PROPOSTA DE UM *LOGBOOK* DIGITAL PARA OS MOTORES PW118 DA FAB

PROPOSAL OF A DIGITAL *LOGBOOK* FOR THE PW118 FAB ENGINES

Guilherme da Mata Gonzaga de Souza [1]
Leila Scanfone [2]
Luísa Amaral de Almeida [3]

RESUMO

Este trabalho aborda a criação de *logbook* digital para os motores PW118 da frota da Força Aérea Brasileira (FAB). Tal abordagem se faz necessária devido ao grande impacto financeiro gerado pela gestão deficiente da informação destes motores e de falta de ferramentas que garantam a confiabilidade, controle e acessibilidade destes dados. O objetivo deste trabalho é apresentar uma proposta de metodologia para criação de um *logbook* digital, a fim de substituir os atuais documentos físicos dos motores PW118 utilizados pela FAB. Este propósito será alcançado mediante pesquisa bibliográfica acerca do *logbook* na aviação, da tecnologia de banco de dados relacional e de bibliotecas da linguagem Python capazes de implementar o *logbook* na versão digital. Em seguida, aplicou-se a pesquisa documental nos arquivos físicos dos motores PW118 e criação do banco de dados em linguagem Python, por meio da biblioteca Django. O estudo evidenciou que o *framework* Django é uma ferramenta apropriada para o desenvolvimento da metodologia proposta, capaz de armazenar e gerenciar os dados contidos nos *logbooks* físicos de maneira mais eficiente. Além disso, apresentou ferramentas complementares da linguagem Python para validar e testar um *logbook* digital, considerando a necessidade de confiabilidade de um documento tão importante na manutenção aeronáutica.

Palavras-chave: *Logbook*. Motor. Banco de Dados. Django.

ABSTRACT

This work addresses the development of a digital logbook for the PW118 engines in the Brazilian Air Force (FAB) fleet. This approach is necessary due to the significant financial impact caused by the inefficient management of information regarding these engines and the lack of tools to ensure the reliability, control, and accessibility of this data. The goal of this work is to present a methodology proposal for the development of a digital logbook to replace the current physical logbooks of PW118 engines currently used by the FAB. This purpose will be achieved through bibliographical research on current regulations and documentary analysis of the data contained in

1 Pós-Graduando em logística pelo Instituto de Logística e Aeronáutica e bacharel em Engenharia Aeronáutica pelo Instituto Tecnológico de Aeronáutica. E-mail: gonzagagmgs@fab.mil.br.

2 Doutora em Administração pela Universidade Federal de Minas Gerais. Professora do Grupo Educacional UNIS. E-mail: leila.scanfone@professor.unis.edu.br.

3 Mestre em Engenharia Eletrônica e Computação pelo Instituto Tecnológico de Aeronáutica, especialista em Logística pelo Instituto de Logística de Aeronáutica e bacharel em Engenheira da Computação pelo Instituto Tecnológico de Aeronáutica. E-mail: luisalaa@fab.mil.br.

physical logbooks. Subsequently, documentary research was conducted on the physical files of PW118 engines, and a database was created using the Python language, utilizing the Django library. The study has revealed that the Django framework is a suitable tool for the development of the proposed methodology, capable of storing and managing the data contained in the physical logbooks more efficiently.
Keywords: Logbook. Engine. Database. Django.

1 INTRODUÇÃO

Os motores da série PW118, fabricados pela empresa canadense *Pratt & Whitney*, equipam as aeronaves C-97 Brasília (EMB-120), as quais vem desempenhando um papel significativo no cenário Nacional, cumprindo missões de transporte de carga e de passageiros da Força Aérea Brasileira (FAB), bem como de outros Órgãos Públicos, em todas as regiões do País. Em razão da demanda de missões para essas aeronaves (transporte de órgãos, ajuda humanitária e evacuação aero médica) essa frota tem voado, em média, 5.000 horas por ano conforme dados de utilização dos últimos anos, sendo 13 aeronaves ativas equipadas com 2 motores cada.

Nos últimos 10 anos, a assessoria técnica deste motor passou por três Parques de Material Aeronáutico diferentes, a saber, PAMA-AF, PAMA-SP e PAMA-LS, este último atualmente o parque central da unidade propulsora. Devido a essas múltiplas transferências, muito conhecimento e documentação técnica se perdeu. A título de exemplo, todos os motores, ao sair do fabricante, possuem um *logcard* com os principais dados que atestam a aeronavegabilidade do componente. Os motores da FAB, porém, não possuem mais esta documentação, de forma que todos os dados confiáveis do motor estão disponíveis em seus *logbooks* físicos, os quais só possuem validade dentro da própria Força Aérea.

Ainda, o sistema de dados digital atual, o Sistema Integrado de Logística de Material e de Serviços (SILOMS), não possui dados 100% confiáveis (SOUSA, 2021) para confirmar, ou até mesmo corrigir, os dados físicos. Além disso, alguns componentes importantes não estão incluídos no SILOMS devido às limitações do sistema, como é o caso dos itens rotativos, chamados de LLP - *LIFE LIMITED PARTS*.

