed="true" stored="true"
       multiValued="true"/>
    <field name="groups" type="string" indexed="true" stored="true"
       multiValued="true"/>
    <field name="organization" type="string" indexed="true" stored="true
       " multiValued="false"/>
```

Código 16 – Campos do esquema do *CKAN* no *Apache Solr* (continuação)

```
    <field name="capacity" type="string" indexed="true" stored="true"
       multiValued="false"/>
    <field name="permission_labels" type="string" indexed="true" stored=
       "false" multiValued="true"/>

    <field name="res_name" type="text_general" indexed="true" stored="
       true" multiValued="true" />
    <field name="res_description" type="text_general" indexed="true"
       stored="true" multiValued="true"/>
    <field name="res_format" type="string" indexed="true" stored="true"
       multiValued="true"/>
    <field name="res_url" type="string" indexed="true" stored="true"
       multiValued="true"/>
    <field name="res_type" type="string" indexed="true" stored="true"
       multiValued="true"/>
    <field name="text" type="text" indexed="true" stored="false"
       multiValued="true"/>
    <field name="urls" type="text" indexed="true" stored="false"
       multiValued="true"/>

    <field name="depends_on" type="text" indexed="true" stored="false"
       multiValued="true"/>
    <field name="dependency_of" type="text" indexed="true" stored="false
       " multiValued="true"/>
    <field name="derives_from" type="text" indexed="true" stored="false"
        multiValued="true"/>
    <field name="has_derivation" type="text" indexed="true" stored="
       false" multiValued="true"/>
    <field name="links_to" type="text" indexed="true" stored="false"
       multiValued="true"/>
    <field name="linked_from" type="text" indexed="true" stored="false"
       multiValued="true"/>
    <field name="child_of" type="text" indexed="true" stored="false"
       multiValued="true"/>
    <field name="parent_of" type="text" indexed="true" stored="false"
       multiValued="true"/>
    <field name="views_total" type="int" indexed="true" stored="false"/>
    <field name="views_recent" type="int" indexed="true" stored="false"/
       >
    <field name="resources_accessed_total" type="int" indexed="true"
       stored="false"/>
    <field name="resources_accessed_recent" type="int" indexed="true"
       stored="false"/>
```

Código 17 – Campos do esquema do *CKAN* no *Apache Solr* (continuação)

