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

Reestruturação dos ambientes de desenvolvimento e produção do portal CulturaEduca

Arthur Coser Marinho

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Tiago Silva da Silva

São Paulo
2022

Agradecimentos

Agradeço à equipe do CulturaEduca por esse ano de trabalho conjunto no projeto, aos meus professores que guiaram meu aprendizado durante toda essa jornada, aos meus sócios que enfrentaram as dificuldades comigo lado a lado, aos meus amigos que me apoiaram todo esse tempo e à minha família que sempre me deu suporte incondicional.

Resumo

Arthur Coser Marinho. **Reestruturação dos ambientes de desenvolvimento e produção do portal CulturaEduca**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

O sistema do CulturaEduca, portal para visualização de dados georreferenciados de educação e sócio-demográficos, teve seu desenvolvimento parado por quatro anos. A falta de atualizações das dependências do sistema em conjunto com a falta de documentação contida no projeto dificultaram o desenvolvimento de melhorias. Este trabalho tem por objetivo a reestruturação dos ambientes de desenvolvimento e produção do sistema, de modo a possibilitar o desenvolvimento de novas funcionalidades, consertos de *bugs* e atualização das versões das dependências. Utilizando uma arquitetura baseada em *containers*, uma das principais tecnologias utilizadas na atualidade nas práticas de *DevOps*, realizamos o isolamento das aplicações que compõem o sistema. Foi escolhido o *Docker* como plataforma de gerenciamento de *containers* por conta de sua popularidade e gama de funcionalidades que abrangem todas as necessidades do projeto. A *"containerização"* do ambiente de produção inicialmente facilitou a realização de uma das demandas iniciais de migração do servidor que hospeda o portal, assim como a criação de um ambiente de homologação para validação de novas funcionalidades. Em seguida, a *"containerização"* do ambiente de desenvolvimento tornou o processo de configuração da máquina local dos desenvolvedores muito mais rápido, facilitando a entrada de novos desenvolvedores no projeto e a aplicação de atualizações nas dependências do sistema. Porém o processo de atualização das versões das dependências tomou mais tempo que o esperado e evidenciou o problema de negligenciar sua realização ao longo do tempo.

Palavras-chave: Ambiente de desenvolvimento. Ambiente de produção. DevOps. Container. Docker. Dependências.

Abstract

Arthur Coser Marinho. **Restructuring of the development and production environments of the CulturaEduca portal**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

The CulturaEduca system, a portal for viewing georeferenced education and socio-demographic data, had its development halted for four years. The lack of updates to the system dependencies along with the lack of documentation in the project made it difficult to develop improvements. This work aims to restructure the development and production environments of the system, in order to enable the development of new features, bug fixes and update versions of the dependencies. Using an architecture based on containers, one of the main technologies used today in DevOps practices, we isolate the applications that make up the system. Docker was chosen as the container management platform due to its popularity and range of features that cover all the project needs. The containerization of the production environment initially facilitated the accomplishment of one of the initial demands of migration of the server that hosts the portal, as well as the creation of a staging environment for the validation of new functionalities. Then, the containerization of the development environment made the process of configuring the developers' local machine much faster, making it easier for new developers to join the project and to apply updates to system dependencies. However, the process of updating the dependencies versions took longer than expected and showed the problem of neglecting its implementation over time.

Keywords: Development environment. Production environment. DevOps. Container. Docker. Dependencies.

Lista de Figuras

Sumário

Capítulo 1

Introdução

1.1 Contextualização

Este trabalho faz parte do desenvolvimento do CulturaEduca (https://culturaeduca.cc/), um portal colaborativo para visualização de dados georreferenciados de educação e sócio-demográficos.

O CulturaEduca foi criado a partir do projeto "Mapeamento das Iniciativas de Cultura e Educação", convênio estabelecido entre o Instituto Lidas[1] e a Secretaria de Políticas Culturais do Ministério da Cultura (SPC/MinC) no fim de 2011.

Conforme o texto original dos responsáveis do CulturaEduca, o objetivo do projeto foi mapear iniciativas que promovem a interface entre cultura e educação, contribuindo para a troca de metodologias e práticas no contexto da implementação de um política de cultura para educação. Mais especificamente, a construção de um portal *web* que possuísse um módulo de mapas para a visualização e interação com informações georreferenciadas, fórum virtual para troca de metodologias e espaço para construção de um acervo digital.

É descrito que o Instituto Lidas realizou a coleta e a padronização de bases de dados de equipamentos públicos (escolas, museus, bibliotecas, Centros de Referência da Assistência Social, Unidades Básicas de Saúde etc.) reunidas através de contatos junto a órgãos públicos, o que possibilitou a sobreposição e o cruzamento de dados provenientes de diversas fontes (IBGE, INEP etc) em um só portal.

Apontam também que os mapas apresentam dados sobre as particularidades de cada território: demografia, renda, escolaridade, entre outros. Essas informações são estratégicas para embasar o planejamento de ações comunitárias e/ou de políticas públicas, como também para constituírem uma avaliação diagnóstica dos respectivos territórios para a construção dos próprios projetos políticos pedagógicos das escolas públicas, fornecendo

[1] O Instituto Lidas é uma Associação de Direito Privado sem fins lucrativos fundada em 1990 que tem como finalidades a produção de informação e conhecimentos sobre o espaço urbano para subsídio de ações de organizações e movimentos sociais e para políticas públicas, bem como a produção de metodologias de análise e intervenção territorial para promoção do pertencimento e apropriação do local de vivência pelos cidadãos.

também subsídios para o desenvolvimento do currículo em diálogo com a realidade da comunidade.

Em 2017, um Acordo de Cooperação Técnica entre o Instituto Lidas e a Faculdade de Educação da Universidade de São Paulo (FEUSP) foi aprovado pelo Conselho de Cultura e Extensão Universitária, objetivando a divulgação e uso do Portal CulturaEduca entre licenciandos, docentes e colaboradores da rede pública, no âmbito do Programa de Formação de Professores[2]. Desde então o portal vêm apoiando iniciativas de mapeamento colaborativo do território de escolas públicas parceiras do Núcleo de Avaliação Institucional da FEUSP (NAI-FEUSP), fomentando o acesso à dados e informações georreferenciadas potencialmente relevantes aos projetos de estágio, aos projetos de pesquisa e aos projetos pedagógicos, subsidiando, assim, a formulação de políticas públicas educacionais[3].

No início de 2021, no âmbito dos projetos do NAI-FEUSP, iniciou-se uma nova parceria com o Instituto de Matemática e Estatística (IME-USP) e o Instituto de Ciência e Tecnologia da Universidade Federal de São Paulo (ICT-UNIFESP), no qual este trabalho se insere. Essa parceria tem como objetivo levantar requisitos para consolidar o portal CulturaEduca como uma ferramenta de apoio à elaboração de políticas públicas na área de educação, além de melhorias técnicas identificadas na implementação do sistema. Foi então formada uma equipe composta de alunos de graduação supervisionada por professores para execução dessas melhorias.

O código-fonte do projeto está disponível em: https://gitlab.com/ccsl-usp/culturaeduca.

1.2 Motivação e objetivos

O desenvolvimento do sistema se iniciou de fato em novembro de 2014 e, após uma atualização da versão do *framework* em abril de 2015, não foi feita nenhuma atualização de suas dependências desde então. Não foi feita também nenhuma documentação de como configurar os ambientes de desenvolvimento e produção.

Além dos riscos de segurança associados, não atualizar as dependências de um sistema dificulta sua manutenção e novos desenvolvimentos. E, combinada com a falta de documentação, fica cada vez mais difícil garantir que os ambientes de desenvolvimento e produção estejam consistentes entre diferentes máquinas.

O objetivo deste trabalho é reestruturar os ambientes de desenvolvimento e produção do CulturaEduca de modo a possibilitar o desenvolvimento de correções pendentes e de novas funcionalidades, ao mesmo tempo que realizar gradualmente a atualização das tecnologias utilizadas no sistema.

[2] Acordo de cooperação para a divulgação e uso do plataforma CulturaEduca: http://www4.fe.usp.br/wp-content/uploads/estagios/acordo-coop-lidas-feusp.pdf

[3] Mais sobre a parceria com do NAI-FEUSP com o Lidas: https://sites.usp.br/naifeusp/cultura-educa/

1.3 Estruturação da monografia

O capítulo 2 contém o referencial teórico, contendo conceitos fundamentais para o entendimento do desenvolvimento realizado no trabalho.

No capítulo 3 foi feito o relato do desenvolvimento realizado ao longo do trabalho. Pode ser considerado um estudo de caso referente à reestruturação dos ambientes de desenvolvimento e produção do portal CulturaEduca.

O capítulo 4 possui uma análise qualitativa para avaliação do impacto do trabalho feita a partir de retrospectivas com as partes interessadas no desenvolvimento do sistema.

Ao final, o capítulo 5 contém as considerações finais com as limitações do trabalho, conclusões a partir do que foi desenvolvido e próximos passos a serem tomados para manutenção do sistema.

Capítulo 2

Referencial teórico

A reestruturação dos ambientes de desenvolvimento e produção do portal CulturaEduca requer um entendimento de toda estrutura que compõe seu sistema. O trabalho foi realizado seguindo princípios de engenharia de *software*, *"a aplicação de uma abordagem empírica e científica para encontrar soluções eficientes e econômicas para problemas práticos em software"* - Farley, 2021, e de *DevOps*, *"um conjunto de práticas que tendem a reduzir o tempo entre aplicar uma mudança em um sistema e essa mudança ser colocada em produção, garantindo alta qualidade no processo"* - Bass *et al.*, 2015. Este capítulo contém explicações das principais tecnologias utilizadas ao longo do desenvolvimento.

2.1 Ambiente de desenvolvimento

Para realizar o desenvolvimento do sistema, um desenvolvedor precisa de uma estação de trabalho. Nos referimos a essa estação como a máquina local do desenvolvedor. Nessa máquina é preciso estarem configuradas todas as bibliotecas e programas necessários para executar o sistema localmente, suas dependências, assim como outros programas para dar apoio ao longo do desenvolvimento que não serão necessários na implantação do sistema (como editores de texto, depuradores de código, ferramentas de teste etc.). Chamamos esse ambiente de ambiente de desenvolvimento. Ao longo deste trabalho será dado foco nas dependências para execução do sistema quando nos referimos a esse ambiente.

2.2 Ambiente de produção

Para que os usuários do sistema consigam acessá-lo, precisamos hospedá-lo em uma máquina capaz de servir esse sistema. É preciso que essa máquina tenha então todas as dependências e configurações necessárias para a execução das aplicações que compõem o sistema, além de garantir a segurança do acesso às informações armazenadas. Chamamos esse ambiente de ambiente de produção. A implantação e atualização de funcionalidades nesse ambiente é uma das etapas mais sensíveis, pois envolve lidar com dados reais dos usuários e garantir que as mudanças não irão interromper seu funcionamento nem resultar na perda de dados.