Neste contexto, este trabalho aborda a proposta de metodologia para criação de *logbook* digital para implementação nos motores PW118 da frota da FAB, a fim de melhorar o gerenciamento dos grupos motopropulsores em relação à acessibilidade, segurança, confiabilidade e controle.

Tal abordagem se faz necessária devido ao grande impacto financeiro causado pela gestão deficiente da informação destes motores e a falta de uma ferramenta confiável, capaz de garantir a veracidade dos dados, controle efetivo e principalmente a acessibilidade destas informações por parte do parque central.

É importante ressaltar também que a contribuição deste trabalho poderá ser replicada para outros projetos, podendo trazer melhorias para toda a frota da FAB. Não somente em termos documentais, como na utilização do *logbook* digital como backup, mas também na qualidade da informação, a qual é a base para produção de melhorias e estudos em todas as áreas.

O objetivo deste trabalho é apresentar uma proposta de metodologia para criação de um *logbook* digital, a fim de substituir os atuais *logbooks* físicos dos motores PW118 utilizados atualmente pela FAB.

Este intento será conseguido mediante a revisão bibliográfica da tecnologia de banco de dados relacional e de bibliotecas da linguagem Python capazes de implementar o *logbook* na versão digital. Ainda, será feita uma análise documental dos *logbooks* físicos dos motores PW118 da FAB e programação dos campos do documento em Python, por meio da biblioteca Django, seguido de testes de validação. A escolha pela linguagem de programação Python se deve ao seu nível de estabilidade e suporte (ALESHI, SEKER, BABICEANU, 2019), fácil integração com praticamente todas as tecnologias de banco de dados atuais, além de diretrizes institucionais do PAMA-LS, o qual será abordado em maiores detalhes nas seções seguintes.

2 REVISÃO BIBLIOGRÁFICA

2.1 O CONCEITO DE *LOGBOOK* NA AVIAÇÃO

A legislação brasileira responsável por regulamentar a aviação civil é o Regulamento Brasileiro de Aviação Civil (RBAC) n° 91 (AGÊNCIA NACIONAL DE AVIAÇÃO CIVIL, 2021). Neste regulamento estão previstos diversos requisitos para garantir a aeronavegabilidade, entre eles os relacionados aos registros da aeronave. O artigo 91.417 do RBAC 91 prevê que devem ser mantidos permanentemente os registros de: tempo total de voo de cada célula, motor, hélice e rotor; a presente situação de partes com o tempo de vida limitado de cada célula, motor, hélice, rotor e equipamento; o tempo desde a última revisão geral de itens instalados na aeronave; o tempo desde a última inspeção obrigatória, requerida no programa de inspeções; a situação atualizada das diretrizes de aeronavegabilidade e de segurança; cópias de formulários de grandes alterações ou reparos.

Todos estes dados usualmente são registrados em um caderno físico chamado de "*logbook*". Este documento acompanha a aeronave ou motor em toda a sua operação e quando é recolhido para realizar inspeções e manutenções, preventivas ou corretivas. Sem o devido registro, mesmo que a aeronave esteja tecnicamente em condições de voo, ela não será aeronavegável. Em caso de perda, roubo, extravio ou qualquer outro dano que comprometa a verificação dos registros do *logbook*, deverá ser feito um levantamento histórico de todos os dados necessários, o que pode se mostrar um tanto quanto oneroso (ALESHI, SEKER, BABICEANU, 2019).

O regulamento brasileiro, ainda, não obriga a utilização de um *logbook* físico. No artigo 91.1025 item (q) está previsto, além de outros requisitos de segurança, “um sistema adequado (que pode incluir um sistema codificado ou eletrônico) que proporcione a preservação e a recuperação das informações” (AGÊNCIA NACIONAL DE AVIAÇÃO CIVIL, 2021, p. 86).

Diante deste cenário, não há um impedimento legal da utilização, inclusive da substituição, dos *logbooks* físicos por uma versão digital, desde que sejam garantidos a confiabilidade e segurança dos dados.

2.1.1 A utilização do *logbook* na FAB

No contexto da Força Aérea Brasileira, todas as aeronaves possuem o livro de registros da aeronave - LOGBOOK (LRA - LOG BOOK), conforme previsto no MCA 66-7 - Manual de Manutenção, Doutrina, Processos e Documentação de Manutenção (BRASIL, 2017), no formato físico. O referido manual afirma que “a qualidade será rastreável pelo registro de todos os fatos significativos na história de uma aeronave, de forma a garantir a sua aeronavegabilidade” (BRASIL, 2017, p. 25), não somente no registro físico como também por meio do SILOMS. Ou

seja, para a FAB a relevância de possuir o LRA está diretamente ligada à qualidade e rastreabilidade.

