Universidade da Beira Interior
Departamento de Informática

Nº 243 - 2022: *Automatic Deployment Tool for Unreliable and Unstable Networks*

Elaborado por:

Leonardo da Silva Almeida

Orientadores:

Professor Doutor Carlos Manuel Chorro Simões Barrico

Professor Doutor Rui Manuel da Silva Fernandes

11 de julho de 2022

Agradecimentos

Nesta secção inicial quero agradecer a toda a minha família pelo apoio durante o desenvolvimento deste projeto e durante o meu percurso académico. Quero deixar um especial obrigado aos meus pais, não só pelo apoio financeiro, mas por todos os abrir olhos que me deram, os conselhos, pela liberdade de poder seguir a minha área de sonho e pelo amor que me deram e irão dar.

Quero agradecer ao Professor Doutor Carlos Barrico e ao Professor Doutor Rui Fernandes por terem concordado em eu realizar este projeto e pela orientação durante o desenvolvimento do mesmo. Um obrigado ao Fernando Geraldes pelo apoio no que toca aos testes realizados no laboratório.

Não posso me esquecer de todas as pessoas a quem posso chamar de amigos, que me apoiaram e deram na cabeça. Um obrigado a todos eles por me terem deixado entrar nas vossas vidas e terem acompanhado a minha vida académica. Um especial obrigado à Beatriz Hau pela paciência e carinho ao longo destes anos, ao Francisco Teófilo por me dar a conhecer a Covilhã e pelos cafés, ao Pedro Dinis por ter estado lá sempre que era preciso e à Catarina Capinha por me integrar nesta universidade e pelo apoio incondicional.

Finalmente, um obrigado à Universidade da Beira Interior e à Covilhã por me acolherem de braços abertos.

Conteúdo

Lista de Figuras

Lista de Excertos de Código

Acrónimos

CLI *Command Line Interface*

CSS *Cascading Style Sheets*

HTML *HyperText Markup Language*

MTV *model-template-view*

PyPI *Python Package Index*

RDBMS *relational database management system*

RSS *Really Simple Syndication*

SEGAL *Space and Earth Geodetic Analysis Laboratory*

SSH *Secure Shell*

UBI Universidade da Beira Interior

URL *Uniform Resource Locator*

W3C *World Wide Web Consortium*

YAML *YAML Ain't Markup Language*

Capítulo

1

Introdução

1.1 Enquadramento

O projeto aqui exposto foi desenvolvido no âmbito da unidade curricular de projeto da Licenciatura em Engenharia Informática na Universidade da Beira Interior (UBI) para ser utilizado pelo *Space and Earth Geodetic Analysis Laboratory* (SEGAL). Este projeto enquadra-se na área de redes de computadores por se tratar de uma ferramenta para redes instáveis.

Apesar de existirem ferramentas para colmatar o problema proposto, estas nem sempre são as mais fáceis de utilizar e nem sempre se enquadram nele. Se o equipamento não estiver ligado à *Internet* constantemente, o administrador tem de estar sempre atento a verificar se o mesmo se encontra *online*. Sendo assim, fica complicado garantir a estabilidade da rede se os equipamentos estão a trabalhar em versões de *software* diferentes.

1.2 Motivação

Com visão futura de especialização profissional em administração de sistemas, por ser uma área que me fascina e de ter facilidade de compreensão em muitos temas relacionados com esta, escolhi este projeto por ser uma ferramenta que torna o trabalho de um *SysAdmin* mais eficiente e menos demorado.

1.3 Objetivos

Este projeto tem como objetivo a criação de uma ferramenta de gestão de um grupo de máquinas e testar a conectividade das mesmas. Sendo assim podemos dividir em vários passos:

1. Desenho de uma base de dados que guarde as informações das máquinas e das instruções/*software*;
2. Criação de uma interface de interação entre o utilizador e essa base de dados, isto é, inserção, visualização e remoção de elementos;
3. Lógica de no ato de inserção de um novo *software* na base de dados, criar uma *task* para enviar o mesmo para as máquinas;
4. Mostrar o sucesso ou o insucesso ao utilizador.

1.4 Organização do Documento

De modo a refletir o trabalho que foi feito, este documento encontra-se estruturado da seguinte forma:

1. O primeiro capítulo – **Introdução** – apresenta o projeto, a motivação para a sua escolha, o enquadramento para o mesmo, os seus objetivos e a respetiva organização do documento;
2. O segundo capítulo – **Estado de Arte** – expõe o atual conhecimento sobre a temática a ser abordada neste projeto;
3. O terceiro capítulo – **Engenharia de Software** – descreve o processo de planeamento do projeto e expõe alguns diagramas;
4. O quarto capítulo – **Tecnologias Utilizadas** – descreve os conceitos mais importantes no âmbito deste projeto, bem como as tecnologias utilizadas durante o desenvolvimento da aplicação;
5. O quinto capítulo – **Implementação e Testes** – explica como as tecnologias foram implementadas, com alguns excertos de código e alguns testes realizados e os seus resultados;
6. O sexto capítulo – **Instalação e Manual de Uso** – expõe como instalar a plataforma e as ferramentas acessíveis ao administrador e utilizador;

7. O sétimo capítulo – **Conclusão e Trabalho Futuro** – apresenta a discussão das conclusões obtidas durante e no final do trabalho realizado, os objetivos atingidos e aqueles que ficaram por realizar, algumas propostas para trabalho futuro e as melhorias a ser feitas.

Capítulo

2

Estado da Arte

2.1 Introdução

Hoje em dia é tomado como garantido o acesso a muitos serviços digitais, mas nem em todo lado é assim. Um administrador de redes, por vezes, não tem acesso físico aos equipamentos que gere. Assim foram criadas tecnologias e ferramentas como o *Telnet* e o *Secure Shell* (SSH). Com estas ferramentas é possível gerir máquinas remotas, instalar pacotes e alterações de ficheiros. Estas ferramentas consomem muito tempo ao administrador para a gestão de várias máquinas, tendo este de configurar cada máquina individualmente.

Com isto a solução proposta é uma plataforma *open-source*, que utilizando as ferramentas básicas de gestão, automatize de certa forma o processo repetitivo de instalar pacotes ou correr *scripts* simples em várias máquinas em simultâneo, quando estas se encontram disponíveis.

Neste capítulo pretende-se que fique exposto as ferramentas básicas existentes e algumas plataformas similares ao projeto proposto. Com este capítulo pretende-se expor essas tecnologias e explicar como este projeto se destinge.

2.2 SSH

A *Secure Shell* é um protocolo de comunicação criptográfico que permite operar de forma segura máquinas remotas. Este protocolo utiliza criptografia de chave pública para autenticar um utilizador numa máquina remota, ficando este com acesso ao terminal da máquina que pretende trabalhar.

Este protocolo é extremamente útil no que toca a gestão de máquinas, mas se o objetivo é realizar o mesmo conjunto de instruções em várias máquinas

do mesmo tipo, este processo torna-se extremamente repetitivo e cansativo.