2.3 *Python*

Python é uma linguagem de programação de propósito geral de alto nível. O código é compilado automaticamente para *bytecode* e executado, logo *Python* é adequado para uso como linguagem de *script*, linguagem de implementação de aplicativo da *web* etc. Como o *Python* pode ser estendido em C e C++, pode fornecer a velocidade necessária até mesmo para tarefas de computação intensiva. E por conta de suas fortes construções de estruturação (blocos de código aninhados, funções, classes, módulos e pacotes) e seu uso consistente de objetos e programação orientada a objetos, *Python* permite escrever aplicativos lógicos e claros para pequenas e grandes tarefas (Kuhlman, 2011).

2.4 Sistema gestor de pacotes

Aplicações *Python* com frequência utilizam pacotes que não fazem parte da biblioteca padrão da linguagem, tornando-se dependências do projeto. Para poder gerenciar essas dependências é aconselhado o uso de sistemas gestores de pacotes (*Tool recommendations - Python Packaging User Guide* 2022).

Um sistema gestor de pacotes é uma ferramenta que automatiza a instalação, atualização, configuração e remoção de distribuições de *software*. Os pacotes gerenciados possuem metadados contendo, entre outros, sua versão e suas dependências de outros pacotes necessárias para rodar corretamente. Desse modo o usuário não precisa gerenciar manualmente todos os pacotes que utiliza e suas respectivas dependências.

2.4.1 *Pip*

O *pip* é o gerenciador de pacotes oficial do *Python*. Por padrão se conecta ao repositório online de pacotes públicos *The Python Package Index (PyPI)* para buscar as dependências, mas pode ser configurado para utilizar outros repositórios (*Pip documentation* 2022).

2.5 Ambientes virtuais de *Python*

Se mais de um projeto da linguagem é desenvolvido em um sistema operacional, suas dependências podem ter conflitos de versão entre si. Além disso, para manter uma aplicação estável, é importante manter suas dependências fixadas, sem o perigo de uma atualização dos pacotes quebrar seu funcionamento.

Um ambiente virtual de *Python* é um diretório com binários e *scripts* de *shell* para simular uma instalação completa da linguagem e seus diferentes módulos sem interferir com a instalação existente no sistema operacional no qual a aplicação está rodando (*Python/Virtual environment* 2022).

Os binários incluem o interpretador da linguagem para execução de *scripts* e um gerenciador de pacotes como *pip* para instalação de módulos dentro do ambiente. Os *scripts* de *shell* servem para ativar ou desativar o ambiente. Quando ativado, o *shell* utiliza os caminhos dentro do diretório do ambiente para execução dos binários e instalação dos

pacotes, tornando as dependências da aplicação independentes da configuração do *Python* no sistema operacional e também de outros ambientes virtuais.

Criar um ambiente virtual para cada projeto permite evitar conflitos de dependências, além de facilitar sua configuração em diferentes máquinas.

2.5.1 *Virtualenv*

Virtualenv é uma ferramenta popular utilizada para criação de ambientes virtuais de *Python*. A partir do *Python 3.3*, um subconjunto de suas funcionalidades foi incorporada como uma biblioteca padrão da linguagem para criação de ambientes virtuais. Mas o pacote possui mais funcionalidades, como poder criar ambientes para versões arbitrárias de *Python*, ser extensível, ser atualizável via gerenciador de pacotes e possuir uma API programática para descrever ambientes (*Virtualenv documentation* 2022).

2.6 *Pipenv*

Pipenv é uma ferramenta que unifica a utilização do sistema gerenciador de pacotes e de ambientes virtuais, removendo a necessidade de executar os comandos de *pip* e *virtualenv* separadamente. A ferramenta também separa os arquivos que descrevem a lista de dependências do projeto entre os arquivos *Pipfile*, uma declaração abstrata de pacotes que compõem as dependências do projeto, e *Pipfile.lock*, uma especificação exata de versões instaladas incluindo pacotes não descritos que são dependências dos pacotes declarados. Carrega automaticamente também arquivos de variáveis de ambiente no ambiente virtual, provê melhor visualização do grafo de dependências, sugere últimas versões de pacotes para melhor segurança, entre outras funcionalidades (*Pipenv documentation* 2022).

2.7 *Container*

A virtualização fornece abstração sobre recursos reais de computação. O nível em que essa abstração é aplicada muda a aparência das diferentes técnicas existentes (Jain, 2020).

A virtualização em nível de sistema operacional é um paradigma no qual o *kernel* permite a existência de múltiplas instâncias isoladas de espaço de usuário. Um dos tipos dessas instâncias são os *containers*, conjuntos de processos que compartilham o espaço de *kernel* do sistema operacional da máquina hospedeira para gerenciamento de recursos, mas todos os caminhos do espaço de usuário estão isolados. É criado um limite lógico dentro do mesmo sistema operacional. Como exemplo, obtemos uma raiz diferente do sistema de arquivos, uma árvore de processos separada, um subsistema de rede separado e assim por diante (Jain, 2020).

Uma imagem é um arquivo estático que contém código executável que permite a criação de um *container* (*Container Images: Architecture and Best Practices* 2021). A imagem contém tudo que um *container* precisa para rodar: o sistema de tempo de execução, bibliotecas, programas utilitários, configurações e especificações da carga de trabalho desejada. Imagens

são compostas de camadas que representam o estado do sistema de arquivos após a execução de cada comando utilizado para sua criação. As camadas são adicionadas sequencialmente sobre uma imagem base e podem ser reutilizadas por diferentes imagens, economizando espaço de disco na máquina hospedeira.

O isolamento do espaço de usuário significa que para uma aplicação rodar dentro de um *container*, todas suas dependências precisam estar instaladas em conjunto, sem poder reutilizar bibliotecas e programas já instalados no sistema operacional da máquina hospedeira. Logo, são evitados conflitos de versões das dependências entre diferentes aplicações. E como a imagem contém o estado final do sistema de arquivos após execução de todos os comandos para sua criação, ela é imutável. Ou seja, todo *container* é gerado de forma consistente, independente do sistema hospedeiro. Isso torna *containers* úteis para padronizar a configuração do ambiente de desenvolvimento, teste e produção entre desenvolvedores de um projeto, enquanto facilitam também a distribuição e implantação de aplicações em diferentes máquinas.

A utilização de *containers* auxilia também em incorporar o princípio de *design* de separação de conceitos (do inglês "*separation of concerns*", termo cunhado por Dijkstra, 1982). Normalmente o princípio se refere à modularidade de seções de código de um programa, mas podemos utilizá-lo em uma escala maior como a implantação de um sistema composto de diversas aplicações em conjunto. Diferentes aplicações possuem dependências distintas, colocando cada uma em um *container* separado simplifica a definição de suas imagens e possibilita a reutilização em outros sistemas.

Para a execução de *containers* é necessária a utilização de um sistema de tempo de execução (*container runtime*). *Container runtimes* são programas responsáveis por carregar imagens de um repositório (local ou *online*), monitorar os recursos do sistema e gerenciar o ciclo de vida dos *containers* (*3 Types of Container Runtime and the Kubernetes Connection* 2021).

2.7.1 *Docker*

Docker é uma das plataformas mais populares atualmente para gerenciamento de *containers*. Seu sistema de tempo de execução, *Docker engine*, utiliza uma arquitetura de cliente-servidor para esse gerenciamento. O cliente *Docker* é uma interface que realiza comunicação com o *Docker daemon* via API REST, através de soquetes UNIX ou interfaces de rede. O *Docker daemon* é um programa que roda em segundo plano para construir, executar e distribuir os *containers* de *Docker*. O cliente e *daemon* não precisam estar necessariamente no mesmo sistema hospedeiro (*Docker overview* 2021).

As imagens utilizadas para criação de *containers* em *Docker* podem ser criadas pelo próprio usuário ou por terceiros e publicadas em um repositório. *Docker* permite a construção de uma imagem automaticamente a partir das instruções contidas em um *Dockerfile*, um documento de texto que contém todos os comandos que o usuário chamaria na interface de linha de comando do sistema. Normalmente imagens utilizam outras imagens como base, incrementando as bibliotecas, programas e configurações já contidos nelas para uma nova aplicação, o que incentiva o compartilhamento de imagens nos repositórios públicos, inclusive distribuições oficiais.

Cada *container* gerado a partir de uma imagem é uma instância separada e todos os arquivos criados dentro do *container* são armazenados em uma camada separada para escrita. Isso significa que enquanto as instâncias compartilham as camadas (exclusivas para leitura) de sua imagem, os novos dados gerados em cada *container* não são persistentes ou compartilhados por padrão. Caso exista a necessidade de persistência e compartilhamento de dados, é possível apontar diretórios dos *containers* para o sistema de arquivos da máquina hospedeira diretamente, *bind mounts*, ou através de *volumes*, gerenciados pelo *Docker* (*Manage data in Docker* 2021).

Para incorporar a separação de conceitos em um sistema isolando diferentes aplicações em *containers* separados é preciso fazer com que eles se comuniquem entre si e com o sistema operacional da máquina hospedeira. Para fazer isso *Docker* possui diversos *drivers* de rede para criar a interface de comunicação necessária. O *driver* padrão é o de ponte de rede (*network bridge*), que isola os *containers* em uma rede própria. Para a comunicação com com o hospedeiro, é preciso criar regras de *firewall* para mapear um porta do *container* para uma porta do *hospedeiro* (*Networking overview* 2021).

2.7.2 *Docker Compose*

Quando lidamos com múltiplos *containers*, pode se tornar complicado gerenciar seus ciclos de vida com todas as ligações e configurações necessárias. *Docker* oferece uma ferramenta chamada *Docker Compose* para mitigar esse problema. *Docker Compose* é outro cliente dentro da arquitetura do *Docker*. Ele utiliza um *compose file*, arquivo em formato YAML, para definir e configurar serviços. Cada serviço gera um ou mais *containers* a partir de uma imagem. E, a partir de um único comando, o *Docker Compose* faz todas as chamadas necessárias para comunicação com o *Docker daemon* para iniciar os *containers*, além das redes que os conectam e o volumes para persistência de dados (*Overview of Docker Compose* 2021).

A definição da imagem pode ser feita indicando um *Dockerfile* para sua construção ou o usuário pode referenciar uma imagem existente em seu repositório (caso não exista no repositório local, ela é buscada em um repositório online). Junto na definição ficam outros parâmetros, como política de reinicialização, portas que devem ser publicadas, variáveis de ambiente, dependências entre serviços, caminhos do sistema de arquivos para persistência de dados em *volumes* etc.

Essa orquestração de *containers* que o *Docker Compose* realiza é também uma maneira de documentar e centralizar a configuração de todas as aplicações de um projeto (bancos de dados, serviços de filas, *caches*, APIs *web* etc). É uma maneira conveniente de desenvolvedores entrarem para o desenvolvimento do projeto, diminuindo extensas documentações de configuração da máquina local a um arquivo e alguns comandos.