Da mesma forma que o RBAC 91, o MCA 66-7 não restringe que o LRA deva ser físico, apesar de haver diversas orientações sobre o seu preenchimento ao longo do documento. Ainda, é importante destacar que já existe um sistema na FAB utilizado para registros digitais de dados de manutenção aeronáutica, não focado nos dados do motor, que é o caso do SILOMS. Este sistema tem por objetivo manter os dados logísticos dos componentes aeronáuticos, porém existem diversas limitações em seu emprego (SOUSA, 2022).

2.2 BANCO DE DADOS RELACIONAL E A LINGUAGEM PYTHON

A tecnologia de banco de dados relacional foi idealizada principalmente por Edgar F. Codd, um cientista da computação britânico que trabalhava na IBM na década de 1970. Um banco de dados relacional é uma abordagem organizada e estruturada para armazenar dados, onde as informações são dispostas em tabelas, com relações entre elas definidas por chaves primárias e externas, facilitando a recuperação e a manipulação dos dados de forma eficiente e consistente (SILBERSCHATZ, KORTH e SUDARSHAN, 2009).

Cada coluna da tabela do banco de dados representa um atributo, ou um tipo de dado, e as linhas são os registros destes atributos, ou os dados a serem armazenados. Essa estrutura é amplamente utilizada em sistemas de gerenciamento de banco de dados em todo o mundo devido à sua eficácia na gestão de informações.

Durante a fase conceitual de um projeto de banco de dados é criado o diagrama de entidade e relacionamento (ERD – *Entity Relationship Diagram)*, o qual é fundamental para o sucesso do projeto (ELMASRI, 2015). Ele permite visualizar todas as interações entres as tabelas de dados, facilitando ainda a etapa da programação do código. Além disso, posteriormente será uma documentação muito importante para a manutenção e atualização do banco de dados e de suas funcionalidades (ELMASRI, 2015).

Existem atualmente diversas tecnologias que permitem o desenvolvimento de banco de dados, ou comumente chamado de *back-end*. Dentre as linguagens mais utilizadas podemos citar Python, Java, PHP, C++, GO, Kotlin, Ruby e JavaScript.

A linguagem Python é uma das mais utilizadas no mundo atualmente. Segundo o site RedMonk, um dos principais fornecedores de estatísticas para programadores, o Python está em segundo colocado no ranking de linguagens mais populares (O'GRADY, 2023). Além disso é uma linguagem dinâmica e aberta. Devido ao fato de sua arquitetura ser orientada a objeto, existem diversas bibliotecas que permitem expandir o uso da linguagem a áreas como *big data*, inteligência artificial, *machine learning*, e inclusive desenvolvimento *web*, por meio de *frameworks* como o Flask e Django (GHIMIRE, 2020).

No ano de 2022, o PAMALS iniciou uma estratégia de automatização e monitoramento de processos utilizando-se da linguagem Python (PAMA-LS, 2023), incluindo aulas de programação para os militares do seu efetivo.

Devido ao seu crescimento, maleabilidade, fácil aprendizado e possibilidade de integração com outras ferramentas, além do alinhamento estratégico no PAMA-LS, o Python foi a escolha para desenvolver este trabalho. Dentre os *frameworks* disponíveis nesta linguagem, optou-se pelo

Django, por possui melhor capacidade de lidar com projetos maiores e que necessitem de melhor organização, além de permitir melhor gerenciamento e manutenção da ferramenta durante a fase de utilização (GHIMIRE, 2020).

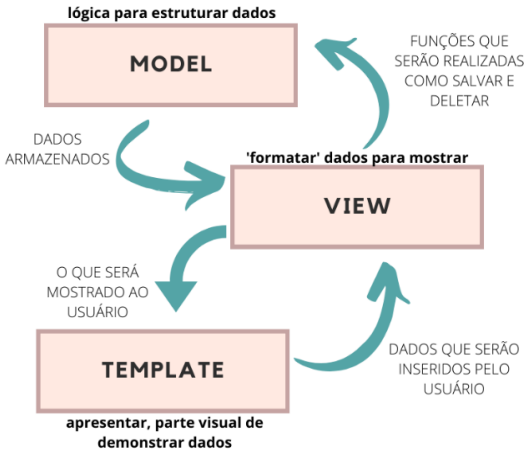
2.3 FRAMEWORK DJANGO

O framework Django tem por objetivo tornar mais rápida a fase de desenvolvimento de projetos Web. Ela fornece, dentro de sua biblioteca, soluções já prontas para uma grande quantidade de problemas inerentes a este tipo de trabalho, como por exemplo, autenticação de usuário, administração do banco de dados, mapeamento dos relacionamentos entre os objetos, além de soluções de segurança já implementadas (*DJANGO SOFTWARE FOUNDATION*, 2023?).

2.3.1 Arquitetura MTV

A ferramenta Django possui um padrão de arquitetura chamada *Model-Template-View* (MTV) que é uma variação do padrão *Model-View-Controller*, comumente utilizado em projetos de desenvolvimento *web*. A Figura 1 mostra um esquema do funcionamento desta arquitetura

Figura 1: Arquitetura MTV (*Model-Template-View*) do Django.