2.3 Plataformas Atuais

2.3.1 *Cockpit Project*

O *Cockpit Project* é uma ferramenta *open-source* que permite gerir remotamente uma máquina *linux* através de uma interface *web*. Esta plataforma é útil para a gestão de uma máquina, na interface existem estatísticas sobre o uso de recursos da máquina, aplicações instaladas e até o acesso a um terminal. Mas quando é necessário gerir várias máquinas utilizando o *Cockpit*, temos de aceder à interface *web* de cada máquina para realizar operações básicas, como atualizar a versão de um pacote. Para além disso, é necessário instalar uma instância desta plataforma em cada máquina que pretendemos gerir.

Esta plataforma, para quem não se sente muito confortável com *linux*, é útil, pois permite realizar operações simples sobre apenas uma máquina ao mesmo tempo. Isto não vai de encontro com o que é pretendido, pois apenas é possível enviar instruções a uma máquina de cada vez.

2.3.2 *Ansible*

Desenvolvido pelo *Red Hat*, o *Ansible* é um motor de automação que permite automatizar, por exemplo, a instalação de pacote. É uma ferramenta que não necessita de agentes para a comunicação entre máquinas, isto é, não é necessário a instalação de uma aplicação cliente em cada máquina que é pretendido gerir.

Para trabalhar com o *Ansible* é necessário conhecer *YAML Ain't Markup Language* (YAML), um formato de serialização de dados legíveis, pois é assim que são configuradas as máquinas e os módulos de instalação para as mesmas. Apesar de existir uma interface *web* opcional, o *Ansible Tower*, o modo preferencial de trabalho desta ferramenta é através do terminal, o que pode ser complicado para utilizadores menos habituados ao mesmo.

2.4 Conclusões

Com este capítulo pretendeu-se mostrar como a plataforma desenvolvida no âmbito deste projeto se distingue de algumas ferramentas já existentes. Pretende-se que a plataforma a desenvolver seja mais simples de usar, não necessitando de instalar agentes nas máquinas cliente e apenas ser necessário conhecer as

instruções que se pretende enviar para as máquinas clientes. Para além disso ser possível enviar estas instruções para várias máquinas e continuar a tentar enviá-las mesmo quando as máquinas não se encontram disponíveis.

Capítulo

3

Engenharia de Software

3.1 Introdução

Este capitulo explora os requisitos necessários para o bom funcionamento do sistema, bem como o casos de uso, a arquitetura do sistema e finalmente, a estrutura da base de dados.

3.2 Requisitos

Os requisitos do sistema podem ser funcionais e não funcionais e como tal estes são os apresentados em baixo.

3.2.1 Requisitos Funcionais

- RF 1 - Visualizar *routers*/máquinas da base de dados;
- RF 2 - Inserir *routers*/máquinas na base de dados;
- RF 3 - Apagar *routers*/máquinas da base de dados;
- RF 4 - Visualizar *scripts*/instruções da base de dados;
- RF 5 - Inserir *scripts*/instruções na base de dados;
- RF 6 - Apagar *scripts*/instruções da base de dados;
- RF 7 - Visualizar estado de sucesso do envio de *scripts*/instruções;
- RF 8 - Visualizar *output* do pós envio de *scripts*/instruções dos/das *routers*/ máquinas;

- RF 9 - *Log-in* e *Log-out* do sistema.

3.2.2 Requisitos Não Funcionais

- RNF 1 - Criação e inserção de um objeto que liga o/a *router*/máquina com o/as *script*/instruções na base de dados;
- RNF 2 - Envio e no caso de falha, tenta o envio recursivamente do/das *script*/instruções para o/a *router*/máquina;
- RNF 3 - Guardar o *output* do pós envio de *scripts*/instruções dos/das *routers*/ máquinas.

3.3 Arquitetura

Nesta secção é exposta a arquitetura do sistema com o auxílio da figura 3.1. Como é possível observar em baixo o cliente comunica com o servidor através do *browser*. O servidor faz a gestão da comunicação entre a *relational database management system* (RDBMS) (*MariaDB* 4.5) e a *framework Django* (4.2), que por sua vez, comunica com as máquinas que estão no exterior.

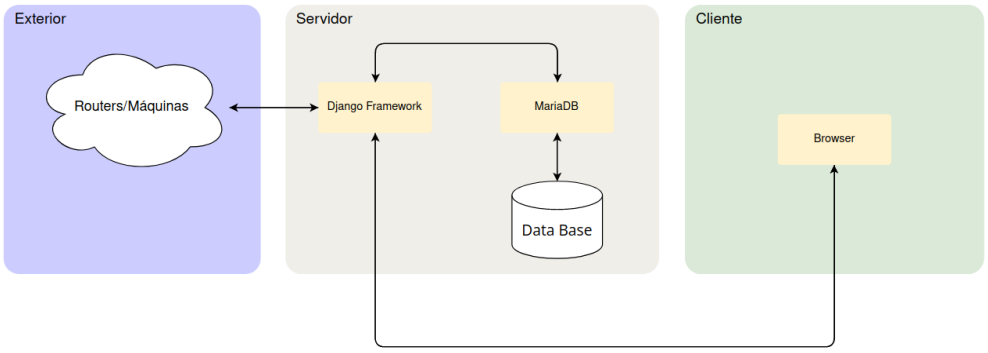

Figura 3.1: Diagrama representativo da arquitetura do sistema.

3.4 Casos de Uso

Nesta secção é apresentado o diagrama de casos de uso. Como pode ser visto na imagem 3.2, o utilizador consegue:

- Fazer o *login* no sistema;

- Gerir os *Routers* (visualizar, adicionar e remover);
- Gerir os *Scripts* (visualizar, adicionar e remover);
- Visualizar os *logs* que inclui o *output* do que é enviado para os *Routers*.

Já o administrador pode gerir os utilizadores e fazer o *login* no sistema.