2.8 Sistema de Gerenciamento de Banco de Dados

Um Sistema de Gerenciamento de Banco de Dados (SGBD) é um sistema de software que permite aos usuários definir, criar, manter e controlar o acesso a um banco de dados (Connolly e Begg, 2014).

Para definição do banco de dados, normalmente é utilizada uma linguagem de definição dos dados (*Data Definition Language - DDL*), que permite a especificação de tipos e estruturas de dados e restrições sobre os dados armazenados. Para inserção, atualização, remoção e leitura de dados do banco, normalmente é utilizada uma linguagem de manipulação dos dados (*Data Manipulation Language - DML*). Ter um repositório central para todos os dados e suas definições permite à DML providenciar uma linguagem de consulta. A linguagem de consulta mais comum é o SQL (*Structured Query Language*), utilizada para bancos de dados relacionais.

Entre outras funcionalidades, o SGDB providencia também sistemas de: segurança, para prevenir acesso não autorizado; integridade, para manter a consistência dos dados; controle de concorrências, para permitir o acesso compartilhado; controle de recuperação, para restaurar o banco de dados para um estado consistente no caso de falhas (Connolly e Begg, 2014).

2.8.1 *PostgreSQL*

PostgreSQL é um SGBD relacional de código aberto que utiliza e estende a linguagem SQL. A origem do sistema ocorreu na Universidade da California, em Berkley, no ano de 1986. O projeto denominado POSTGRES, liderado pelo professor Michael Stonebraker e patrocinado pela "*Defense Advanced Research Projects Agency (DARPA)*, o *Army Research Office (ARO)*, a *National Science Foundation (NSF)*, e a *ESL Inc.* (*A Brief History of PostgreSQL* 2021).

Os conceitos iniciais do sistema e a definição inicial dos modelos de dados foram apresentados em conferências na década de 80 (M. Stonebraker e Rowe, 1986; Rowe e M. R. Stonebraker, 1987) e diversas versões do sistema foram feitas desde então.

Em 1993 o número de usuários do sistema se tornou grande demais e o projeto *POSTGRES* da Universidade da Califórnia terminou, com o sistema na versão 4.2. Em 1994 Andrew Yu e Jolly Chen incluíram um interpretador de SQL ao *POSTGRES* e, sob o novo nome de *Postgres95*, o sistema foi lançado na web como um projeto de código aberto, descendente do projeto *POSTGRES* original nascido em Berkley. Em 1996 o nome foi alterado novamente para *PostgreSQL* para evidenciar a relação das versões mais recentes do projeto com as funcionalidades em SQL.

Além de ser um software grátis e de código aberto, algumas outras funcionalidades fazem do *PostgreSQL* uma escolha interessante de SGBD: definição customizada de tipos de dados, criação de funções e adequação a quase todas as funcionalidades definidas na ISO/IEC 9075 "*Database Language SQL*"(*PostgreSQL: About* 2022).

2.9 Framework para aplicações web

Um *framework* de *software* é um conjunto de códigos-fonte ou bibliotecas que provêm uma gama de funcionalidades comuns a uma classe de aplicações. Ao invés de ter que reimplementar uma lógica de programação comumente usada, um programador pode aproveitar um *framework* que providencia a funcionalidade, focando melhor seu tempo nas regras de negócio de sua aplicação.

Um *framework* para aplicações *web* providencia um conjunto de funcionalidades comumente usadas nesse tipo de aplicação como: gerenciamento de autenticação e controle de acesso de usuários; persistência de dados; *cacheamento* de conteúdo; modelos de documento (*templates*) para separação entre lógica de regras de negócio e de visualização; interfaces para administração de conteúdo.

Um padrão de arquitetura de *software* bastante utilizado por *frameworks* de aplicação *web* é o MVC (*Model-View-Controller*). Esse padrão enfatiza a separação de conceitos entre as regras de negócio da aplicação e a exibição das informações. O *model* é responsável por gerenciar os dados e as regras de negócio. A *view* cuida do *layout* e exibição de conteúdo. O *controller* cuida das rotas (direcionamento de requisições e respostas) entre as partes do *model* e da *view* (Grove e Ozkan, 2011).

2.9.1 Django

Django é um *framework* para aplicações *web* gratuito e de código aberto escrito em *Python*. *Django* segue o padrão MVC e princípios de *design* como consistência, acoplamento fraco, menos código, desenvolvimento rápido, "não repita a si mesmo"(*Don't repeate yourself - DRY)* e "explícito é melhor que implícito"(*Django's design philosophies* 2022).

O desenvolvedor rapidamente consegue construir uma aplicação com mapeamento de objetos do banco de dados, gerenciamento de sessões de usuários, mapeamento de URLs com expressões regulares, sistema de *templates*, entre outras funcionalidades, com uma estrutura de projeto modular.

O *framework Django* foi iniciado em 2003, como um projeto realizado por Adrian Holovaty e Simon Willison do jornal *World* em Lawrence - Kansas, nos Estados Unidos. Em 2005, Holovaty e Willison lançaram a primeira versão pública do *framework*, nomeando-o em homenagem ao guitarrista Django Reinhardt. O *framework Django* opera sob a supervisão da *Django Software Foundation (DSF)*, possui mais de 1000 contribuidores, lançou mais de 15 versões, e possui mais de 3000 pacotes especificamente projetados para funcionar com o *framework* (Rubio, 2017).

2.10 Protocolo de transferência de hipertexto

O HTTP (*Hypertext Transfer Protocol*) é um protocolo de comunicação usado entre máquinas ligadas por uma rede, seja ela interna como LAN (*Local Area Network*) ou em uma mais abrangente como a *internet*. Assim como a maioria dos protocolos bem estabelecidos em computação, ela é regida por uma RFC, um documento com detalhes técnicos a serem seguidos caso queiramos construir um *software* em conformidade com o protocolo. Cada RFC pode sofrer uma alteração ou atualização caso a comunidade ache necessário, tornando assim a versão prévia obsoleta e o protocolo passa a ser regido pela nova versão da RFC.

A regra básica do HTTP é trabalhar no modelo requisição-resposta onde sempre o requerente é o cliente e nunca o servidor. O cliente faz uma requisição ao servidor e retorna uma resposta baseada em regras de negócio, segurança, disponibilidade, entre outros. Dizemos que o HTTP tem métodos, chamadas por vezes de verbos ou ações, cada

qual com seu escopo de atuação e regra de requisição. São eles *GET*, *HEAD*, *POST*, *PUT*, *DELETE*, *CONNECT*, *OPTIONS*, *TRACE* e *PATCH*.

O protocolo HTTP também prevê códigos de retorno para cada requisição solicitada ao servidor, sendo o código necessariamente um número de 3 dígitos, nunca há letras. Os códigos de retorno são divididos em classes. Todo código entre 100 e 199 (ditos 1xx) são respostas que indicam que a requisição foi recebida e entendida. Os códigos entre 200 e 299 (2xx) exprimem sucesso em todo processamento da requisição. Já os códigos entre 300 e 399 (3xx) instruem redirecionamento para o cliente/requerente da solicitação. Para erros temos 4xx e 5xx. O códigos 4xx são os erros originados do cliente, enquanto 5xx são erros originados pelo servidor.

Outra parte fundamental do HTTP é seu cabeçalho. Nele são encontrados: código de retorno, descritos acima; credenciais de acesso, caso a segurança dos recursos são dados por *tokens* de cabeçalhos; codificações aceitas pelo cliente, como compactação usando *gzip*; estruturas de resposta aceitas, como *JSON* ou *XML*; e assim por diante.

Para finalizar, temos o corpo de uma requisição ou resposta. Nele está o conteúdo principal de toda movimentação por esse protocolo. Se desejarmos buscar uma informação acerca de um recurso no servidor, essa informação virá no corpo da resposta; caso desejarmos criar uma entidade no servidor, as informações da entidade a ser criada será descrita no corpo.

2.10.1 Problemas de segurança do HTTP

Durante a comunicação entre cliente-servidor via HTTP, é possível o tráfego de dados sensíveis. Sempre que entramos em um site, o navegador faz um *GET* na página inicial do servidor e temos comumente um login onde devemos fornecer nossas credenciais. Esse formulário então é enviado ao servidor através de algum método HTTP pertinente que, por sua vez, confrontará as credenciais e, em caso de sucesso, devolverá um *cookie* de sessão válido para que possamos continuar efetuando nossas operações dentro do domínio.

A comunicação entre cliente e servidor, não é direta nas aplicações no mundo real. Seu computador está no mínimo a três dispositivos de distância do servidor. Ele está conectado a um roteador (primeiro dispositivo), que está conectado com o seu provedor de *internet* (segundo dispositivo), que está conectado ao servidor do site o qual está acessando (terceiro dispositivo). É possível "escutar"o tráfego de rede entre os computadores e ter acesso às credencias contidas no envio na comunicação com o site que está acessando.

Outra falha de segurança que também pode surgir por esse modelo dado pelo HTTP é a não autenticidade do servidor. Do modo como o HTTP foi escrito, qualquer dispositivo pode se passar pelo servidor. Se alguém de forma maliciosa tiver um roteador "provendo"*WI-FI* para as pessoas, quando alguém se conecta nesse roteador, ele diz que o servidor de DNS (serviço de tradução de nomes de domínio para endereços de IP) é o computador pessoal do atacante. Quando a vítima se conecta na rede *wi-fi* maliciosa e acessa um site, ela pode ser levada a uma página falsa que se parece com a original. A vítima então entra com suas credenciais e tem seus dados roubados pelo atacante. E o motivo do sucesso desse ataque é dado pelo fato de que, quando simplesmente usamos HTTP, não há como saber se o site que estamos acessando é o verdadeiro ou não.

2.11 Protocolo de transferência de hipertexto seguro

O HTTPS (HTTP Secure) nasceu basicamente a partir da necessidade de suprir duas grandes falhas de segurança do protocolo HTTP: a não autenticidade do servidor e a não criptografia entre as máquinas que se comunicam.

2.11.1 Criptografia simétrica e assimétrica

Os algoritmos de criptografia podem ser classificados em duas grandes categorias: de chave simétrica e de chave assimétrica (Agrawal e Mishra, 2012).

Na criptografia simétrica, usamos uma mesma senha para criptografar e descriptografar uma mensagem. Um exemplo de criptografia simétrica é a AES e um de seus escopos de atuação é em criptografia de unidades de armazenamentos como discos rígidos, *pendrives*, *SSDs* e afins. Outro cenário de atuação é na comunicação entre máquinas. Porém, para que ambas partes possam se comunicar de forma segura, elas já devem ter conhecimento prévio da chave em questão.