Fonte: (SILVA, 2020).

A camada *Model* da arquitetura MTV é responsável pela criação do banco de dados. Ao se criar um projeto Django, é criado o arquivo *models.py*. Neste arquivo são abstraídos as classes e os relacionamentos entre elas a fim de estruturar o banco de dados. Dessa forma, não é necessário realizar consultas complexas diretamente ao banco de dados, pois é possível acessá-lo por meio do próprio Python através do Django (SANTIAGO, 2020).

A segunda camada, *Template*, é responsável pelas informações que são acessíveis aos usuários. Ela é composta por arquivos HTML, CSS e JavaScript, nos quais estão descritos como os dados serão apresentados. O Django possui um site administrativo como solução *built-in* em seus *templates*, o qual permite acessar o banco de dados e gerenciar acessos, usuários e perfis (SANTIAGO, 2020).

Por último, a camada *View* é responsável pela interligação das camadas *Template* e *Model*, formatando os dados para que possam ser armazenados e transferindo-os entre camadas para que fiquem visíveis ao usuário (SANTIAGO, 2020).

2.3.2 Integração com banco de dados

O framework possui duas fases, uma de desenvolvimento e outra de produção. Na primeira é utilizado o SQLite como banco de dados padrão, o qual é gratuito. Durante esta fase, é criado o banco de dados e o site administrativo local, acessado apenas pelo computador do desenvolvedor, para visualização e testes, já sendo possível realizar o gerenciamento de usuários e criações de perfis específicos. Esses perfis, porém, utilizam o *UserModel*, que é a classe de usuário padrão do Django (*THE DJANGO PROJECT*, 2023?).

Na segunda fase, que é a de produção o sistema é preparado para utilização por diversos clientes em uma estrutura de servidor. E é possível alterar a estrutura de banco de dados utilizada. Através dos Models desenvolvidos no Django, pode-se escolher utilizar bancos de dados em SQLite (padrão), MariaDB, MySQL e Oracle (*THE DJANGO PROJECT*, 2023?).

2.3.3 Bibliotecas de testes

O framework Django e a linguagem Python possuem algumas ferramentas que auxiliam na realização de testes automatizados. O Python possui uma biblioteca de testes padrão chamada de "*unittest*", da qual o framework se utiliza para criar testes de aplicações da plataforma, por meio da subclasse '*TestCase*' (HIETANEN, 2019).

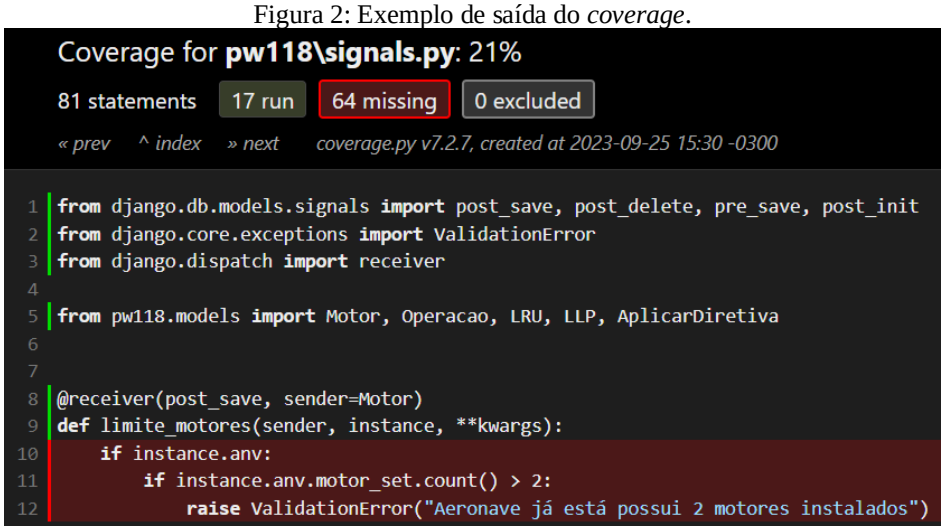
Entretanto, para realizar os testes é necessário criar objetos dos modelos desenvolvidos. Para tanto, existe ainda outra biblioteca do Python chamada *ModelMommy* (HIETANEN, 2019). Esta biblioteca cria dados aleatórios para o banco de dados, tornando os testes ainda mais robustos, uma vez que o tipo de *input* também é um parâmetro a ser testado.