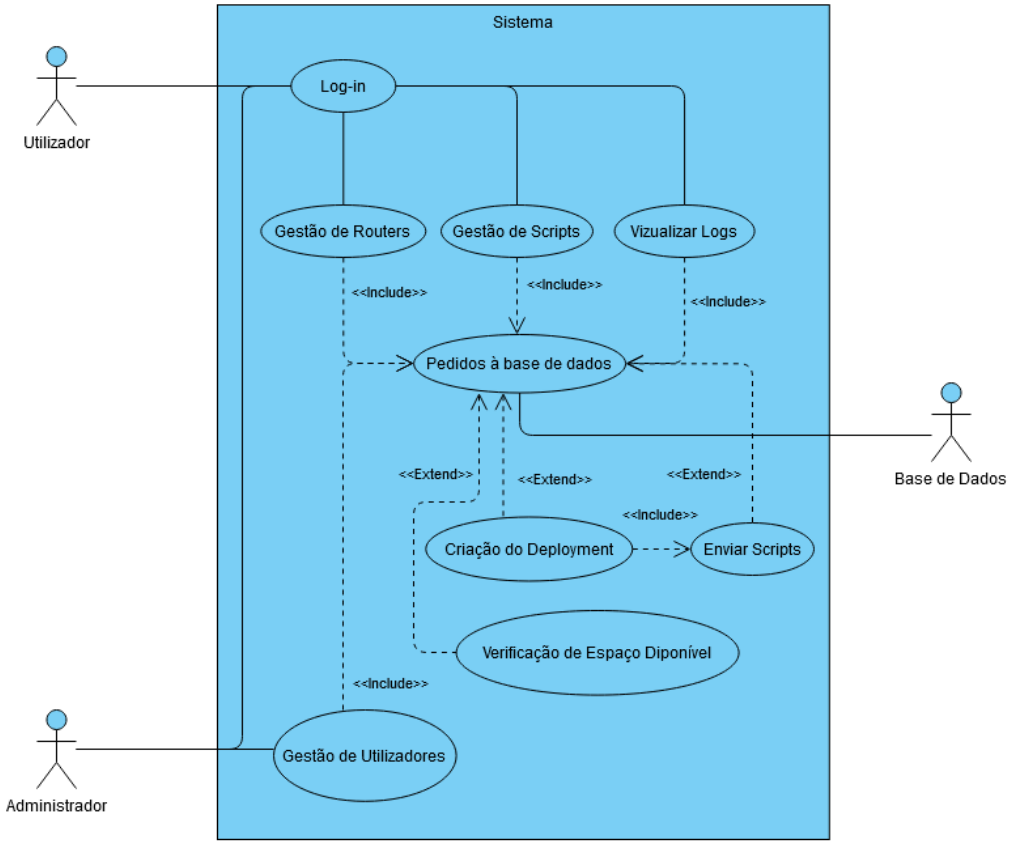

Figura 3.2: Diagrama de casos de uso.

3.5 Base de dados

Aqui é apresentado a estrutura física da base de dados. Esta é composta por três tabelas relacionais. Como podemos observar na figura 3.3 a primeira tabela (*Router*) é utilizada para guardar as informações das máquinas ou *routers* com nove elementos. Já a terceira tabela (*Scripts*) é utilizada para armazenar as informações relativas aos *scripts* ou instruções que são adicionados pelo

utilizador bem como o *path* para o ficheiro em si. Esta é constituída por seis elementos. Finalmente na segunda tabela (*Deployment*) é utilizada para armazenar as informações sobre a entrega das instruções às máquinas e é constituída por duas chaves estrangeiras e outros quatro elementos.

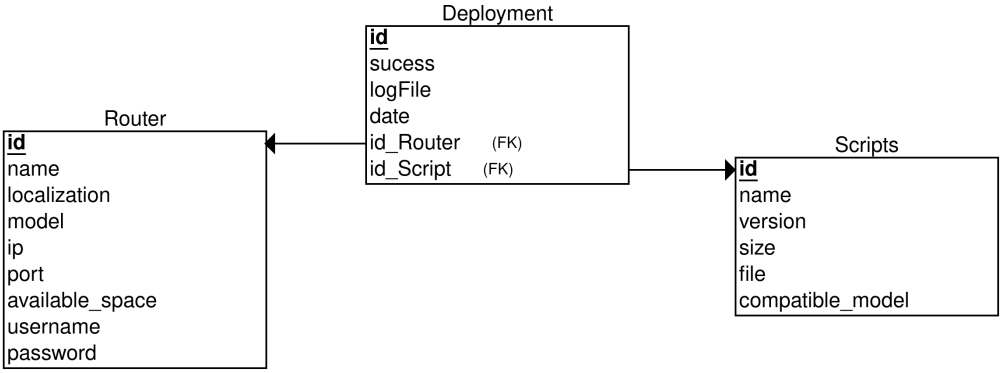

Figura 3.3: Estrutura da base de dados.

3.6 Conclusões

Com este capítulo pretendeu-se que ficasse retido uma ideia de como o sistema atual está desenvolvido, através dos requisitos, o diagrama de casos de uso, a arquitetura do sistema e a estrutura da base de dados facilitando assim a compreensão do sistema em causa.

Capítulo

4

Tecnologias e Ferramentas Utilizadas

4.1 Introdução

Neste capítulo é exposto as várias tecnologias utilizadas na realização deste projeto, dando destaque à linguagem *Python*, à *web framework Django* e à livraria *Netmiko*. É descrita a ferramenta utilizada para alojar a base de dados e é exposto um pacote essencial para a elaboração do projeto, o *Django Background Tasks*. Também se expõe outra ferramentas e linguagens utilizadas mas de forma mais sucinta.

4.2 *Django*

Django é uma *web framework* [1], *open-source* baseada em *Python* (4.3) que segue o padrão de arquitetura *model-template-view* (MTV) e é mantida pela *Django Software Foundation*. Lançada em 2005, foi desenhada para ser de rápido desenvolvimento e para ser usada em aplicações baseadas em base de dados. Vem incluída com vários extras para acelerar o desenvolvimento como a autenticação de utilizadores e *Really Simple Syndication* (RSS) *feeds*. É utilizado nomeadamente pelo *Instagram* e pela *Mozilla* para os seus *websites*. Foi a escolhida por ser de rápido desenvolvimento e diminuir vários passos repetitivos ao trabalho do programador, como a conexão à base de dados e a criação das suas tabelas, a gestão de utilizadores e autenticação. Estas funcionalidades e outras podem ser consultadas na documentação do *Django* [2].

4.3 *Python*

Python é uma linguagem *high-level*, ou seja, é fortemente abstrata, interpretada, isto é, não necessita de ser compilada para ser executada, mas precisa de um interpretador que corre diretamente as instruções escritas e é multi-paradigma, moldando-se facilmente ao que o programador necessita. Foi apresentada em 1991 por Guido von Rossum e hoje é desenvolvida pela *Python Software Fundation*. É uma linguagem acessível que inclui uma extensa biblioteca padrão o que a torna numa linguagem fácil de aprender.

4.4 *Netmiko*

Como mencionado na secção a cima, o *Python* (4.3) tem uma vasta biblioteca de pacotes, disponíveis no *Python Package Index* (PyPI), que aumenta as funcionalidades da linguagem. Um destes pacotes é o *Netmiko*, um pacote que permite algumas automações de rede em dispositivos que suportam *screen-scraping* (ato de copiar informação que é apresentado no ecrã do dispositivo e pode ser usada para outro propósito) para captar o *output* dos comandos enviados para outros dispositivos em rede. Este pacote utiliza o protocolo de rede SSH para enviar comandos e receber o *output* desses mesmos, e suporta vários tipos de dispositivos como *Linux* e *Cisco IOS*. É possível consultar a documentação em [3].

4.5 *MariaDB*

MariaDB é um servidor de base de dados que teve origem num *fork* do *MySQL* RDBMS. Foi desenvolvido com o intuito de ser *open-source* e grátis pois em 2009 existiu preocupações com uma possível aquisição do *MySQL* RDBMS pela parte da *Oracle Corporation*. Desde então tem sido mantido pela *MariaDB Foundation* [4] e tenta manter alta compatibilidade com o *MySQL*, mas mesmo assim consegue utilizar novos recursos como o *ColummStore*.