Na criptografia assimétrica, existem duas chaves: a pública e a privada. Nesse cenário, temos o detentor da privada e somente ele pode ter posse dessa chave; caso essa chave saia de seu domínio, toda a comunicação está comprometida. Essa entidade geralmente é um servidor de algum serviço. A outra entidade, geralmente o cliente que quer se comunicar com o servidor de forma segura, deve possuir a chave pública. A chave pública deve ser acessível a qualquer entidade pois ela não representa risco algum na segurança e é necessária para que a comunicação se dê de forma segura. Perceba que nesse contexto somente uma entidade pode enviar mensagem pois não temos dois pares de chaves e somente a chave privada consegue ler o texto original.

2.11.2 HTTPS em si

Para assegurar o sigilo de comunicação, inicia-se a comunicação usando um par de chave assimétrica e depois várias chaves simétricas são rotacionadas. Esse modo de criptografia é chamado de criptografia híbrida (*Hybrid encryption* 2022).

O início da comunicação usando o par de chaves da criptografia assimétrica se dá pelo cliente pedindo ao servidor a chave pública do servidor. O cliente então gera uma chave de criptografia simétrica e, usando a chave pública do servidor, criptografa a chave da criptografia simétrica e envia ao servidor. O servidor por sua vez descriptografa a mensagem e recebe e armazena a chave gerada da criptografia simétrica. A partir desse momento, todas as comunicações serão protegidas por uma criptografia simétrica. Dessa forma é resolvido o problema de sigilo entre as partes da comunicação.

Para solucionar o problema de autenticidade dos servidores, a pessoa responsável pelo servidor deve solicitar a uma Autoridade Certificadora (CA - *Certification Authority*) um certificado digital (par de chave pública e privada). Os navegadores atuais estão configurados para aceitar por padrão somente certificados assinados por CAs conhecidas. Caso o navegador se depare com um certificado não assinado por um CA que ele entende como

confiável ou que a data de validade tenha expirado, uma mensagem é exibida para o usuário do sistema para que ele decida se deseja correr o risco de talvez entrar em um site falso ou não.

Capítulo 3

Desenvolvimento

O desenvolvimento deste trabalho foi iniciado em abril de 2021, com entrada na equipe de desenvolvimento do projeto. A equipe foi composta de quatro estudantes de graduação que iniciaram o desenvolvimento em conjunto. Para organização de tarefas e acompanhamento de progresso, foram feitas reuniões semanais com os responsáveis do portal e professores supervisores. A equipe foi divida em duplas para realização de programação em pares no desenvolvimento das demandas do sistema.

No início foi feito estudo da documentação de *Django*, o *framework* de aplicações *web* utilizado pelo sistema do CulturaEduca, e leitura do código-fonte do projeto. O primeiro obstáculo encontrado foi a configuração do ambiente de desenvolvimento para executar a aplicação nas máquinas locais dos desenvolvedores.

3.1 Gerenciamento de pacotes

O sistema do CulturaEduca seguia a prática comum de gerenciar as dependências do projeto com o *pip* em um ambiente virtual configurado com *virtualenv*. Assim, um novo desenvolvedor deveria conseguir configurar o ambiente de desenvolvimento em sua máquina local criando um novo ambiente virtual e instalando os pacotes listados como dependências do projeto.

Os pacotes listados como dependências de um projeto podem ter sua versão indefinida, definida com uma regra ou fixada com valor específico. Além disso, cada pacote possui suas próprias dependências, com definições específicas de versões, criando um grafo de dependências. Se algum pacote desse grafo não tiver a versão fixada, a instalação não é determinística. Ou seja, se a instalação ocorrer em momentos distintos, as versões dos pacotes que compõem o grafo podem ser diferentes. E essa diferença de versões pode gerar inconsistência no funcionamento do sistema ou até mesmo quebrar o processo de instalação dos pacotes.

Para resolver este problema, vários gerenciadores de pacotes implementam o arquivo *lock*. Esse arquivo contém as versões exatas dos pacotes e suas dependências utilizadas no momento da instalação. Assim, inserindo o arquivo na ferramenta de controle de versões do projeto, a instalação se torna determinística para todos os desenvolvedores.

No início do desenvolvimento deste trabalho, verificou-se a existência de conflitos no momento da instalação das dependências, criadas pela falta de um arquivo *lock*. A solução encontrada foi utilizar a ferramenta *pipenv*, que provê a funcionalidade do arquivo *lock* e centraliza o uso do *pip* e do *virtualenv*, entre outras funcionalidades.

Para se aproximar ao máximo da então versão em produção, o servidor foi acessado e a lista das versões exatas dos pacotes instalados pelo *pip* foi gerada. A partir dessa lista, o ambiente de desenvolvimento foi configurado na máquina local dos desenvolvedores. Com o arquivo *lock*, estava garantido que todos tinham exatamente as mesmas versões das dependências.

3.2 Dependências do sistema operacional

Um dos problemas encontrados no início do desenvolvimento do trabalho foi a configuração do ambiente de desenvolvimento na máquina local dos desenvolvedores para conseguir executar o sistema. Além das dependências de pacotes *Python*, é necessária a instalação de diversos programas e bibliotecas no sistema operacional em versões específicas.

Caso o usuário já possua uma versão diferente do programa ou biblioteca em seu sistema operacional, a mudança de versão pode ser problemática. É preciso um trabalho extra para gerenciamento das diferentes versões para cada projeto ou pode ser que não seja possível ter a mesma dependência em versões diferentes.

No CulturaEduca, a versão do *PostgreSQL* utilizada é a 9. A título de exemplo, a versão estável atual é a 14. Portanto caso o desenvolvedor já possua uma versão diferente em sua máquina, será preciso gerenciar as diferentes versões. Outra dependência do sistema é um conjunto de bibliotecas de cálculos geoespaciais em versões antigas, por conta da compatibilidade com a versão do *Django* utilizada. Essas bibliotecas não permitem a gestão de múltiplas versões no sistema operacional.

Mais um desafio de configuração do ambiente de desenvolvimento para diferentes desenvolvedores é a possibilidade de que cada um utilize um sistema operacional diferente, que pode significar instruções diferentes para instalação das dependências. Isso dificulta a escrita de documentação do processo.

Outro desafio é a configuração do ambiente de produção, o servidor que vai hospedar a aplicação. Precisamos garantir que o que está sendo desenvolvido nas máquinas locais dos desenvolvedores irá funcionar quando a implantação for realizada.

Dadas as dificuldades apontadas, uma solução satisfatória seria um isolamento dessas dependências. Similar ao que os programas de ambientes virtuais como *virtualenv* providenciam para o isolamento de pacotes de uma linguagem, mas a nível de sistema operacional, para instalação de programas e bibliotecas.

3.3 Virtualização do sistema

A partir do momento que começamos hospedar múltiplas aplicações em um mesmo sistema operacional, como na figura 3.1, a complexidade da configuração necessária do ambiente e a chance de conflito de dependências crescem de tal forma que pode se tornar impraticável. O isolamento de cada aplicação em sua própria infraestrutura de *hardware* seria complicado demais e um desperdício de recursos. Logo, uma solução é isolar as aplicações a nível de *software* dentro de um mesmo *hardware*. Realizamos então a virtualização da infraestrutura, em que simulamos todo o ambiente que a aplicação requer para funcionar, isolando-a de outras aplicações em um mesmo sistema operacional hospedeiro.

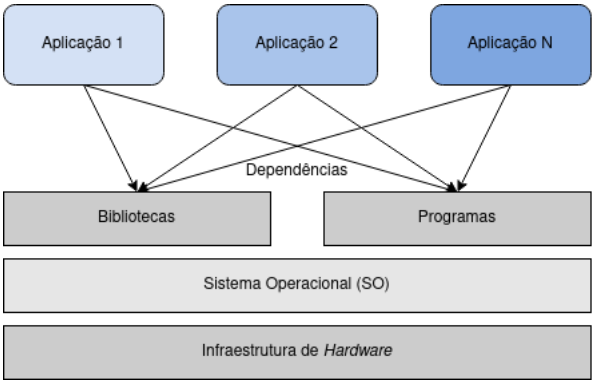

Figura 3.1: *Aplicações sem virtualização*

Uma solução é utilizar máquinas virtuais, através de uma tecnologia chamada *Hypervisor*. Ela permite subdividir uma máquina física em máquinas virtuais que rodam como sistemas operacionais convidados, como mostrado na figura 3.2. Como estamos simulando uma máquina completa para a aplicação, precisamos alocar processamento e memória dedicados a cada uma. Compartilhamento de recursos da infraestrutura de *hardware* se torna difícil de otimizar, pois tende-se a prover mais recursos do que o estritamente necessário para levar em conta picos de carga da aplicação (Agarwal, 2021).

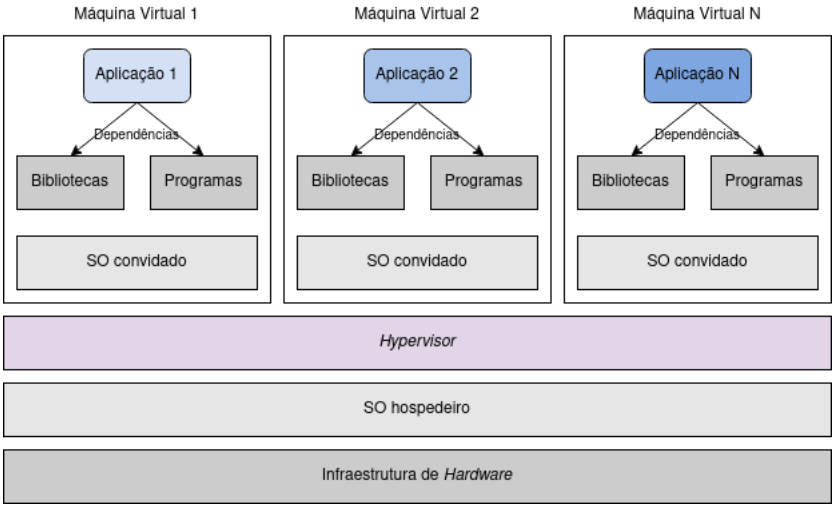

Figura 3.2: *Aplicações em máquinas virtuais*

Outra solução é a utilização de *containers*. *Containers* permitem o isolamento de aplicações para resolver o problema de conflitos de dependências. Como podemos ver na figura 3.3, compartilhamos o *kernel* do sistema operacional hospedeiro ao invés de precisar de um sistema operacional convidado e não é preciso alocar recursos de processamento e memória separadamente para cada *container*, diminuindo o desperdício de recursos computacionais.