```
      <field name="metadata_created" type="date" indexed="true" stored="
         true" multiValued="false"/>
      <field name="metadata_modified" type="date" indexed="true" stored="
         true" multiValued="false"/>
      <field name="indexed_ts" type="date" indexed="true" stored="true"
         default="NOW" multiValued="false"/>
      <field name="title_string" type="string" indexed="true" stored="
         false" />
      <field name="data_dict" type="string" indexed="false" stored="true"
         />
      <field name="validated_data_dict" type="string" indexed="false"
         stored="true" />
      <field name="_version_" type="string" indexed="true" stored="true"/>
      <dynamicField name="*_date" type="date" indexed="true" stored="true"
          multiValued="false"/>
      <dynamicField name="extras_*" type="text" indexed="true" stored="
         true" multiValued="false"/>
      <dynamicField name="res_extras_*" type="text" indexed="true" stored=
         "true" multiValued="true"/>
      <dynamicField name="vocab_*" type="string" indexed="true" stored="
         true" multiValued="true"/>
      <field name="cidade" type="string" indexed="true" stored="true"/>
      <field name="estado" type="string" indexed="true" stored="true"/>
      <field name="cidade_atuacao" type="string" indexed="true" stored="
         true" multiValued="true"/>
      <field name="estado_atuacao" type="string" indexed="true" stored="
         true" multiValued="true"/>
      <field name="pais_atuacao" type="string" indexed="true" stored="true
         " multiValued="true"/>
      <field name="data_inicio" type="data_presencas" indexed="true"
         stored="true"/>
      <field name="linguagens" type="string" indexed="true" stored="true"
         multiValued="true"/>
      <dynamicField name="*" type="string" indexed="true"  stored="false"/
         >
  </fields>
```

# APÊNDICE G – PROTOCOLOS *VRRP* E *NFS*

## G.1 PROTOCOLO *VRRP*

Iremos ter apenas uma máquina com o *proxy reverso*. Como apenas uma irá contê-lo e estamos utilizando *Docker Swarm* para construir o *cluster*, o *proxy* reverso irá redirecionar para os *containers* dos serviços. Poderíamos fazer apenas uma máquina com balanceamento de carga. No entanto, queremos manter a consistência do *proxy* também, assim como para o *CKAN* e a aplicação externa. Logo, é preciso haver sempre um endereço para o usuário final, apesar do *proxy* poder estar em quaisquer uma das máquinas. Para isso, podemos utilizar o *Keepalived*, que é uma implementação do protocolo *VRRP*.