2.3.4 Biblioteca coverage

Uma das funcionalidades internas do Django é a criação de testes. Por meio do arquivo *test.py* é possível programar testes automatizados. Como o objetivo deste trabalho é propor uma metodologia de criação de banco de dados, serão realizados somente testes funcionais, nos quais os *outputs* dos testes são comparados com os valores esperados da função específica (MARTINI, 2019)

A fim de verificar a extensão de código testado, é recomendado a utilização da biblioteca *coverage* (HORA, 2023). De acordo com a documentação da biblioteca, ela é utilizada para verificar a amplitude dos testes, emitindo um relatório que identifica quais partes do código foram testadas, e quais não foram (COVERAGE, 2023?). A Figura 2 mostra um exemplo de relatório de saída do *coverage*. Nela é possível identificar quais linhas de código ainda não estão cobertas por testes. No exemplo, elas aparecem destacadas em vermelho.

Figura 2: Exemplo de saída do *coverage*.



Fonte: O próprio autor (2023).

3 MATERIAL E MÉTODO

Conforme salientou-se na introdução, para atingir o objetivo deste trabalho de propor uma metodologia de criação de um *logbook* digital para os motores, foi realizada a revisão bibliográfica na seção 2, a qual será seguida de uma pesquisa documental do *logbook* dos motores PW118. A linguagem Python foi escolhida para o desenvolvimento do sistema, alinhando-se aos objetivos estratégicos do PAMA-LS, e dentro dela o *framework* Django foi utilizado para a modelagem dos dados. Dessa forma, este trabalho configura uma pesquisa bibliográfica e documental, quanto aos procedimentos aplicados. Além disso, trata-se de uma pesquisa qualitativa, quanto à natureza; aplicada, quanto à finalidade; e descritiva, quanto ao objetivo (SCANFONE e VASQUEZ, 2018).

No que tange à revisão bibliográfica, realizada na seção 2, foi possível verificar que não há regulamentação que obrigue que os registros dos motores sejam físicos. Observou-se ainda a oportunidade de utilização da linguagem Python e do *framework* Django para a aplicação na pesquisa em questão. Por fim, foi realizada uma pesquisa sobre bibliotecas auxiliares capazes de testar o código de programação e avaliar a cobertura dos testes realizados.

A pesquisa documental teve como escopo a análise dos dados do *logbook* dos motores PW118. Os motores escolhidos como alvo da análise, bem como suas características consideradas relevantes para compor o material aplicado na pesquisa e para construir o *logbook* digital, são apresentados na subseção 4.1.

Assim, os procedimentos utilizados neste trabalho foram:

- Pesquisa bibliográfica sobre *logbook* na aviação, banco de dados relacionais e bibliotecas da linguagem Python para implementação do *logbook* digital;
- Levantamento dos dados contidos no *logbook* físico, mediante análise documental;
- Criação do diagrama de modelagem do banco de dados, considerando quais dados serão inseridos na documentação digital;
- Desenvolvimento do *logbook* digital utilizando Python e Django; e
- Realização testes de funcionamento.

4 RESULTADOS

4.1 LEVANTAMENTO DOS DADOS CONTIDOS NO *LOGBOOK* DOS MOTORES PW118

Conforme foi explanado na subseção 2.1.1, todos os dados relevantes para manutenção dos motores devem estar presentes no *logbook*. Isso inclui o histórico de operação (em horas, pousos e ciclos), boletins e diretivas de aeronavegabilidade cumpridas, lista de itens instalados no motor. Quantos aos itens instalados no motor, há ainda a necessidade de se armazenar dados de horas ou ciclos desde a última manutenção e desde a instalação no motor em questão, registro das manutenções realizadas e de grandes serviços ou reparos.

No LRA físico atual estes dados estão dispostos da seguinte forma:

a) Ficha de custódia: esta ficha do *logbook* é um campo imprescindível para a correta contagem de horas de voo e pousos de um motor. Neste campo são preenchidas as datas da instalação e de remoção do motor em uma dada aeronave, constando também os dados de horas de voo e quantidade de pousos da aeronave e do motor no momento da instalação e remoção, e o operador da aeronave. Esta ficha é utilizada como última verificação da contagem de horas de voo e pousos de um motor, pois pode ser confrontado com a utilização da própria aeronave, que possui um *logbook* à parte;

b) Operação: o *logbook* físico do motor também contém o seu histórico de operação ao longo dos anos. O controle é feito mensalmente com o operador lançando a quantidade total de horas de voo e pousos do motor. Ao final do ano a ficha é finalizada, e então é contabilizado o total de horas e pousos do motor no período para iniciar a ficha do ano seguinte.

c) Inspeções: neste campo são lançadas as inspeções programadas e não-programadas realizadas no motor, com a descrição da inspeção e a data de cumprimento.

d) Grandes serviços: além das inspeções, o motor pode realizar grandes serviços como, por exemplo, um reparo da caixa de engrenagens, ou uma Revisão Geral. Os dados destes serviços são preenchidos de forma similar às inspeções, sendo boa prática arquivar junto ao *logbook* o relatório destas intervenções.

e) Diretivas técnicas: neste campo são lançadas as diretivas e boletins aplicados e não aplicados. Assim como as manutenções realizadas nos motores e seus itens, este campo é de grande relevância e que agregam valor aos respectivos objetos.