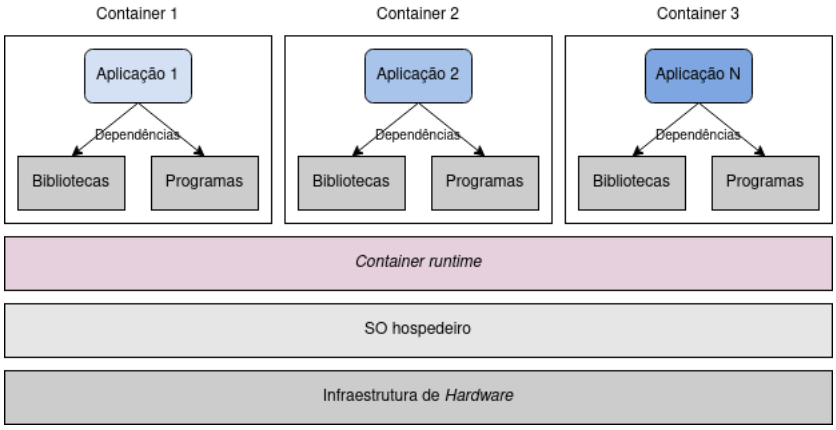

Figura 3.3: *Aplicações em containers*

Por conta da camada a mais de abstração ocupada pelo sistema operacional convidado em máquinas virtuais, *containers* se tornam uma opção mais leve de virtualização de aplicações. Nos quesitos de performance do processador, taxa de transferência de memória, performance de leitura/escrita de disco, teste de carga e velocidade de operações, soluções baseadas em *containers* de *Docker* demonstram melhor performance do que soluções em máquinas virtuais (Potdar *et al.*, 2020).

Máquinas virtuais continuam com um papel importante na indústria de tecnologia da informação, possibilitando oferecer serviços de diferentes sistemas operacionais e múltiplas máquinas para clientes em uma única infraestrutura de *hardware*. Enquanto *containers* oferecem uma solução para um problema crítico para desenvolvedores: executar *software* de maneira confiável, distribuída e com alta escalabilidade em qualquer ambiente computacional (Agarwal, 2021).

Assim foi escolhido realizar a *"containerização"* do sistema como forma de virtualização para tornar os ambientes de desenvolvimento e produção consistentes para o CulturaEduca. O *Docker* foi escolhido como plataforma de gerenciamento de *containers* por sua popularidade e pela alta gama de funcionalidades que centralizam todo o *pipeline* de distribuição de *software* (Kane e Matthias, 2018).

3.3.1 *Containerização* do SGBD

O SGBD do sistema, *PostgreSQL*, utiliza um protocolo de comunicação baseado em mensagens que é suportado via TCP/IP, seja em rede local ou via *internet*. Isso permite que o banco de dados não precise estar hospedado na mesma máquina que a aplicação. Mas, mesmo estando na mesma máquina, o banco de dados não precisa estar no mesmo espaço de usuário. Dessa maneira, é possível executar o SGBD em um *container* separado

da aplicação e a comunicação entre os dois é feita via TCP/IP através de uma porta exposta pelo *container*.

Assim, é possível utilizar uma imagem do SGBD, obtida no repositório oficial, com a versão específica utilizada pelo sistema em produção e garantir que todos os desenvolvedores irão utilizar a mesma versão. Outra vantagem trazida pelo uso de *containers* foi a facilidade de configuração do *PostGIS*, uma extensão externa de código aberto para armazenamento de dados geoespaciais e cálculos envolvendo esses dados. Os responsáveis pela extensão mantêm um repositório para distribuição de imagens de *PostgreSQL* com *PostGIS* já instalado.

A primeira etapa da *containerização* da aplicação foi a criação do *compose file* com a definição do serviço de banco de dados. Essa definição contém a configuração que é aplicada para cada *container* iniciado por esse serviço. A título de exemplo, as opções utilizadas foram:

- *image*: especifica a imagem que será utilizada para iniciar o *container*;
- *restart*: define a política de reinicialização do *container* (em casos de falhas ou mudança do estado da máquina hospedeira);
- *ports*: define quais portas do *container* serão expostas para o hospedeiro;
- *environment*: define variáveis de ambiente que estarão diponíveis dentro do *container* (no caso da imagem do *PostGIS*, essas variáveis são utilizadas para configuração inicial do banco de dados);
- *volumes*: definição de volumes, utilizados para persistência de dados gerados dentro do *container* (essencial para persistir os dados armazenados no banco de dados);

Inicialmente no desenvolvimento do trabalho, apenas o banco de dados do sistema estava dentro de um *container*. Os desenvolvedores haviam configurado o sistema operacional de suas máquinas locais e foram encontradas alternativas temporárias para resolução de erros provenientes das inconsistências entre versões das dependências. Assim, foi possível realizar o estudo do código, resolução de *bugs* e desenvolvimento de novas funcionalidades paralelamente à *containerização* da aplicação web.

3.3.2 *Containerização* do ambiente de produção

Uma demanda dos responsáveis pelo portal CulturaEdura se encaixou com o objetivo deste trabalho de *containerizar* a aplicação: migrar a versão em produção para ser hospedada em outro servidor. Uma nova máquina virtual foi disponibilizada no *interNuvem USP*, serviço de computação em nuvem da Universidade de São Paulo.

A tarefa então passou a ser criar um *Dockerfile*, um documento de texto que contém todos os comandos que um usuário pode chamar na linha de comando para montar uma imagem. A imagem contém a instalação de todas as dependências e configurações necessárias para execução da aplicação. Ao montar uma imagem, obtemos algumas vantagens sobre configurar manualmente a máquina do servidor:

- Durante a escrita do arquivo podemos verificar se o ambiente que estamos montando

está correto e testar a sequência de comandos sem necessidade de acesso ao servidor. O comportamento de uma imagem é o mesmo independente da máquina que vai rodar o *container* gerado a partir dela.

- Caso haja necessidade de uma nova migração de servidor no futuro, a configuração do ambiente da aplicação já está pronta para implantação em nova máquina.
- Com as dependências da aplicação isoladas em sua imagem, a máquina do servidor pode ser utilizada para hospedar outras aplicações (de preferência também *containerizadas*) que poderiam ter conflito de dependências.

O primeiro passo do *Dockerfile* é escolher o *template* (imagem base) utilizado para iniciar a configuração da imagem. Como o servidor que estava hospedando o sistema utilizava o *Ubuntu* (distribuição de *Linux* baseada no *Debian*) como sistema operacional, a imagem oficial do *Ubuntu* foi escolhida como *template*. Em termos práticos, é como se a configuração da imagem da aplicação iniciasse logo após a instalação do sistema operacional.

A aplicação utiliza um módulo do *framework* para auxiliar na manipulação de dados geoespaciais. Esse módulo utiliza as seguintes bibliotecas de código aberto:

- GEOS: realização de operações geométricas em *C++*;
- PROJ: conversão de dados geoespaciais em diferentes sistemas de coordenadas;
- GDAL: suporte para leitura de formatos de dados espaciais em vetor ou *raster*.

Mas, por conta da versão do *framework*, a versão atual dessas bibliotecas não possuem suporte pelo módulo. Portanto a instalação dessas bibliotecas precisam ser feitas a partir do código-fonte. Logo, as instruções seguintes em nosso *Dockerfile* se resumiram em baixar os códigos-fonte, execução de *scripts* de configuração, compilação do código e instalação para as três bibliotecas.

As imagens de *Docker* são criadas utilizando camadas. Uma camada é criada para cada comando descrito no *Dockerfile*. Ou seja, cada camada contém as mudanças no sistema de arquivos da imagem entre os estados antes e depois da execução do comando. Para acelerar o processo de construção das imagens, o *Docker* mantém um *cache* de cada camada. Se um comando não muda, ele não precisa ser executado novamente durante a construção de uma imagem, podemos utilizar o resultado de sua execução do *cache* de camadas. Quando um comando é alterado, o *cache* de sua camada e todas as camadas subsequentes são invalidados. Por esse motivo, a instalação das bibliotecas que requerem a compilação de seus códigos-fonte foi realizada no início, antes de qualquer outra dependência. Como é um processo demorado, o resultado da instalação fica no *cache* de camadas e não precisa ser realizado toda vez que alteramos algum comando seguinte. E também não temos comandos anteriores que invalidariam esse *cache*.

O passo seguinte no *Dockerfile* foi a instalação do resto das dependências do sistema operacional para execução da aplicação. Encontrar essas dependências envolveu investigar a máquina do servidor que estava hospedando o sistema no momento, por conta da falta de documentação.

A partir disso, a configuração do projeto foi iniciada com a criação e definição de

permissões das pastas para as quais o código-fonte seria copiado. Uma nova dependência que ainda não estava sendo utilizada e foi inclusa no passo anterior foi o *pipenv*. Copiando os arquivos *Pipfile* e *Pipfile.lock* do projeto para a imagem, podemos então instalar o *framework* e outras dependências de pacotes do *Python* com suas versões fixadas. Inicialmente foi feita a cópia apenas desses dois arquivos por conta do *cache* de camadas. Essas dependências de pacotes mudam pouco ao longo do desenvolvimento, diferente do resto dos arquivos do projeto. Portanto, se todo código-fonte fosse copiado nessa etapa, toda alteração do projeto iria alterar o *cache* da camada e a instalação das dependências precisaria ser feita novamente.

Por último foi feita a cópia de arquivos de configuração e do código-fonte do projeto para dentro da imagem, por ser a camada que mais muda. Ao final da imagem, normalmente é colocado também um comando que será executado no momento que um *container* for iniciado a partir da imagem. Como a aplicação é *web*, temos que iniciar um processo que mantenha ela rodando indefinidamente. No caso deste sistema, foi criado um *script* que inicia os programas necessários para executar a aplicação e mantê-la rodando em segundo plano.

Temos então uma imagem pronta que inicia nossa aplicação *web* no momento que o *container* for gerado a partir dela. Quaisquer configurações necessárias da aplicação que dependam de fatores externos à imagem são passadas para o *container* através de variáveis de ambiente no momento de sua execução.

Adicionamos então o serviço para gerar o *container* da aplicação em nosso *compose file*. Em conjunto com o serviço que gera o *container* do banco de dados, o sistema está pronto para ser hospedado em um servidor. No caso desse serviço utilizamos as seguintes opções:

- *build*: especifica que vamos construir a imagem a partir de um *Dockerfile* ao invés de utilizar uma imagem pronta de um repositório;
- *restart*: define a política de reinicialização do *container* (em casos de falhas ou mudança do estado da máquina hospedeira);
- *ports*: define quais portas do *container* serão expostas para o hospedeiro (e que serão utilizadas para direcionar o tráfego do sistema);
- *depends_on*: outros serviços dos quais este depende, que serão iniciados antes (iniciamos o serviço da aplicação após o do banco de dados para garantir que não haverá tentativa de conexão antes de ele estar disponível);
- *environment*: define variáveis de ambiente que estarão diponíveis dentro do *container* (colocamos aqui dados de acesso para envio de emails, conexão com banco de dados e outras configurações necessárias);
- *volumes*: definição de volumes, utilizados para persistência de dados gerados dentro do *container* (no caso desta aplicação utilizamos para persistir os arquivos enviados pelos usuários e o *cache* gerado por algumas funcionalidades);

Após a construção da imagem para reproduzir da forma mais fiel possível o ambiente de produção que estava sendo usado para o sistema, foi feita a implantação no novo

servidor. O processo se resumiu em: clonar o repositório do projeto, configurar o *compose file*, executar o *Docker Compose* para construir a imagem e iniciar os *containers* e configurar um servidor HTTP para apontar para a porta exposta pelo *container* da aplicação *web*. As únicas dependências da máquina do servidor passaram a ser o *Docker* e um servidor HTTP. Depois de validar com os responsáveis pelo projeto que o sistema estava funcionando de maneira correta no novo servidor, o sistema do servidor antigo foi parado e foi feito um *backup* do banco de dados. O *backup* foi passado para o novo servidor e usado para popular o banco de dados do novo servidor. Por último foi alterada a configuração de DNS do domínio do sistema para apontar para o IP do novo servidor e a migração foi concluída com sucesso.