O protocolo VRRP (HIGGINSON et al., 1998) está na camada de rede do modelo OSI e tem como princípio a confiabilidade de uma rede ao mantê-la constante e consistente. Um conjunto de máquinas irá possuir um roteador virtual (**virtual router**). Esse roteador virtual irá, obviamente, conter um endereço ao menos, que chamamos de virtual IP ou VIP. Esse endereço será atribuído, a princípio, à interface que está se comunicando com outras máquinas através do protocolo. Baseado em prioridade, aquele que tiver a maior terá a posse desse roteador virtual e, portanto, o endereço virtual. A essa máquina com a posse atribui-se o nome de *MASTER* (mestre). Às demais atribui-se o nome de *BACKUP* (reforços). Cada máquina inicia-se em um desses estados baseado em configurações de prioridade. Há também o estado de inicialização (*INITIALIZE*) ou erro (*FAULT*). Nesse caso, existe algum problema de configuração ou alguma interface envolvida na troca de pacotes *VRRP* está comprometida. É possível que existam várias máquinas *MASTER*, mas somente uma possui o endereço virtual. O que decide a posse é justamente a prioridade como mencionada.

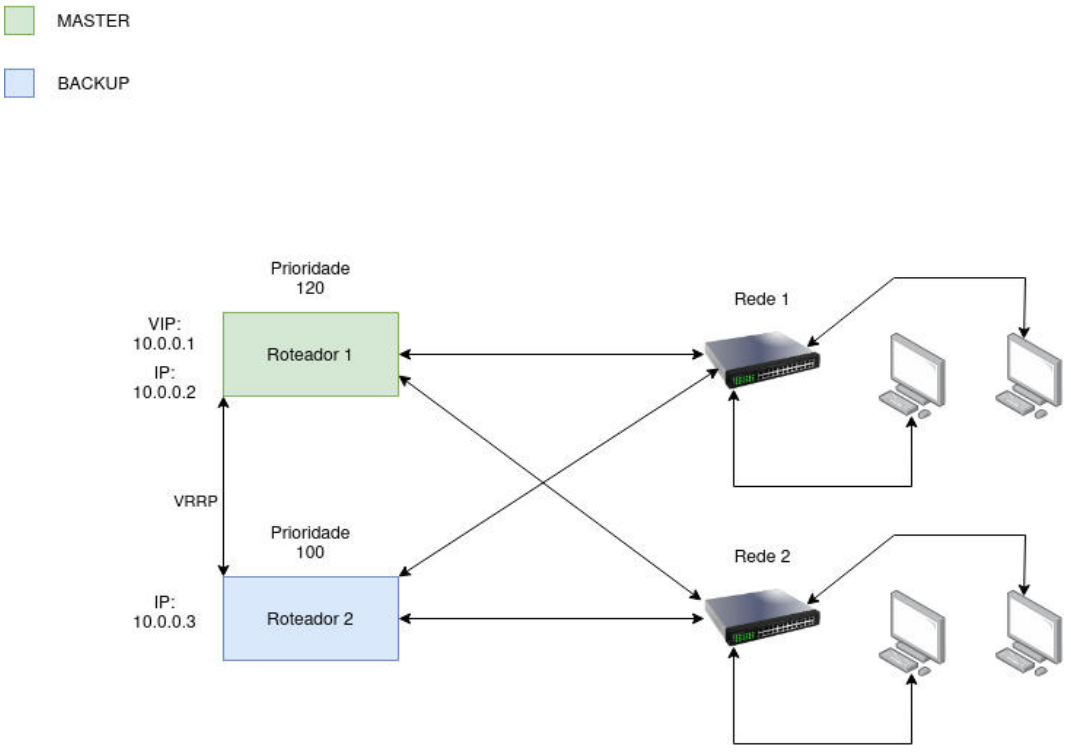
A cada intervalo de tempo definido, que chamamos de *advert int* (tempo de notificação), as máquinas enviam pacotes *VRRP* para verificar disponibilidade e prioridade. Esse tempo, geralmente, é de um segundo. Cada máquina irá checar a prioridade atual com a prioridade recebida no pacote. Se ela for maior (menor), ela se torna *MASTER* (*BACKUP*). Logo, irá possuir o VIP. Portanto, para nossa estrutura de *proxy* reverso único, isto é fundamental. Um usuário deverá conectar-se a apenas um endereço. Como o HAProxy pode estar em qualquer uma das máquinas, cada uma com seu endereço IP, é preciso que exista apenas um. E, dessa forma, o protocolo *VRRP* nos ajuda nesse objetivo.

A vantagem de utilizar esta abordagem em relação a várias máquinas com *HAProxy* com balanceamento é que, caso uma delas não esteja disponível, o usuário pode receber um aviso de impossibilidade de conexão, pois o endereço dessa máquina retornado na consulta *DNS* tornou-se indisponível. Consequentemente, apenas uma máquina contendo

o *HAProxy* e que seja eleita com base na disponibilidade do protocolo *VRRP* permite que o usuário sempre acredite que o serviço esteve disponível e sem interrupções.

A figura 36 contém o diagrama do protocolo para duas redes internas conectadas a dois roteadores. Normalmente, haveria apenas um roteador. Se esse sofresse alguma interrupção, toda a rede interna ficaria inacessível. No entanto, com dois roteadores trocando informações *VRRP* e configurando um IP virtual com base em prioridades (o roteador 1 possui uma maior que o 2), em caso de falha de um não ocorre indisponibilidade alguma. Já na figura 37, temos a ideia desse projeto. A presença do *HAProxy* na máquina orienta que a máquina deve possuir o endereço virtual. O *Keepalived*, aplicação que implementa o *VRRP*, apresenta muitas possíveis configurações para um hospedeiro. Dentre elas, a capacidade de aumentar ou diminuir a prioridade com base na execução de um processo. No caso, o processo do *HAProxy* aumenta a prioridade de forma que os demais entrem em estado *BACKUP*. Com isso, a máquina com *HAProxy* torna-se *MASTER*. Todos os endereços de serviços nesse *cluster* devem resolver para o endereço virtual e, portanto, irão atingir a máquina.