f) Histórico de estocagem: local onde é preenchido qual o regime de estocagem, local e data, para o caso de o motor ter sido retirado de uma aeronave para o armazém.

g) Movimentação de itens: de modo similar ao que é realizado com a ficha de custódia, neste campo são lançadas as movimentações de itens do motor que não possuem ficha histórico, como é o caso dos itens rotativos. Nela são preenchidos os dados de ciclos do motor e do item no momento de sua instalação e remoção, bem como é calculada a quantidade de ciclos restantes para a substituição do item.

h) Histórico de serviço de componentes e acessórios (HSCA): O Boletim de Itens Controlados dos Motores PW118, PW118A e PW118B, possui uma lista de itens que devem ser

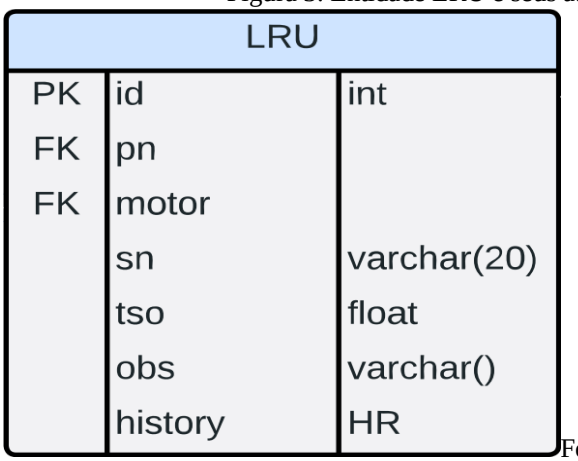
controlados com e sem ficha histórico. Em termos gerais, os itens controlados com ficha histórico são principalmente os acessórios, ou LRUs (*Line Replacement Units*), além do motor, a caixa de redução (RGB – *Reduction Gear Box*) e a turbo-máquina (TMM – *Turbo Machine*). Os LRUs possuem vencimentos por hora de voo, e sua ficha histórico contém os dados descritivos, dados de horas do componente e do motor no momento de instalação e de remoção e operador que realizou a atividade (PAMA-SP, 2020a).

Todos estes campos (características do motor e de seus componentes) foram escolhidos para serem inseridos no banco de dados, compondo o *logbook* digital desenvolvido com o framework Django. Além disso foram criadas diversas funcionalidades com a programação em Python, a fim de operacionalizar a relação entre os campos de forma lógica e representativa da realidade, fazendo com que o framework escolhido durante a pesquisa bibliográfica se tornasse aplicável ao estudo de caso em questão.

4.2 DIAGRAMA DE MODELAGEM

Partindo do conceito de banco de dados relacional, foi criado o diagrama de relacionamento (ERD – *Entity Relationship Diagram*). Primeiramente foram criadas as entidades, cujos elementos serão os objetos a serem gravados no banco de dados, como por exemplo um motor ou um item acessório. Cada entidade possui características, também chamadas de atributos. A Figura 3 exemplifica a entidade LRU (*Line Replacement Unit*) e seus atributos.

Figura 3: Entidade LRU e seus atributos.



Fonte: O próprio autor (2023).

Conforme pode-se notar na figura, entre os atributos do LRU estão seu número serial, o pn associado e outras informações como TSO (*Time Since Overrall* – tempo desde a última revisão), sendo que cada informação possui um tipo (*int*, *float*, *varchar*, entre outros). O Apêndice A mostra o diagrama com todas as entidades criadas e os relacionamentos entre elas.

Como o objetivo do trabalho é criar um *logbook* digital, foram escolhidas para armazenar no banco de dados todas as informações contidas no *logbook* físico considerado no estudo de caso (conforme subseção 3.1). Nos tópicos a seguir serão discutidos como cada campo do *logbook* foi tratado na versão digital.

4.2.1 Ficha de custódia e Estocagem

Na versão digital foram criadas as entidades Aeronave, Motor e HistoricalMotor, que correspondem à ficha de custódia do *logbook*. As duas primeiras possuem os dados atuais de horas de voo, pousos da aeronave e ciclos para o motor. A última tabela armazena o histórico de qualquer mudança ocorrida em qualquer campo da entidade Motor, inclusive quando a este é atribuído uma aeronave. Este processo exclui a necessidade de que o usuário informe os dados da aeronave e do motor no momento da instalação, nem mesmo a data, já que esses dados serão armazenados automaticamente no histórico com o dado atual de cada objeto.

Em substituição ao campo estocagem, na entidade Motor há o parâmetro observação, no qual, para que o usuário lance como foi realizada a estocagem. E a entidade HistoricalMotor incumbe-se de salvar o histórico desse lançamento.

4.2.2 Operação

Este dado de operação é um dos mais críticos, pois pode apresentar erros de cálculo e de preenchimento. Ainda, como na FAB é comum ocorrer a transferência de aeronaves entre operadores, o preenchimento por vezes é incompleto ou até inexistente em alguns casos. Além das transferências de aeronaves, há também a permuta dos próprios motores entre os operadores, agravando ainda mais os erros acima.