3.3.3 *Containerização* do ambiente de desenvolvimento

A principal motivação de colocar o ambiente de desenvolvimento em *containers* foi a dificuldade inicial encontrada pela nova equipe de desenvolvimento do sistema de configurar as respectivas máquinas locais. E como há previsão de uma rotatividade alta de desenvolvedores, facilitar o processo de configuração do ambiente de desenvolvimento traz valor ao projeto.

O *compose file* e *Dockerfile* do ambiente de desenvolvimento é muito similar aos do ambiente de produção. A instalação de dependências tanto do sistema operacional e de pacotes do *Python* é a mesma, a principal diferença é a configuração necessária para executar a aplicação.

O *Django*, *framework* utilizado pelo projeto, possui recomendações diferentes para executar a aplicação em modo de desenvolvimento ou produção. Em modo de desenvolvimento o próprio *framework* serve a aplicação sem a necessidade de um servidor HTTP dedicado. Porém é recomendado apenas para o modo de desenvolvimento, pois não possui a robustez e funcionalidades requeridas para servir a aplicação na internet. Além disso, nesse modo, a aplicação verifica mudanças nos arquivos do código-fonte para reiniciar automaticamente e mostrar o resultado para o desenvolvedor, o que é desnecessário em um ambiente de produção.

Com essa funcionalidade de verificação de mudanças dos arquivos temos uma diferença no final do *Dockerfile* de desenvolvimento em comparação com o de produção. A cópia do código-fonte do projeto para dentro da imagem. Se fosse mantida a instrução de cópia dos arquivos, toda vez que fosse feita uma alteração do código do projeto, a imagem da aplicação precisaria ser construída novamente para iniciar o *container*. Por mais que o *cache* das camadas anteriores se mantenha, ter que reconstruir a imagem e reiniciar o *container* toda vez que o código é modificado torna-se um fardo no processo de desenvolvimento. Além disso perdemos a funcionalidade do *framework* de reiniciar a aplicação automaticamente a partir das mudanças do código.

Para resolver este problema, configuramos um novo volume para o serviço de nossa aplicação no *compose file*. Vimos que volumes são usados para persistência de dados e que isso é feito fazendo com que o *container* aponte um caminho do seu sistema de arquivos para o sistema de arquivos da máquina hospedeira. Assim, se configurarmos um volume que aponte o caminho do código-fonte da aplicação dentro dele para o caminho do código-

fonte em nossa máquina local, não precisamos mais copiar todos os arquivos para dentro da imagem quando formos executar a aplicação. Assim, a imagem não precisa ser reconstruída toda vez que alteramos o código e o *framework* consegue continuar reiniciando a aplicação automaticamente toda vez que verificar uma mudança.

Assim, a única dependência de sistema operacional da máquina local do desenvolvedor é o *Docker*. A configuração do ambiente de desenvolvimento se resume a clonar o repositório, configurar o *compose file* e iniciar os *containers*, o sistema estará acessível na porta exposta pelo *container* da aplicação.

3.4 Certificado SSL

No início do desenvolvimento, um problema verificado com o sistema do CulturaEduca foi a falta de HTTPS na versão em produção. Páginas *web* servidas via HTTP são vulneráveis a interceptação, injeção de conteúdo e roubo de *cookies*, que podem ser usados para sequestrar contas de usuários. A utilização de HTTPS resolve a maioria desses problemas e já representa 80% (*Percentage of Web Pages Loaded by Firefox Using HTTPS* 2022) de todos os carregamentos de páginas da internet.

Com a migração do sistema para um novo servidor, aproveitou-se para garantir que as páginas seriam servidas via HTTPS. Para habilitar o protocolo é necessário obter um certificado SSL de uma Autoridade Certificadora (AC) e configurá-lo no servidor *web*. Foi escolhido obter o certificado da *Let's Encrypt*, uma AC gratuita, automatizada e aberta que se tornou possível graças à organização sem fins lucrativos *Internet Security Research Group (ISRG)*.

No caso em que se tem acesso via linha de comando ao servidor, é recomendado utilizar o *Certbot*, uma ferramenta de *software* gratuita e aberta que utiliza o protocolo ACME (*Automatic Certificate Management Environment*) para obter e configurar certificados da *Let's Encrypt* no servidor de maneira automatizada. A ferramenta foi instalada no servidor e os passos descritos na documentação foram seguidos para gerar o certificado SSL e habilitar HTTPS no site do CulturaEduca (https://culturaeduca.cc).

3.5 Ambiente de homologação

O ambiente de homologação (ou pré-produção) é um ambiente idêntico ao ambiente de produção para realização de testes. Para aumentar a confiança no lançamento de uma nova versão do sistema, todas as configurações e procedimentos provenientes da atualização são executados em um ambiente que simula o que é utilizado na versão que está no ar. Assim é possível verificar através de testes manuais ou automatizados se o sistema continua estável. Esse ambiente pode ser utilizado também para realização de testes de performance, que normalmente são sensíveis à infraestrutura na qual o sistema está implantado.

Além dessas vantagens listadas tornarem a existência de um ambiente de homologação uma prática comum na indústria, a demanda por esse ambiente foi reportada pelos responsáveis do projeto para a visualização e validação de novas funcionalidades desenvolvidas.

A implantação do ambiente de homologação no servidor foi facilitada pela *containerização* do ambiente de produção. Para simular um ambiente idêntico, é utilizado um *compose file* similar que utiliza o mesmo *Dockerfile* para a montagem da imagem da aplicação. São modificadas apenas as variáveis de ambiente referentes a credenciais, a definição de volumes onde serão persistidos os dados e a porta do *container* que será exposta ao hospedeiro. A partir disso, foi configurado um novo subdomínio que redireciona o tráfego para o *container* da aplicação no ambiente de homologação, enquanto o tráfego do domínio principal continua sendo direcionado para o *container* da aplicação em produção.

3.6 Atualização de versões

O uso de pacotes e bibliotecas de terceiros traz vários benefícios para o desenvolvimento de *software*, principalmente a economia de tempo. Os desenvolvedores podem focar nas regras de negócio da aplicação em vez de investir o tempo necessário para desenvolver e testar funcionalidades comuns. Essas dependências de terceiros permitem utilizar um código que já foi testado em diversos casos diferentes e receberam *feedback* de outros desenvolvedores que provavelmente resultaram em consertos e melhorias de seu funcionamento. Seu uso pode melhorar a qualidade e estabilidade da aplicação e tornar o código mais modular.

Porém a dependência de pacotes e bibliotecas de terceiros traz também desvantagens para o projeto. A aplicação pode ficar refém de vulnerabilidades e *bugs* encontrados nessas dependências, tendo que contar com o suporte contínuo delas e esperar por atualizações. E dependendo de quão acoplada é a dependência ao funcionamento do sistema, a necessidade de atualização ou troca dela pode requerer mudanças significativas no código. O uso excessivo de dependências de terceiros pode também ser detrimentosa para a performance da aplicação.

Dado que os pacotes e bibliotecas de terceiros estão constantemente evoluindo, com novas funcionalidades e consertos de *bugs* e vulnerabilidades, é importante então que os desenvolvedores mantenham atualizadas as dependências de um projeto. Porém é uma tarefa difícil de ser priorizada, principalmente se o projeto não possuir testes automatizados que verifiquem se a aplicação continua funcionando corretamente após as atualizações.

Em vista que as dependências do CulturaEduca não recebiam atualização desde abril de 2015, foi listada como demanda do projeto a atualização de suas versões. O foco da atualização se deu na versão do *framework* de aplicação *web* utilizado, *Django*, e na versão de sua linguagem de programação, *Python*. O processos de atualização destes dois pontos envolve um processo longo de verificação da compatibilidade do resto das dependências do projeto a cada passo dado.

Para realizar o processo de atualização de dependências em paralelo com o desenvolvimento do projeto, foi criada uma nova ramificação (*branch*) no repositório *git* do código-fonte. Conforme novas funcionalidades ou consertos de *bugs* eram realizados, o código era mesclado à ramificação de atualizações, para verificar a necessidade de modificações por conta das atualizações.

A *containerização* dos ambientes de desenvolvimento e produção do sistema se mostrou

útil novamente durante este processo. As atualizações de versões podem incluir alteração das dependências do sistema operacional e, consequentemente, da configuração desses ambientes. Gerenciar diferentes configurações do sistema operacional em uma mesma máquina seria trabalhoso demais, principalmente se o desenvolvedor precisar modificar o código na ramificação estável do projeto onde novos desenvolvimentos estavam sendo feitos. Cada ramificação possui seu *Dockerfile* e *compose file* versionados, permitindo a montagem de imagens diferentes. O desenvolvedor pode então alternar livremente entre cada ramificação para realizar o desenvolvimento necessário em cada uma.

3.6.1 Atualização do *framework* de aplicação *web*

Como o desenvolvimento do sistema é centrado na utilização do *Django*, o processo de atualização de dependências do projeto se resumiu a atualizar sua versão gradualmente, verificando quais dependências passavam a ser incompatíveis a cada passo e atualizá-las também, modificando o código do projeto conforme necessário. Caso alguma dependência não fosse mais compatível, alguma alternativa era buscada.

Segundo a documentação do *Django*, suas versões são numeradas na forma **A.B.C**. Sendo **A.B** um *feature release*, contendo novas funcionalidades e melhorias. Enquanto **C** é um número incremental de *patch release* e contém apenas consertos de *bugs* e vulnerabilidades, compatível com todas as versões do mesmo *feature release*. Na *feature release*, **A** é a versão principal e **B** a secundária. Além disso, algumas *feature releases* são designadas como *long-term support (LTS) releases*. São versões que garantem suporte por um determinado período de tempo (tipicamente 3 anos) para correções de segurança e *bugs* que resultam em perda de dados (*Django's release process* 2022).