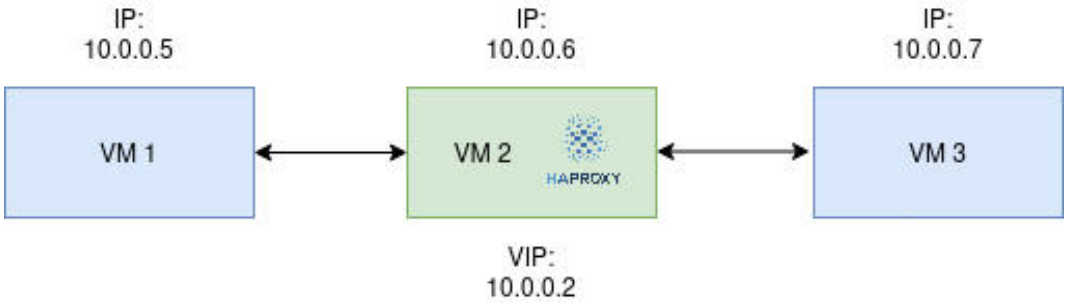
Figura 36 – Diagrama do protocolo *VRRP* com duas redes internas e dois roteadores




O roteador 1 possui o endereço virtual por ter maior prioridade

Fonte: Elaboração própria

Figura 37 – Diagrama do protocolo *VRRP* com três máquinas de um *cluster*




A presença do HAProxy norteia a posse do endereço virtual

Fonte: Elaboração própria

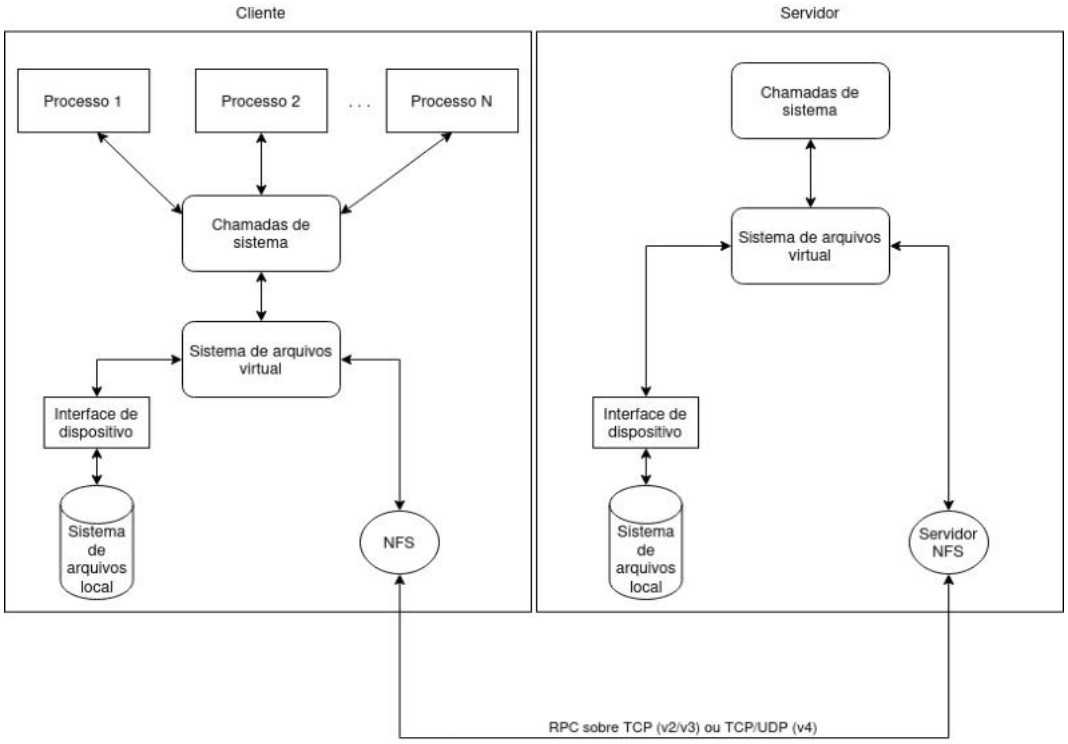
## G.2 PROTOCOLO *NFS*

O *NFS* (*Network File System*) é um protocolo da camada de aplicação do modelo OSI, definido na RFC 1094 (NFS..., 1989) que serve para montar sistemas de arquivos de um servidor em clientes que os requisitarem. O cliente passa a utilizar o sistema como se ele fosse um sistema de arquivos local. Como o próprio nome sugere, é um protocolo que atua com conexões de redes e no modelo cliente-servidor. Dessa forma, é possível centralizar os recursos em um ponto. As versões iniciais 2 e 3 utilizavam comunicação remota procedural (*RPC*), já a versão 4 utiliza apenas os protocolos *TCP* e *UDP*.

Um servidor disponibiliza diretórios com base no serviço de montagem que faz parte do protocolo. Apenas clientes autorizados podem acessar os diretórios disponibilizados. Como existe o problema de permissão de leitura e escrita, é preciso alguns cuidados. No nosso caso, o *NFS* é basicamente utilizado pelo *Docker*. Existe uma opção que permite que o usuário administrador de sistemas *Linux*, *root*, mantenha seus privilégios. Dessa forma, os diretórios raízes para os *containers* podem ser criados através deles. E, com isso, disponibilizados para a escrita de quaisquer usuários/grupos dos processos dos *containers*. Alternativamente, pode-se usar o protocolo *LDAP* para atribuir usuários a grupos e disponibilizar diretórios com permissões específicas a esses grupos. E uma abordagem comum e recomendada nos manuais de *NFS* na versão 2 ou 3 é a de tornar os diretórios pertencentes ao usuário *nobody* e ao grupo *nogroup*. Quando o protocolo não consegue mapear o usuário e o grupo da máquina cliente para a servidora, eles são mapeados para ambos descritos anteriormente. Ou seja, cria-se um acesso anônimo. Dessa forma, se um diretório tem a posse de usuário para *nobody* e de grupo para *nogroup*, qualquer usuário e grupo pode acessá-lo. Na figura 38 está um diagrama muito simplificado do protocolo. Onde foi representado o sistema de arquivos local, é possível que os diretórios estivessem em dispositivos de armazenamento em uma *SAN*, por exemplo. Logo, não seriam locais. Também foram abstraídas especificações e particularidades da arquitetura *Unix*

no diagrama, bem como a interação correta em nível de camada de rede e aplicação e as chamadas procedurais.

Figura 38 – Diagrama simplificado do protocolo *NFS*




Fonte: Elaboração própria

# APÊNDICE H – CONFIGURAÇÃO DE SERVIÇO E REGRAS PARA O *IPTABLES*

Código 18 – Configuração para o serviço *systemd* que cria/remove as regras do *IPTables*