4.6 *Django Background Tasks*

É um pacote que pertence ao PyPI e trabalha especificamente com a *framework Django*. Utiliza uma base de dados para criar tarefas que necessitam de ser executadas no *background* e múltiplas vezes num dado espaço de tempo. É possível consultar a documentação em [5].

4.7 *HTML* e *CSS*

O *HyperText Markup Language* (HTML) é a linguagem de marcação padrão para todas as páginas *web*. Utiliza semântica para descrever a estrutura da página num ficheiro recebido de um servidor. Esse ficheiro é então utilizado pelo *browser* que o transforma numa representação visual e interativa para o utilizador. Esta linguagem é assistida por outra tecnologia, o *Cascading Style Sheets* (CSS) para estilizar o que é apresentado ao utilizador. O CSS é mesmo isso, uma linguagem de folhas de estilo. É desenvolvido pelo *World Wide Web Consortium* (W3C) e foi lançado em 1996.

4.8 Outras Ferramentas

Querendo dar mais foco nas tecnologias e ferramentas já mencionadas, esta secção descreve ferramentas utilizadas, mas podem ser consideradas como secundárias. Esta são:

- ***Visual Studio Code*** é um editor de texto leve mas versátil com várias extensões para suportar múltiplas linguagens de programação e de marcação. Foi escolhido por essa descrição, por ser fácil de utilizar e ter o *highlighting* que permite rapidamente identificar erros de sintaxe;
- ***Firefox*** é um *web browser open-source* desenvolvido pela *Mozilla Foundation* e por um grupo vasto de colaboradores voluntários. É leve, rápido e foi essencial para o desenvolvimento deste projeto por se tratar de uma aplicação *web*;
- ***Bulma*** é uma *framework* grátis e *open-source* que fornece componentes pré-feitos de *front-end* para a criação mais rápida de interfaces interativas. Foi possível a implementação com a ajuda da documentação [6].

4.9 Conclusões

Neste capítulo foi realizada uma breve explicação das tecnologias utilizadas no desenvolvimento do projeto, nomeadamente da *framework Django* a qual está escrita em *Python* e é a base deste projeto. Essencial para a este projeto foi o pacote *Netmiko* que facilita as conexões por *Command Line Interface* (CLI). Para além disto, foi exposto a ferramenta de eleição para alojar a base de dados, a *MariaDB* e as várias outras tecnologias de auxilio ao desenvolvimento.

Capítulo

5

Implementação e Testes

5.1 Introdução

Neste capítulo vai ser explicado como este sistema funciona e como foi construido, explicando como as tecnologias e ferramentas (capítulo 4) foram implementadas e expondo alguns excertos de código para mostrar o tal. Aqui também é mostrado os testes feitos para comprovar que o sistema desenvolvido cumpre os objetivos propostos.

5.2 Uso da *Framework Django*

Como explicado no capítulo anterior (4.2), a *framework* utiliza um modelo MTV então para a construção do sistema foram necessários a construção de *models*, *views* e *templates*.

Para além destes três pilares o *Django* oferece outras ferramentas chamadas de *apps* que facilitam a criação de um aplicação *web* como *Uniform Resource Locator* (URL) personalizados e dinâmicos, uma página web administradora pré-criada para o auxilio da gestão da plataforma criada e um gestor de utilizadores. Este leque de ferramentas pode ser expandido com a instalação de pacotes extra que foi o caso deste projeto, como referido na secção 4.6.

5.2.1 *Models* e Ligação à Base de Dados

Os *models* não deixam de ser as tabelas da base de dados com alguma *metadata* adicional. Cada *model* é representado por uma classe que é uma subclasse do *django.db.models.Model*. Cada classe tem as suas variáveis definidas tal como está exposto na secção 3.5. Estas variáveis são uma instância da classe *Field* que pertence à livraria do *Django* 4.2. É possível ver um exemplo disso no excerto 5.1.

```
class Router(models.Model):
    name = models.CharField(max_length=25)
    localization = models.CharField(max_length=25)
    model = models.CharField(max_length=25)
    ip = models.GenericIPAddressField()
    porta = models.IntegerField(default=22)
    available_space = models.DecimalField(max_digits=10,
        decimal_places=2)
    username = models.CharField(max_length=25)
    password = models.CharField(max_length=30,default='R@nd0mP@ss')
```

Excerto de Código 5.1: *Model Router*.

Como a ligação da base de dados é tratada pela *framework*, sendo apenas preciso configurar os dados de acesso à mesma, a *framework* consegue criar as tabelas na base de dados e as *querys* necessárias para o resto do sistema. Futuramente quando for necessário aceder à base de dados é mais simples pois não é preciso escrever a *query*, basta chamar uma função (6).

No servidor *MariaDB* (4.5) foi apenas necessário criar a base de dados e um utilizador para a aceder, não foi preciso qualquer outra interação com a RDBMS.

Para a criação das tabelas na base de dados foi preciso fazer uma migração dos objetos criados, isto foi atingido com a chamada dos comandos no terminal:

```
python3 manage.py makemigrations
python3 manage.py migrate
```

O primeiro encapsula todas as alterações e/ou criações necessárias as tabelas e o segundo realiza as mesmas.

5.2.2 *Views*

As *views* são funções que a *framework* executa sempre que é aberta uma página *web*. Isto permite usar alguma lógica em cada página apresentada, como por exemplo como tratar os dados recebidos de um formulário ou que dados serão apresentados na página em questão. Estas funções trabalham em conjunto com os *templates* (5.2.4), para apresentar ao utilizador o que é pretendido em cada página.

No excerto de código 5.4 é apresentado como isto é feito. Em todas as funções *view* é recebido pelo menos três parâmetros para ser mais fácil perceber o que está ser pedido ao sistema e que utilizador realiza esse pedido.

A primeira função (5.2) é utilizada para guardar na base de dados um novo *router*, utilizando um *form* (5.4) onde é especificado que parâmetros o utilizador preenche.

Já a segunda função (5.3) é utilizada para apresentar os *routers* guardados na base de dados, para tal é utilizada uma função para ir buscar todos os elementos na tabela *Router*. Seguidamente é devolvido a função *render* que faz mesmo isso, faz *render* a um *template* dado como parâmetro, bem como um dicionário que inclui a lista de todos os elementos. Como estes são apresentados é explicado mais à frente na subsecção 5.2.4.

```
    def add_router(request,*args, **kwargs):
        print(args, kwargs)
        print(request.user)

        form = RawRouterForm()
        if request.method == "POST":
            form = RawRouterForm(request.POST)

            if form.is_valid():
                router = Router.objects.create(**form.cleaned_data)
                print("Saved new router")
                router.save()
                id = router.id
                check_available_space(id)
                return redirect("/pages/router_db/")
            else:
                print(form.errors)
        context = {
            'form': form
        }

        return render(request, "pages/router_form.html", context)
```

Excerto de Código 5.2: *View* de formulário de inserção de *routers*.

```
    def router_db(request,*args, **kwargs):
        print(args, kwargs)
        print(request.user)

        context = {
            "list" : Router.objects.all()
        }

        return render(request, "pages/router_db.html", context)
```

Excerto de Código 5.3: *View* da apresentação dos *routers*.