Para tornar esse dado confiável, foi criada a entidade “Operacao”, em um relacionamento *many-to-one* com a entidade “Aeronave”. Ou seja, cada operação só estará relacionada a uma, e somente uma, matrícula, ao passo que uma matrícula poderá ter várias operações. Cada vez que o usuário cria uma operação, os dados da aeronave e de todos os componentes instalados nela são atualizados, o que será apresentado com maiores detalhes na seção 4.3. Não obstante, foi adicionado o campo ‘partidas’, que é o número de partidas de motor em uma dada operação.

O controle por partidas, também chamadas de ciclos, de motor é o sugerido pelo manual do fabricante, porém a FAB não realiza esse tipo controle por limitações do SILOMS. Como substituição, a FAB considera cada ciclo como um pouso de aeronave. Com isso, a quantidade de ciclos ao longo do tempo se torna maior do que a contada a partir do número de partidas com decolagem. Isso porque existem ocasiões em que a aeronave pousa e decola sem a necessidade de desligar o motor.

4.2.3 Ficha histórico de acessórios

Os itens acessórios são controlados por seriais. Por isso, foram criadas duas entidades: PnLRU e LRU. A primeira cria um PN (*Part Number)*, que é um modelo do LRU. A segunda, LRU, atribui um SN (*Serial Number*) e as quantidades de horas para este serial. Assim como foi feito para os motores, também foi criada uma entidade ‘HistoricalLRU’ a qual armazena todo o histórico de modificações em qualquer atributo para um determinado objeto LRU.

4.2.4 Instalação e remoção de rotativos

Os itens rotativos do motor, também chamados de LLP (*Life Limited Parts*), são compostos pelas turbinas, compressores, estatoras e seus itens menores e, como o nome sugere, são itens com tempo de vida limitado. Conforme o Boletim de Itens Controlados, eles também são monitorados, porém sem ficha histórico, uma vez que raramente um desses componentes é removido antes de seu vencimento. O principal dado que é armazenado no *logbook* físico é quando o item foi instalado, quantos ciclos ele possuía na instalação, e quantos ciclos o motor possuía na instalação.

Apesar de não precisar de ficha histórico, estes itens são difíceis de se controlar sistemicamente. Prova disso é que os rotativos não estão incluídos na configuração real da aeronave no SILOMS, ao contrário dos itens acessórios, ou seja, não são controlados via sistema. Isso ocorre porque o vencimento dos LLPs não depende somente do PN, ou modelo, do item em si, mas também do modelo do motor no qual ele está sendo instalado. Por exemplo, na Figura 4 vemos o item "IMPELLER, LP", que possui um fator 1 para o modelo de motor PW118, e um fator 1.38 para os modelos 'A' e 'B'.

Figura 4 - Tabela de ciclos acumulados do Impeller, LP.

DESCRIPTION	DETAIL PART NO.	ENGINE MODEL	ABB'D CYCLE FACTOR	FLIGHT COUNT FACTOR	ACCUMULATED TOTAL CYCLE LIMIT
Impeller, LP	3033524	**PW118**	10	1.0	25000
	3033524	**PW118A, PW118B**	1	1.38* 1.0**	25000
	3035569	**PW118**	10	1.0	25000
	3035569	**PW118A, PW118B**	1	1.38* 1.0**	25000
	3040689	**PW118**	10	1.0	25000
	3040689	**PW118A, PW118B**	1	1.38* 1.0**	25000

Fonte: PWC (2018).

Para resolver este problema foram criadas as entidades FCF e PnLLP, além da entidade Modelo. Esta contém os três modelos de motores utilizados na frota, PW118, PW118A e PW118B. Ao criar um serial de motor necessariamente deverá ser atribuído um modelo a este objeto. A entidade PnLLP é semelhante à PnLRU, especificando um PN de LLP. Como foi explicado, este PN terá um fator específico dependendo do modelo de motor no qual estiver instalado. Assim, a entidade FCF especifica qual o fator dado o PN do item rotativo e o modelo de motor.

Do mesmo modo feito para os acessórios, também foi criada uma entidade LLP, que cria um serial para um PN já criado de PnLLP. No entanto, ao invés de ser controlado por horas de voo, os rotativos são controlados por ciclos de motor, e não realizam manutenções preventivas (desconsiderando os grandes reparos feitos no motor, como HSI e *Overhaul*, que são considerados como manutenções do motor).

4.2.5 Diretivas e Manutenções Programadas

Nos *logbooks* físicos esses dados são amplamente negligenciados e raramente preenchidos. Os dados contidos no SILOMS são os mais utilizados, porém sem muita confiabilidade, devido às limitações do próprio sistema.