```
[Unit]
Description=Restaura regras personalizadas do iptables
After=docker.service
PartOf=docker.service
[Service]
Type=oneshot
ExecStart=/usr/local/sbin/adiciona-regras-iptables.sh
RemainAfterExit=yes
[Install]
WantedBy=multi-user.target
```

Código 19 – *Script* que adiciona as regras para os nós do *cluster swarm*

```
#!/bin/bash

# Regras para loopback, conexões estabelecidas e relacionadas, SSH pela
    199, portas TCP/UDP para o Docker Swarm e VRRP pela 199

iptables -P INPUT ACCEPT
iptables -F INPUT
iptables -A INPUT -i lo -j ACCEPT
iptables -A INPUT -s 10.199.0.0/16 -p tcp -m tcp --dport 22022 -j ACCEPT
iptables -A INPUT -p tcp -m multiport --dports 7946,2377 -j ACCEPT
iptables -A INPUT -p udp -m multiport --dports 4789,7946 -j ACCEPT
iptables -A INPUT -s 10.199.0.0/16 -p vrrp -j ACCEPT
iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
iptables -P INPUT DROP
```

Código 20 – *Script* que adiciona as regras para o servidor *NFS*

```
#!/bin/bash

# Regras para servidor NFS, RPC e SSH na 199

iptables -P INPUT ACCEPT
iptables -F INPUT
iptables -A INPUT -i lo -j ACCEPT
iptables -A INPUT -s 10.199.0.0/16 -p tcp -m tcp --dport 22022 -j ACCEPT
iptables -A INPUT -p tcp -m multiport --dports 111,2049 -j ACCEPT
iptables -A INPUT -p udp -m multiport --dports 111,2049 -j ACCEPT
iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
iptables -P INPUT DROP
```

# APÊNDICE I – CONFIGURAÇÃO DO *HAPROXY*

Código 21 – Configuração do *HAProxy*