Neste excerto é apresentado a classe que funciona de maneira similar ao *model*. Esta classe é instanciada na função de *view* que comunica como é feito o *render*. A classe *Form* para o *router* é similar, adaptando as variáveis para o *script*.

```
    class RawRouterForm(forms.Form):
        name = forms.CharField(max_length=25)
        localization = forms.CharField(max_length=25)
        model = forms.CharField(max_length=25)
        ip = forms.GenericIPAddressField()
        porta = forms.IntegerField()
        username = forms.CharField(max_length=25)
        password = forms.CharField(max_length=30, widget=forms.
            PasswordInput())
```

Excerto de Código 5.4: *Class Form* do *Script*

Este processo é similar para a apresentação dos *scripts* e dos *logs*. Na função de *view* de adição de *scripts*(5.5) tem a particularidade de criar também o objeto *deployment* que guarda a relação entre o *router* e o *script*. É criado e inserido na base de dados um *deployment* para cada *router* que o *script* é compatível. Com isto é chamada a função *send to router* (5.11), que é explicada mais à frente, para cada objeto *deployment* criado.

```
...
routers_update = Router.objects.filter(model=form.cleaned_data.get('
    compatible_model'))
for dp_router in routers_update:
    new_dp = Deployment.objects.create(router=dp_router,update=
        new_script,date=date.today())
    send_to_router(new_dp.id, verbose_name="Deployment", repeat=60,
        repeat_until=date.today()+timedelta(minutes=60))
    print("New Deployment " + str(new_dp.router.name))

return redirect("/pages/script_db/")
...
```

Excerto de Código 5.5: Excerto da função de *view* da adição de *scripts*.

Para eliminar um objeto da base de dados também é necessário uma *view*. Isto é implementado de maneira simples, como pode ser comprovado no excerto 5.6. Como trabalhamos com uma base de dados relacional, primeiro é necessário eliminar os objetos na tabela *deployment* que estão associados ao *script* ou *router* que é pretendido eliminar. Em seguida é eliminado o objeto pretendido.

Esta função de *view* não faz *render* a um *template* mas redireciona para outra página, isto é explicado melhor mais à frente na secção 5.2.4.

```
def delete_script(request, script_id):
    print(request.user)

    my_script = Script.objects.get(id=script_id)

    if Deployment.objects.filter(update=my_script).exists():
        dp = Deployment.objects.get(update=my_script)
        dp.delete()

    my_script.delete()

    return redirect('/pages/script_db')
```

Excerto de Código 5.6: Função de *view* de eliminação de um *script*.

5.2.3 Personalização de *URLs*

Como mencionado na introdução desta secção (5.2), a *framework* permite criar URLs personalizados à necessidade e gosto do programador. Assim é possível criar um mapa personalizado das funções de *view*, por exemplo, se

o utilizador pretender aceder diretamente, se estiver autenticado, pode aceder diretamente ao formulário de inserção de um *script* ou visualizar os *logs*. Para isto basta criar uma lista que "liga"as *views* com um nome. Isto pode ser visualizado no excerto 5.7.

```
urlpatterns = [
    path('',views.router_db),
    path('router_form/',views.add_router),
    path('router_db/',views.router_db),
    path('script_form/',views.add_script),
    path('script_db/',views.script_db),
    path('logs/',views.deployment_db),
    path('delete_router/<int:router_id>', views.delete_router, name=
        'delete_router'),
    path('delete_script/<int:script_id>', views.delete_script, name=
        'delete_script'),
    path('view/<int:dp_id>', views.show_file, name='show'),
]
```

Excerto de Código 5.7: Excerto de código de apresentação dos URLs personalizados.

Pode ser observado na linha 9 que é possível enviar parâmetros para as *views* através dos URLs. Este é o caso de como é possível eliminar um objeto da base de dados (exposto no excerto 5.6).

5.2.4 *Templates*

O terceiro componente da arquitetura do *Django* (4.2) são os *templates*. Tal como referido na subsecção 5.2.2 os *templates* trabalham juntos com as funções de *view*, enquanto estas últimas contêm a lógica principal de como a página funciona, os *templates* descrevem a formatação do *front-end*. Estes ficheiros estão escritos em HTML (4.7) e o embelezamento é feito com a assistência da *framework Bulma* (4.8).

A *framework Django* (4.2) contém uma linguagem de *markup* embutida que trabalha em conjunto com o HTML para criar mais alguma lógica ao *front-end*. Com isto é possível "enviar" uma lista de elementos e na própria marcação apresentar todos os elementos numa tabela de uma maneira mais dinâmica, como pode ser visualizado no excerto 5.8.

Aqui também é possível "chamar" outras *views* como podemos ver na linha 14 do excerto 5.8 que redireciona para a *view* referida no excerto 5.6.

```
<table class="table is-coverable is-fullwith">
...
    {% for router in list %}
    <tr>
        <td colspan="2">{{ router.id }}</td>
        <td colspan="2">{{router.name}}</td>
        <td colspan="2">{{router.localization}}</td>
        <td colspan="2">{{router.model}}</td>
        <td colspan="2">{{router.ip}}</td>
        <td colspan="2">{{router.porta}}</td>
        <td colspan="2">{{router.available_space}}</td>
        <td colspan="2">
            <a class="button is-danger is-small" href="{% url '
                delete_router' router.id %}">Remove</a>
        </td>
    </tr>
    {% endfor %}
</table>
```

Excerto de Código 5.8: Excerto de código de apresentação de elementos no *front-end*.

Para além de facilitar a apresentação de elementos, em conjunto com a função de *view* respetiva, também facilita a captação de dados para serem tratados na *view*. Assim é apenas necessário indicar para apresentar um formulário já cedido ela função que faz *render* à página. É possível verificar isto com o excerto de código 5.9.

```
<form method="POST">
    {% csrf_token %}
    ...
        <div class="block">
            <h3 class="title">Router adding form</h3>
        </div>
            <table class="table">
                {{ form }}
                <td class="buttons">
                    <input class="button is-success" type="submit"
                        value="Save">
                </td>
            </table>
    ...
</form>
```

Excerto de Código 5.9: Excerto de código de captação de dados no *front-end*.

Com uma lógica similar o *login* do utilizador também é assim realizado com o auxilio de uma plataforma de gestão de utilizadores já embutida na *frameworrk Django*. Esta trata da segurança das palavra-passe e da verificação se o utilizador está autenticado. Para essa verificação é apenas necessário a adição de uma lógica simples no ficheiro *template* como é possível verificar no excerto 5.10. Se o utilizador não se encontrar autenticado este é redirecionado para uma página de *log-in*.

```
{% block content %}
{% if user.is_authenticated %}
...
{% else %}
    {% include 'notLogin.html' %}
{% endif %}
```

Excerto de Código 5.10: Excerto de autenticação.