É recomendado que a atualização seja feita aumentando gradualmente a versão secundária. O *framework* possui uma política de depreciação em que avisa sobre a remoção de funcionalidades em versões futuras. Assim, os desenvolvedores recebem avisos sobre o que será removido e podem alterar o código conforme necessário. Porém não existe o mesmo tipo de aviso para as dependências de terceiros. E sem a existência de testes automatizados, foi preciso testar manualmente o sistema a cada modificação.

A versão do *Django* no projeto no início do trabalho era a *1.8* (*LTS* com fim de suporte em 2018 - *Download Django* 2022). Logo, a primeira atualização realizada foi para a versão *1.9*. Após essa atualização, o sistema gerenciador de pacotes precisa reinstalar todas as dependências, pois precisa realizar o cálculo do novo grafo de dependências. Se alguma versão de algum pacote não for compatível com a atualização, a instalação não é concluída. A partir da informação dos pacotes incompatíveis, foram buscadas suas documentações no repositório público e atualizadas suas versões ou buscadas soluções alternativas.

Após sucesso na instalação das dependências do sistema gerenciador de pacotes, o sistema foi testado para verificar erros de execução. Conforme os erros eram detectados, a mensagem de erro referente à pilha de execução apontava para o ponto de código que precisava ser modificado para se adequar às novas versões das dependências. O processo foi repetido para as atualizações para as versões *1.10* e *1.11* (*LTS* com fim de suporte em 2020).

A atualização para a versão *1.11* foi mesclada na ramificação do repositório que estava

sendo utilizada para desenvolvimento e eventualmente incorporada à ramificação principal. Atualmente a versão em produção está nessa versão. A versão *1.11* é a última versão do *Django* com suporte para *Python 2*. Logo, o foco passou a ser a atualização do projeto para utilização do *Python 3*.

3.6.2 Atualização da linguagem de programação

Python 2 foi lançado em 2000, mas após alguns anos foi verificado a necessidade de realizar grandes mudanças para melhorar a linguagem. Em 2006 foi lançado o *Python 3*. Em 2008, foi anunciado o fim do suporte da versão *2* em 2015. Porém, em 2014, o suporte foi estendido até 2020 por conta da alta quantidade de desenvolvedores ainda utilizando a versão. A partir de 2020 *Python 2* deixou então de ter suporte dos responsáveis por manter a linguagem. Logo, é altamente recomendável a atualização para a nova versão, já que esta não terá mais reporte de *bugs*, consertos ou mudanças.

Com a mudança foi então necessário atualizar as dependências do sistema para versões que tenham compatibilidade com *Python 3*. Verificou-se que algumas dependências do projeto não possuiam suporte para a nova versão. Então uma quantidade considerável de tempo foi dedicada para encontrar soluções alternativas, seja em outros pacotes ou implementando a funcionalidade diretamente. Além disso, o código foi revisado para aplicar as mudanças de sintaxe necessárias para migração para *Python 3*.

Durante a migração do código para *Python 3* foram encontrados problemas em dependências que seriam resolvidos em versões subsequentes mas que dependiam de uma versão mais recente do *framework*. Portanto, junto dessa atualização foi iniciada a atualização para a versão 2.0 do *Django*. Neste processo foi identificada uma dependência altamente acoplada ao código do projeto sem suporte para a nova versão do *framework*, o que evidencia um dos problemas de dependência de pacotes de terceiros. A solução decidida foi incorporar o pacote no código do projeto e desenvolver a migração para a nova versão.

O desenvolvimento deste trabalho chegou até esta etapa e não houve tempo suficiente para finalizar essa migração.

Capítulo 4

Análise qualitativa

4.1 Desenvolvedores

No final do desenvolvimento deste trabalho, foi feita uma retrospectiva com os desenvolvedores sobre a experiência no projeto, com perguntas qualitativas focadas na configuração do ambiente de desenvolvimento.

4.1.1 Experiência no início do desenvolvimento

No início do trabalho o código-fonte do projeto não possuía documentação referente à configuração do ambiente de desenvolvimento do projeto. E devido a conflitos de dependências, a instalação de pacotes do *Python* listadas como requisitos do projeto não era concluída e ninguém conseguia executar a aplicação. Estava disponível uma gravação de uma reunião com um desenvolvedor anterior auxiliando uma das pessoas responsáveis pelo portal a encontrar alternativas para configurar o ambiente de desenvolvimento em sua máquina local. *"O vídeo era muito longo e difícil de achar as instruções que precisávamos. Além disso algumas soluções eram específicas do sistema operacional daquela pessoa, referentes a problemas diferentes dos que estávamos tendo."* - membro da equipe.

Depois de encontrar soluções alternativas para os problemas de dependências dos pacotes de *Python* e das dependências do sistema operacional, todos conseguiram executar o sistema em sua máquina local. Mas foi consenso que o tempo e trabalho gasto para configurar o ambiente de desenvolvimento foi longo demais.

4.1.2 Experiência com o ambiente *containerizado*

Depois de configurar o ambiente de desenvolvimento com *Docker*, os desenvolvedores precisaram reconfigurar o ambiente de desenvolvimento para utilizar o ambiente *containerizado*. A única dependência do sistema operacional passou a ser o *Docker*. Após configurar o serviço da aplicação no *compose file*, todos conseguiram construir a imagem e executar os *containers* em pouco tempo. *"Uma vez precisei formatar o computador e reinstalar o projeto na minha máquina e foi muito rápido."* - membro da equipe.

Dois dos desenvolvedores que utilizam *Windows* estavam usando o *WSL* ("Subsistema do Windows para Linux", ferramenta que permite executar um ambiente GNU/Linux diretamente no *Windows*, sem modificações e sem a sobrecarga de uma máquina virtual tradicional ou instalação *dualboot*) para os *containers* terem acesso ao *kernel* do Linux. Ambos tiveram alguns problemas referentes à utilização do *Docker*: um referente à publicação das portas dos *containers* na máquina local; e outro refente ao cache de camadas de imagens, que de vez em quando era perdido e a construção da imagem precisava ser refeita do início.

Aparentemente o consenso foi que a dificuldade inicial é entender como os *containers* funcionam mas que facilitam bem o processo de configurar o projeto. *"Meu único problema no início foi não conhecer a tecnologia do Docker, mas com o tempo e com a prática se tornou fácil iniciar o ambiente e ver a facilidade que ele traz para realizar mudanças nesse ambiente e replicar isso para todos os desenvolvedores."* - membro da equipe.

Além disso foi apontado o fato que o *Dockerfile* e *compose file* servem também como documentação da configuração dos ambientes de desenvolvimento e produção. Isso ajuda a entender quais são as dependências do sistema e quais são os programas que utiliza em sua inicialização.

4.1.3 Dificuldades enfrentadas ao longo do desenvolvimento

Um das dificuldades enfrentadas durante o processo de desenvolvimento foi a falta de conhecimento do *Django*. O que no início parecia falta de documentação e comentários no código eram apenas o funcionamento normal do *framework*. *"Django tem muita coisa implícita, que para quem está começando atrapalha muito."* - membro da equipe.

Para os desenvolvedores que trabalharam no processo de atualização de versões, o processo foi mais demorado que o esperado e acabou não indo tão longe. *"O processo de atualização das dependências foi muito trabalhoso por conta do desenvolvimento do projeto estar parado há muito tempo, era preciso ficar procurando documentações antigas para entender os problemas e tentar achar soluções."* - membro da equipe.

4.1.4 Entrada de novos desenvolvedores

Ao final foi perguntado sobre como viam a experiência de um novo desenvolvedor na equipe e quais seriam os conhecimentos necessários para se ambientar no projeto.

"Vai ser mais fácil começar porque vai conseguir fazer o projeto rodar logo no começo, ao invés de gastar tempo pra aprender como configurar o ambiente. Se não tiver familiaridade com Docker pode ser uma dificuldade." - membro da equipe.

"Mesmo sem conhecimento, com a documentação feita para instalar e executar o Docker, é tranquilo instalar o ambiente." - membro da equipe.

"O setup está mais prático, não precisa mais gastar semanas nisso. Mas é preciso entender melhor o padrão do framework para pode desenvolver." - membro da equipe.

4.2 Responsáveis

Outra retrospectiva com perguntas qualitativas foi feita com uma das pessoas responsáveis pelo CulturaEduca para identificar o impacto do trabalho na visão dos clientes do projeto.

4.2.1 Estado do projeto antes da parceria

O CulturaEduca estava sem equipe de desenvolvimento. O projeto ficou quatro anos parado. Sabia-se que a atualização de versões seria um problema e várias demandas de novas funcionalidades e consertos de *bugs* estavam se acumulando. *"Eu tinha uma instalação local com script para deploy e acesso ao servidor. Eu entrava só para fazer uma alteração pontual (mexer em algum texto aqui ou ali), mas tinha medo de quebrar alguma coisa."* - responsável do projeto.

Não existia documentação da configuração do ambiente de desenvolvimento. Quando necessário, foi gravada uma reunião na qual foi feito um acompanhamento do processo por um desenvolvedor anterior. Já foi possível identificar no processo que eram necessários vários ajustes por conta de conflitos das dependências e as alternativas eram específicas à máquina da pessoa.

4.2.2 Experiência com o ambiente *containerizado*

Ainda que não tenha função de desenvolvedora de *software*, esta pessoa é responsável pelas bases de dados que compõem o CulturaEduca. Portanto é importante o ambiente de desenvolvimento do projeto configurado para realizar as alterações necessárias referente à atualização das bases e ficar bem informada sobre o desenvolvimento do projeto.

Assim como os desenvolvedores, a dificuldade inicial foi com os conceitos de *containers* e uso do *Docker*. Depois da explicação que recebeu, ficou fácil de entender e seguir a documentação que foi criada para configuração do ambiente. *"O processo foi muito mais rápido, deixou de ter erros de conflitos que precisavam de alguma gambiarra para continuar a configuração do meu computador. Simplificou o manuseio também, depois que está tudo montado. Com apenas um comando o ambiente está pronto pra rodar."* - responsável do projeto.

4.2.3 Visão geral

Outras mudanças foram facilitadas com o desenvolvimento do trabalho, como a criação do ambiente de homologação (toda alteração estava sendo aplicada diretamente na produção), inclusão de uma ferramenta de *analytics* (como foi *containerizada* não teve nenhum conflito com os sistemas no servidor) e organização do projeto em um repositório público de uma ferramenta de controle de versão aberta. *"Eu entendo que o maior ganho que tivemos foi "desengessar"o projeto e poder voltar a fazer novos desenvolvimentos"* - responsável do projeto.

Foi apontado também que as dificuldades de desenvolvimento do CulturaEduca são comuns em projetos de entidades de caráter social, que nem sempre possuem recursos para

manutenção de uma equipe de desenvolvimento com expertise para resolver problemas com soluções atuais e que estão sendo usadas pelo mercado. Nesse sentido a parceria com a Universidade e criação de um ambiente de pesquisa sobre o portal foi essencial.