```
defaults
        mode http
        timeout client 10s
        timeout connect 10s
        timeout server 10s

resolvers docker
        nameserver dns 127.0.0.11:53
        hold other           30s
        hold refused         30s
        hold nx              30s
        hold timeout         30s
        hold valid           10s
        hold obsolete        30s

frontend fe_http
        bind :80
        use_backend be_presencas_ckan if { hdr(host) -i ckan.tcc.ufrj.br
            || dst 10.199.8.30 }
        use_backend be_presencas_cadastros if { hdr(host) -i presencas-
           cadastros.tcc.ufrj.br }

backend be_presencas_ckan
        server ckan presencas-ckan_ckan:5000 check resolvers docker init
           -addr none

backend be_presencas_cadastros
        option forwardfor
        server cadastros presencas-cadastros_nginx:80 check resolvers
           docker init-addr none
```

# APÊNDICE J – CONFIGURAÇÃO DE FILTRO E CADEIA PARA O *FAIL2BAN* PARA PROTEÇÃO DA AUTENTICAÇÃO DA APLICAÇÃO EXTERNA, RECEITA *COMPOSE* DO *FAIL2BAN* E CONFIGURAÇÃO DO *KEEPALIVED*

Código 22 – Filtro de autenticação incorreta

```
[Definition]

failregex = \[.+\] ERROR <HOST> Nome de usuário ou senha incorretos
ignoreregex =
```

Código 23 – Cadeia para ação contra autenticação incorreta

```
## Version 2024/12/16
# Fail2Ban jail configuration for Presenćas

[DEFAULT]
bantime.increment = true

bantime.maxtime = 5w

bantime.factor = 24

bantime = 1h

findtime = 24h

maxretry = 5

[presencas]
banaction=iptables[type=multiport]
enabled = true
chain = DOCKER-USER
port = 8000,80
filter = presencas
logpath = /presencas-log/presencas.log
```

Código 24 – Receita *compose* do *Fail2Ban*

```
version: '3.9'
services:
  fail2ban:
    image: crazymax/fail2ban
    container_name: fail2ban
    cap_add:
      - NET_ADMIN
      - NET_RAW
    environment:
      - TZ=America/Sao_Paulo
    network_mode: host
    volumes:
      - /docker-volumes/presencas-cadastros/fail2ban/config:/data
      - /docker-volumes/presencas-cadastros/fail2ban/presencas.log:/
          presencas-log/presencas.log:ro
    restart: unless-stopped
```

Código 25 – *Script* para iniciar o *Fail2Ban*

```
#!/bin/bash

if [[ $(docker ps -q --filter name=haproxy) != "" && $(docker ps -q --
    filter name=fail2ban) == "" ]]
then
        docker compose -f /usr/local/bin/fail2ban/docker-compose.yml up
            -d
fi
```

Código 26 – *Script* para desativar o *Fail2Ban*

```
#!/bin/bash

if [[ $(docker ps -q --filter name=haproxy) != "" || $(docker ps -q --
    filter name=fail2ban) ]]
then
        docker compose -f /usr/local/bin/fail2ban/docker-compose.yml
            down
fi
```

Código 27 – Configuração do Keepalived no nó *MASTER*

```
global_defs {
        default_interface enX0
        enable_script_security
}
vrrp_track_process haproxy {
        process haproxy
        weight 20
}
vrrp_instance vrrp_roteador {
        state MASTER
        interface enX0
        track_process {
                haproxy
        }
        virtual_router_id 220
        priority 100
        virtual_ipaddress {
                10.199.8.30/16
        }
        notify_master   "/usr/local/bin/fail2ban/executa-fail2ban.sh"
           root
        notify_backup   "/usr/local/bin/fail2ban/desliga-fail2ban.sh"
           root
}
```