Tirando proveito da possibilidade de integração de lógica para dinamizar as páginas, também foi implementado um método de redução de código escrito. É possível com a *framework* criar um ficheiro base com o *head* do HTML e este ser a base para todos os outros ficheiros. Isto é, todos os outros ficheiros com a formatação de apresentação de dados e entrada de dados estendem desse ficheiro base. Assim foi possível criar uma barra de navegação que é incluída no início de cada ficheiro.

5.2.5 *Background Tasks*

O pacote *Django Background Tasks* (4.6) permite criar funções que correm no *background* da *framework*. Para tal é necessário especificar que a função é uma *task* (1). Quando a função é chamada, a *framework* cria a *task* e armazena um objeto na base de dados. Este objeto contém a informação sobre a função para mais tarde correr a *task*.

5.3 Implementação do *Netmiko*

O *Netmiko* é a biblioteca que permite a conexão com os *routers*. Com isto, foi criada uma função que recebe um parâmetro do objeto *deployment* e com essa informação transmite as variáveis necessárias para a conexão, onde se encontra o ficheiro *script* e onde guardar o *output*. No final atualiza o objeto *deployment* com o sucesso ou insucesso da *task*. Isto tudo pode ser observado no excerto 5.11.

```
@background()
def send_to_router(dp_id):
    dp = Deployment.objects.get(id=dp_id)
    dp_router = dp.router
    router_info = {
        'device_type': 'autodetect',
        'ip': str(dp_router.ip),
        'username': dp_router.username,
        'password': dp_router.password,
        'port': dp_router.porta,
    }

    guesser = SSHDetect(**router_info)
    best_match = guesser.autodetect()
    router_info['device_type']=best_match
    net_connect = ConnectHandler(**router_info)
    net_connect.enable()

    file_path = os.path.join(BASE_DIR, 'uploads/' + str(dp.update.
        file))
    instruction = open(file_path, 'r')
    lines = instruction.readlines()
    output = net_connect.send_config_set(lines)
    instruction.close()

    log_path = os.path.join(BASE_DIR, 'uploads/output/' + str(
        dp_router.name))
    log = open(log_path,'a')
    log.write('\n' + str(datetime.now()) + '\n')
    log.write(output)
    log.close()

    dp.success = True
    dp.logFile = log_path
    dp.save()
```

Excerto de Código 5.11: Função de conexão ao *router*.

5.4 Testes

Nesta fase final do projeto, foi necessário realizar alguns testes para validar o bom funcionamento da plataforma e se esta cumpre os objetivos propostos (1.3). Para a validação foi utilizado, principalmente, a página administradora (6.2), para verificar os resultados esperados em relação à adição e remoção dos objetos na base de dados. Para validar que as instruções foram realizadas

na máquina remota foi utilizado um *Raspberry Pi* com as seguintes especificações:

```
Modelo: Raspberry Pi 3 Model B Rev 1.2
CPU: BCM2835
Memoria: 1GB
Sistema Operativo: Rasbian 11
```

Este dispositivo é para onde as instruções foram enviadas e verificadas se foram realizadas. Estes testes estão descritos nas subsecções em baixo.

5.4.1 Teste de Adição e Remoção de *Routers* e *Scripts*

Para testar estas funcionalidades foram realizados testes iguais, tanto para a gestão de *routers*, como para a gestão de *scripts*. Como tal apenas será explicados os testes para os *routers*.

O primeiro teste foi a adição de uma nova máquina à base de dados. Para isso foi necessário preencher o formulário para o efeito (figura 6.5). Para verificar se o *router* foi adicionado corretamente, foi utilizada a página de apresentação de *routers* (figura 6.4) e a página administradora (figura 6.2) onde se pode verificar também se a máquina foi adicionada.

O segundo teste foi a eliminação de um objeto da base de dados. Para isso foi necessário aceder à página de apresentação de *routers* (figura 6.4) e selecionar o botão de remover. Como o *router* desapareceu tanto na página de apresentação, como na página administradora então o teste foi bem sucedido.

5.4.2 Teste de criação do Objeto *Deployment*

Para testar o sucesso desta funcionalidade foi adicionado um *router* e um *script* à plataforma. Após adicionar foi verificado na página administradora (figura 6.2) e na página de *logs* (figura 6.6), que o objeto *deployment* que associa o *router* e o *script* foi criado.

5.4.3 Teste de Envio de Instruções e Validação de Instruções

Juntamente com a criação do objeto *deployment* é criada a *task* que envia as instruções. O sucesso desta operação é verificada, novamente, na página administradora. Agora se foi realizada as instruções e se a comunicação foi realizada, é verificado os dados do objeto *deployment* atualizado na página de *logs*. Com isto, foi verificado no *Raspberry Pi* se as instruções foram recebidas e executadas, examinando o *output* esperado com o recebido, validando assim esta função.

5.5 Conclusões

Com este capítulo foi exposto como as tecnologias(capítulo 4) foram implementadas, nomeadamente a *framework Django* e o pacote *Netmiko*. Foi descrito em pormenor os três pilares da *framework* e como estes trabalham em conjunto para apresentar a informação ao utilizador e como é enviado os *scripts* para os *routers*. Neste capítulo também foram apresentados alguns testes realizados com o objetivo de verificar se os objetivos propostos foram atingidos.

Capítulo

6

Instalação e Manual de Uso

6.1 Introdução

Com vista a ser mais esclarecedor como dar *deployment* à *web app*, este capítulo tem como objetivo explicar quais as dependências necessárias para o bom funcionamento da *web app*. Para além disso é explicado, de maneira sucinta, como utilizar a *web app*.

6.2 Instalação das Dependências e Configuração da *Framework*

Algumas das dependências deste projeto já foram expostas no capítulo 4. A ferramenta foi desenvolvida em *linux* mas também pode ser corrida num sistema *Windows*. É necessário instalar o pacote *Python* (e o pacote PyPI se necessário), utilizando o gestor de pacotes, no caso de um sistema *linux*.

Com estes instalados, é preciso instalar os pacotes de *Python* utilizando o comando *pip install*. Os pacotes necessários são:

1. *django*;
2. *django-background-tasks*;
3. *mysqlclient*;
4. *netmiko*.

Com estes pacotes instalados, é necessário a instalação do servidor de base de dados, que neste caso foi utilizado o *MariaDB* (4.5). Este *software* é

instalado utilizando, novamente, o gestor de pacotes, no caso de um sistema *linux*.