Capítulo 5

Considerações finais

5.1 Conclusões

Uma das principais limitações encontradas foi a falta de tempo disponível para o desenvolvimento das demandas do projeto. Ainda que parceria com a universidade tenha permitido aproximar estudantes de computação para o projeto, um sistema do porte do CulturaEduca requer uma equipe de desenvolvedores experientes trabalhando em tempo integral, mas seu cenário atual não é esse. A falta de documentação do projeto, unido à falta de experiência nas tecnologias utilizadas no sistema, tornou o desenvolvimento lento. E não foi possível alocar todo o tempo disponível nos objetivos do trabalho, precisando-se também utilizar tempo na organização do projeto, levantamento de requisitos, distribuição de tarefas, consertos de *bugs*, suporte para outros desenvolvedores etc.

Devido à limitação de tempo também, não foi possível realizar a análise qualitativa com uma metologia formal como a teoria fundamentada em dados. Foi possível apenas realizar uma análise subjetiva das respostas recebidas nas retrospectivas com as partes interessadas no desenvolvimento do sistema.

Em comparação com as experiências iniciais, a reestruturação dos ambientes de desenvolvimento e produção do CulturaEduca com a aplicação de uma arquitetura *containerizada* trouxe valor ao projeto. Facilitou o processo de desenvolvimento de novas funcionalidades e correções de *bugs*, aumentando a confiança de que as alterações funcionarão como esperado na versão em produção. Assim como facilitou o processo de atualização das versões, possibilitando alternar entre diferentes configurações do ambiente sem ter que reconfigurar todas as dependências na máquina local. E novas alterações do ambiente proveniente dessas atualizações facilmente são distribuídas para os outros desenvolvedores.

Principalmente em um projeto social com incerteza sobre a equipe desenvolvedores como o CulturaEduca, facilitar o processo de entrada de um novo desenvolvedor diminuindo o tempo necessário para configuração do ambiente faz com que seja possível trazer valor ao projeto mais rápido. E simplificar o processo de implantação do sistema, encapsulando a configuração dos ambientes de produção e homologação torna o ciclo de vida de desenvolvimento do sistema mais confiável.

A atualização das dependências de terceiros em projetos de *software* não são praticadas com frequência por desenvolvedores (Kula *et al.*, 2018). Além de novas funcionalidades e melhorias, consertos de *bugs* e atualizações de segurança, uma das recomendações da documentação do *framework* utilizado apontam que atualizar sua versão com frequência faz com que atualizações futuras sejam menos dolorosas. Isso ficou evidenciado durante a realização deste trabalho.

5.2 Próximos passos

Por conta das limitações do trabalho, não foi possível avançar o quanto se esperava na atualização de versões das dependências do CulturaEduca. O próximo passo é continuar a atualização dessas dependências o máximo possível, reestruturando o código do projeto conforme necessário.

Uma tarefa que se julga bem importante também é a implementação de testes automatizados. É possível entender que não foram implementados no início do projeto para economizar tempo de desenvolvimento. Mas a falta deles é uma das causas da demora no processo de atualização atualmente. *"Com o planejamento adequado, uma ferramenta de teste apropriada e um processo definido para introduzir testes automatizados, o esforço total de teste exigido com testes automatizados representa apenas uma fração do esforço de teste exigido com métodos manuais"* - Dustin, 1999.

Referências

[*3 Types of Container Runtime and the Kubernetes Connection* 2021] *3 Types of Container Runtime and the Kubernetes Connection*. Set. de 2021. URL: https://www.aquasec.com/cloud-native-academy/container-security/container-runtime/ (acesso em 20/12/2021) (citado na pg. 8).

[*A Brief History of PostgreSQL* 2021] *A Brief History of PostgreSQL*. Nov. de 2021. URL: https://www.postgresql.org/docs/current/history.html (acesso em 05/01/2022) (citado na pg. 10).

[Agarwal 2021] Gaurav Agarwal. *Modern DevOps Practices: Implement and secure DevOps in the public cloud with cutting-edge tools, tips, tricks, and techniques*. Packt Publishing, 2021. ISBN: 9781800562387 (citado nas pgs. 17, 18).

[Agrawal e Mishra 2012] Monika Agrawal e Pradeep Mishra. "A comparative survey on symmetric key encryption techniques". Em: *International Journal on Computer Science and Engineering* 4.5 (2012), pg. 877 (citado na pg. 13).

[Bass *et al.* 2015] Len Bass, Ingo Weber e Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Educational, 2015. ISBN: 9780134049847 (citado na pg. 5).

[Connolly e Begg 2014] Thomas Connolly e Carolyn Begg. *Database systems: A practical approach to design, implementation, and management: Global edition*. 6ª ed. Philadelphia, PA: Pearson Education, 2014 (citado nas pgs. 9, 10).

[*Container Images: Architecture and Best Practices* 2021] *Container Images: Architecture and Best Practices*. Abr. de 2021. URL: https://www.aquasec.com/cloud-native-academy/container-security/container-images/ (acesso em 20/12/2021) (citado na pg. 7).

[Dijkstra 1982] Edsger W Dijkstra. "On the role of scientific thought". Em: *Selected writings on computing: a personal perspective*. Springer, 1982, pgs. 60–66 (citado na pg. 8).

[*Django's design philosophies* 2022] *Django's design philosophies*. Jan. de 2022. URL: https://docs.djangoproject.com/en/4.0/misc/design-philosophies/ (acesso em 15/01/2022) (citado na pg. 11).

[*Django's release process* 2022] *Django's release process*. Jan. de 2022. URL: https://docs.djangoproject.com/en/4.0/internals/release-process (acesso em 15/01/2022) (citado na pg. 25).

[*Docker overview* 2021] *Docker overview*. Dez. de 2021. URL: https://docs.docker.com/get-started/overview/#docker-architecture (acesso em 14/12/2021) (citado na pg. 8).

[*Download Django* 2022] *Download Django*. Jan. de 2022. URL: https://www.djangoproject.com/download/ (acesso em 15/01/2022) (citado na pg. 25).

[Dustin 1999] Elfriede Dustin. *Automated Software Testing: Introduction, Management, and Performance: Introduction, Management, and Performance*. Addison-Wesley Professional, 1999. ISBN: 9780672333842 (citado na pg. 32).

[Farley 2021] David Farley. *Modern software engineering: Doing what works to build better software faster*. Addison-Wesley Professional, dez. de 2021. ISBN: 9780137314911 (citado na pg. 5).

[Grove e Ozkan 2011] Ralph F Grove e Eray Ozkan. "The mvc-web design pattern". Em: *International Conference on Web Information Systems and Technologies*. Vol. 2. SCITEPRESS. 2011, pgs. 127–130 (citado na pg. 11).

[*Hybrid encryption* 2022] *Hybrid encryption*. Fev de 2022. URL: https://developers.google.com/tink/hybrid (acesso em 10/02/2022) (citado na pg. 13).

[Jain 2020] Shashank Mohan Jain. *Linux Containers and Virtualization: A Kernel Perspective*. 1ª ed. APress, 2020. ISBN: 9781484262825 (citado na pg. 7).

[Kane e Matthias 2018] Sean P. Kane e Karl Matthias. *Docker: Up & running: Shipping reliable containers in production*. 2ª ed. O'Reilly Media, 2018. ISBN: 9781492036739 (citado na pg. 18).

[Kuhlman 2011] Dave Kuhlman. *A python book: Beginning python, advanced python, and python exercises*. Platypus Global Media, 2011. ISBN: 9780984221233 (citado na pg. 6).

[Kula *et al.* 2018] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio e Katsuro Inoue. "Do developers update their library dependencies?" Em: *Empirical Software Engineering* 23.1 (2018), pgs. 384–417 (citado na pg. 32).

[*Manage data in Docker* 2021] *Manage data in Docker*. Dez. de 2021. URL: https://docs.docker.com/storage/ (acesso em 29/12/2021) (citado na pg. 9).

[*Networking overview* 2021] *Networking overview*. Dez. de 2021. URL: https://docs.docker.com/network/ (acesso em 29/12/2021) (citado na pg. 9).

[*Overview of Docker Compose* 2021] *Overview of Docker Compose*. Dez. de 2021. URL: https://docs.docker.com/compose/ (acesso em 14/12/2021) (citado na pg. 9).

[*Percentage of Web Pages Loaded by Firefox Using HTTPS* 2022] *Percentage of Web Pages Loaded by Firefox Using HTTPS*. Jan. de 2022. URL: https://letsencrypt.org/stats/#percent-pageloads (acesso em 29/01/2022) (citado na pg. 23).

[*Pip documentation* 2022] *Pip documentation*. Fev de 2022. URL: https://pip.pypa.io/en/stable/ (acesso em 03/02/2022) (citado na pg. 6).

[*Pipenv documentation* 2022] *Pipenv documentation*. Fev de 2022. URL: https://pipenv.pypa.io/en/latest/ (acesso em 03/02/2022) (citado na pg. 7).

[*PostgreSQL: About* 2022] *PostgreSQL: About*. Jan. de 2022. URL: https://www.postgresql.org/about/ (acesso em 05/01/2022) (citado na pg. 10).

[Potdar *et al.* 2020] Amit M Potdar, DG Narayan, Shivaraj Kengond e Mohammed Moin Mulla. "Performance evaluation of docker container and virtual machine". Em: *Procedia Computer Science* 171 (2020), pgs. 1419–1428 (citado na pg. 18).

[*Python/Virtual environment* 2022] *Python/Virtual environment*. Fev de 2022. URL: https://wiki.archlinux.org/title/Python/Virtual_environment (acesso em 01/02/2022) (citado na pg. 6).

[Rowe e M. R. Stonebraker 1987] Lawrence A Rowe e Michael R Stonebraker. *The POSTGRES data model*. Rel. técn. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING e COMPUTER SCIENCE, 1987 (citado na pg. 10).

[Rubio 2017] Daniel Rubio. *Beginning Django: Web application development and deployment with python*. 1ª ed. APRESS, 2017. ISBN: 9781484227879 (citado na pg. 11).

[M. Stonebraker e Rowe 1986] Michael Stonebraker e Lawrence A Rowe. "The design of postgres". Em: *ACM Sigmod Record* 15 (1986), pgs. 340–355 (citado na pg. 10).

[*Tool recommendations - Python Packaging User Guide* 2022] *Tool recommendations - Python Packaging User Guide*. Fev de 2022. URL: https://packaging.python.org/en/latest/guides/tool-recommendations/ (acesso em 03/02/2022) (citado na pg. 6).

[*Virtualenv documentation* 2022] *Virtualenv documentation*. Fev de 2022. URL: https://virtualenv.pypa.io/en/latest/ (acesso em 03/02/2022) (citado na pg. 7).