Com o *MariaDB* instalado é necessário configurar o servidor de base de dados chamando o comando *mysql_secure_installation*, onde é definido a palavra passe de *root*. Em seguida é criado a base de dados e o utilizador para acesso à mesma, para tal é preciso entrar na consola da *MariaDB* utilizando o comando *mysql -u root -p* e em seguida utilizar a seguinte sequência de instruções dentro da consola:

```
CREATE DATABASE ADT;
USE ADT;
CREATE USER 'user'@'localhost' IDENTIFIED BY 'password';
GRANT ALL PRIVILEGES ON ADT TO 'user'@'localhost' WITH
GRANT OPTION;
exit;
```

Com o utilizador criado, precisamos de indicar à *framework* onde está a base de dados, qual a base de dados e que utilizador usar. Isto é possível alterando o ficheiro *settings.py*, da forma exposta no excerto de código 6.1.

```
...
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'ADT',
        'USER': 'user',
        'PASSWORD': 'password',
        'HOST': 'localhost',
        'PORT': '',
    }
}
...
```

Excerto de Código 6.1: Conexão á base de dados no ficheiro *settings.py*

Agora com a base de dados configurada na *framework*, é preciso migrar os objetos para a base de dados utilizando os mesmos comandos expostos na secção 5.2.1 e criar um utilizador administrador para a *framework* com este comando:

```
python3 ./manage.py createsuperuser
```

Finalmente, basta iniciar o servidor *Django* e iniciar a componente das *tasks* com os comandos em baixo apresentados (é necessário correr estes comandos em janelas diferentes). No primeiro comando é possível especificar o endereço da interface e a porta que é usada para a mesma.

```
python3 ./manage.py runserver 0.0.0.0:8000
python3 ./manage.py process_tasks
```

O acesso à interface *web* é feito com o auxilio de um *browser* utilizando o endereço e a porta especificada. Com isto tudo realizado é apresentado a *home page*, como se pode observar na imagem 6.1.

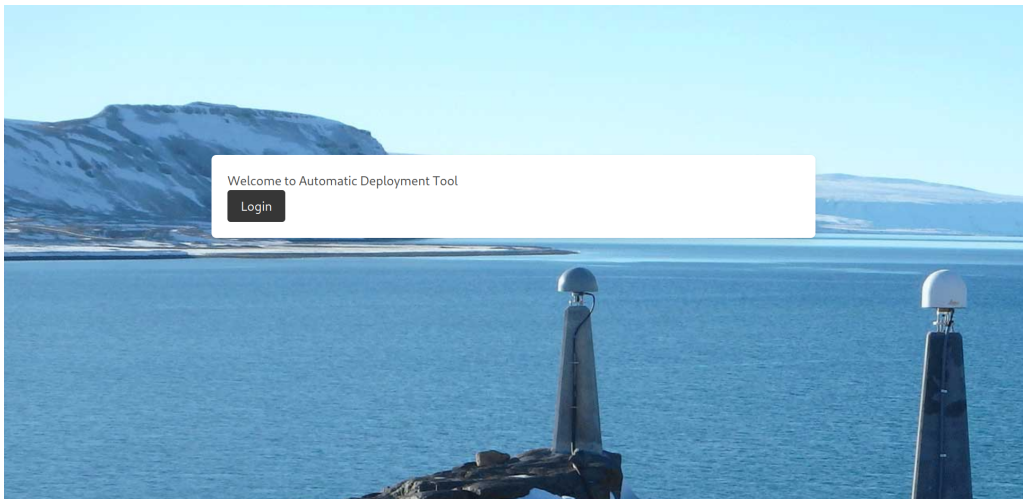

Figura 6.1: Página inicial

6.3 Manual do Administrador

O administrador para além de ter as mesmas permissões de visualizar os dados como um utilizador na página dita normal, este também tem acesso a uma página administradora onde pode gerir os utilizadores, grupos e permissões deles. Esta página pode ser acedida juntando no final do endereço raiz um */admin* e deve ser apresentada uma página com o aspeto da figura 6.2, após o *login*.

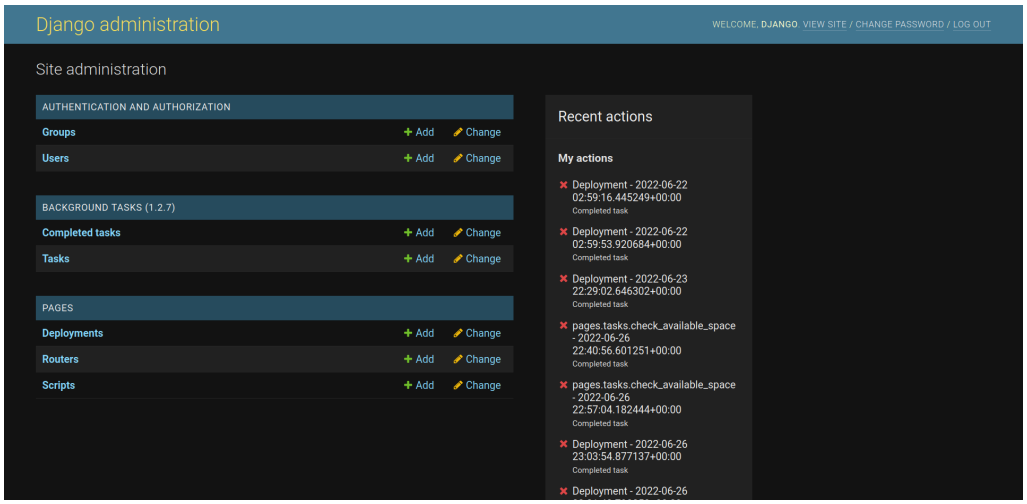

Figura 6.2: Página administradora

6.4 Manual do Utilizador

Do lado do utilizador, este pode gerir que *routers* e *scripts* estão inseridos na base de dados. Para isso, o utilizador precisa de inserir os dados de autenticação na página de *login* (imagem 6.3).

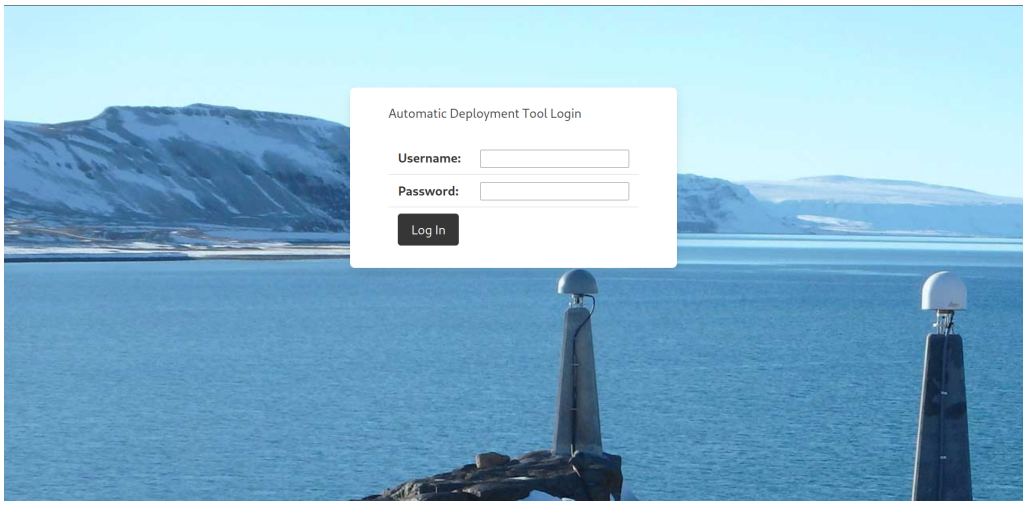

Figura 6.3: Página de *login*.

Esta gestão pode ser realizada acedendo a quatro páginas, aqui é apresentado apenas duas delas (figura 6.4 e figura 6.5) pois o aspeto e funcionamento

são similares. É possível observar que na barra de navegação contém os botões para aceder a todas as páginas necessárias ao utilizador, indica que utilizador está autenticado e a possibilidade de fazer o *log-out* para sair da *web app*.

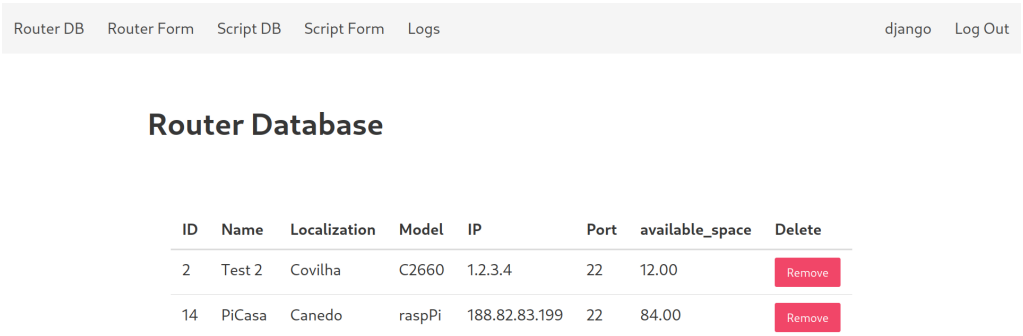

Figura 6.4: Página de apresentação dos *routers*.

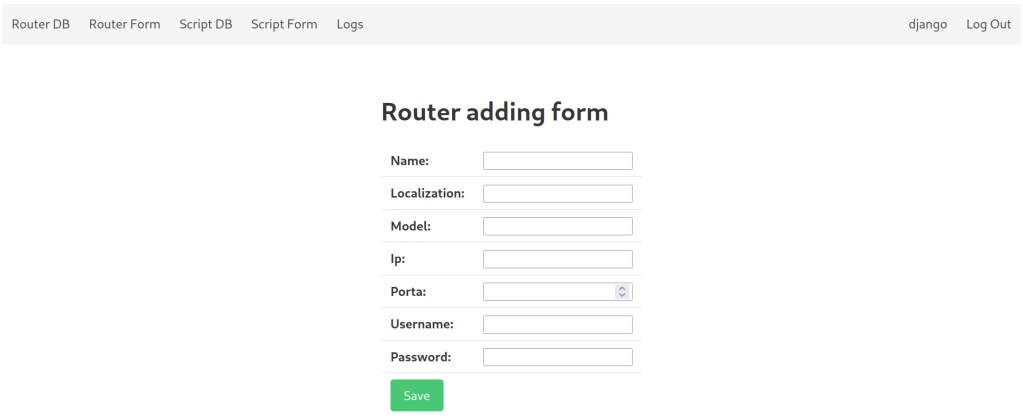

Figura 6.5: Página de criação de um novo *router*.

Finalmente o utilizador pode aceder ao estado da *task* que é criada automaticamente quando adiciona um *script*, acedendo aos *logs*. Esta página tem o aspeto apresentado na figura 6.6. Na coluna *success* é indicado se o *script* foi

enviado com sucesso para o *router* e na coluna *Log File* é possível aceder ao ficheiro que contém o *output* desse *script*.

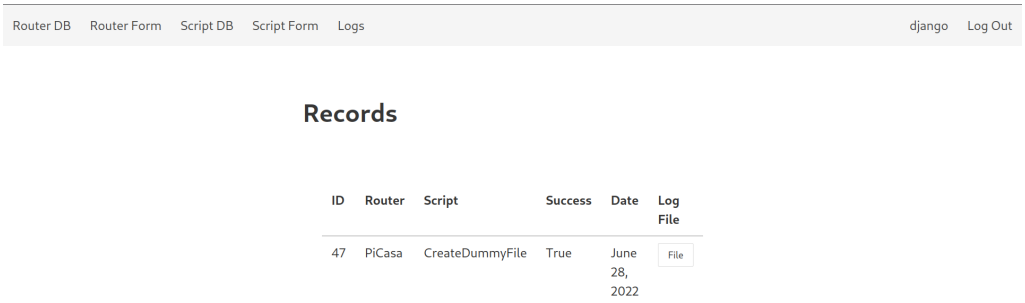

Figura 6.6: Página de apresentação dos *logs*.

6.5 Conclusão

Com a conclusão deste capítulo, pretende-se que tanto o administrador, como os utilizadores do sistema fiquem familiarizados com a *web app* e com as ferramentas de gestão desta.

Capítulo

7

Conclusões e Trabalho Futuro

7.1 Conclusões Principais

Com o desenvolvimento deste projeto foi possível aprofundar o conhecimento da linguagem *Python* (4.3) e aprender novas tecnologias como a *framework Django* (4.2). Também foi criada uma perceção do mercado de *softwares* e ferramentas disponíveis para a gestão de várias máquinas e algumas deficiências nas próprias ferramentas.

7.2 Trabalho Futuro

Com a conclusão deste projeto, foi possível encontrar possibilidades de expansão para tornar a ferramenta mais robusta. Estas são as propostas futuras para o projeto:

- Criar métodos para interpretar o *output* dos *scripts* ou instruções com o intuito de apresentar um *feedback* mais simples;
- Apresentar pacotes já instalados no *router* na ferramenta;
- Criar um método de seleção de instalação de um pacote específico, sendo desnecessário a criação de um *script* para tal;
- Criar um método para agrupar máquinas e fazer o envio de instruções;
- Permitir a personalização da frequência e a duração das tentativas de conexão à máquina, por parte do utilizador.

Bibliografia

[1] Django Software Foundation. Django overview, 2022. [Online] https://www.djangoproject.com/start/overview/, Ultimo acesso: 07/07/2022.

[2] Django Software Foundation. Django documentation, 2022. [Online] https://docs.djangoproject.com/en/4.0/, Ultimo acesso: 07/07/2022.

[3] Natasha Samoylenko. Module netmiko, 2022. [Online] https://pyneng.readthedocs.io/en/latest/book/18_ssh_telnet/netmiko.html, Ultimo acesso: 07/07/2022.

[4] MariaDB Foundation. About mariadb server, 2022. [Online] https://mariadb.org/about/, Ultimo acesso: 07/07/2022.

[5] Múltiplos Autores. Django background tasks, 2022. [Online] https://django-background-tasks.readthedocs.io/en/latest/, Ultimo acesso: 07/07/2022.

[6] Jeremy Thomas. Bulma documentation, 2022. [Online] https://bulma.io/documentation/, Ultimo acesso: 07/07/2022.