Neste trabalho foram incluídas todas as diretivas previstas pelo fabricante e as manutenções contidas no Plano de Manutenção dos Motores PW118, PW118A e PW118B. As entidades Diretivas, MntPreventiva e OverhaulLRU armazenam os dados de todas as diretivas dos motores, manutenções programadas de motor e de LRUs, respectivamente. Foram criadas também as entidades AplicarDiretiva, RealizarManutencaoMotor e RealizarOverhaulLRU, para controlar a realização das diretivas e manutenções (PAMA-SP, 2020b).

4.3 REGRAS DE NEGÓCIO

Os relacionamentos padrão entre os *models* do Django precisam de um tratamento detalhado para representar a realidade. Por exemplo, a relação *one-to-many* entre aeronave e motor permite que sejam instalados mais de dois motores em uma mesma aeronave. Para estes casos foram criadas regras de negócio que garantem o funcionamento de acordo com o observado na realidade.

A Tabela 1 a seguir mostra o nome das funções criadas e o objetivo na implementação de cada uma.

Tabela 1: Descrição das funções desenvolvidas.

Função	Objetivo	Entidade	Signal
limite_motores	Limita a 2 motores por aeronave	Motor	post_save
atualiza_itens	Atualiza os parâmetros dos itens instalados em motor, quando há uma operação.	-	-
limite_lru	Não permite que mais de um LRU seja instalado na mesma posição em um motor.	LRU	post_save
limite_llp	Não permite que mais de um LLP seja instalado na mesma posição em um motor.	LLP	post_save
atualiza_tsn	Atualiza os parâmetros de motor e aeronave após o lançamento de uma operação.	Operacao	post_save
atualiza_tsn_delete	Atualiza os parâmetros de motor e aeronave quando uma operação salva é deletada.	Operacao	post_delete
verifica_diretiva	Verifica a efetividade de uma diretiva, bem como se já foi aplicada.	AplicarDiretiva	-

Fonte: O próprio autor (2023).

A função ‘limite_motores’ garante que não haverá mais de dois motores instalados na mesma aeronave. Ela também retorna um erro de validação (*ValidationError*) caso o usuário tente adicionar um motor em uma aeronave com dois motores instalados.

No caso dos LLP e LRU, temos uma relação de one-to-many entre estes itens e os motores. Ou seja, um motor pode ter vários LLP, por exemplo, mas um LLP só pode estar instalado em um motor. O problema é que existem diferentes posições de LLP: turbina de alta, turbina de baixa, compressor de alta, etc. Por isso, foi criada uma entidade PosicaoMotor, com todas as posições de instalação, tanto para LLP quanto para LRU. Com isso evitou-se criar uma entidade para cada tipo de LLP e LRU, diminuindo espaço de armazenamento e a complexidade do banco de dados.

Utilizando-se dos dados de PosicaoMotor, as funções ‘limite_lru’ e ‘limite_llp’ verificam se a posição na qual o item está sendo instalado está ocupada, se estiver retorna um erro de validação.

Ainda, para garantir que os todos os dados estejam atualizados, foram criadas as funções ‘atualiza_tsn’ e ‘atualiza_tsn_delete’. Quando uma operação é salva, ou deletada, do banco de dados, as respectivas funções atualizam todos os campos de horas de voo, partidas e pousos de todos os itens na árvore da aeronave para a qual foi criada, ou apagada, a operação. Ainda, quando da inclusão de uma operação, caso a aeronave não possua dois motores instalados (sistemicamente) a função retornará um erro de validação e não salvará esta operação no banco de dados. Isto foi feito para evitar erros de controle, uma vez que os dados de horas, pousos e partidas serão alterados exclusivamente por meio desta operação, no caso usual.

Por último, a função ‘verifica_diretiva’ examina, por meio da entidade Efetividade, se a diretiva é aplicável ao serial para o qual está sendo atribuída. Ainda, caso a diretiva já tenha sido aplicada ao motor, a função retorna um erro de validação.

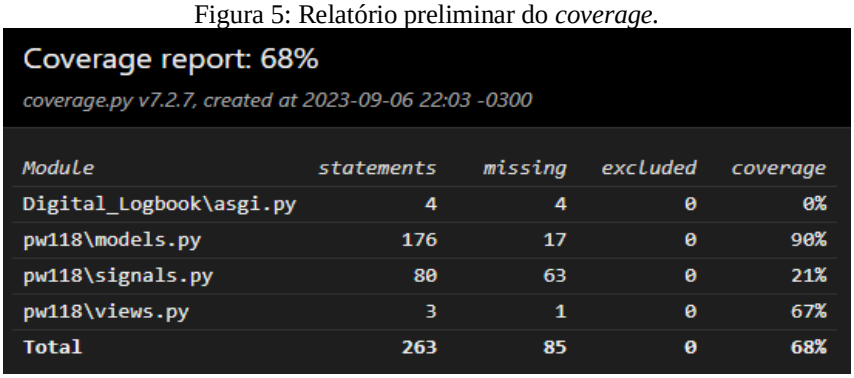
5 VALIDAÇÃO E TESTES

Após a implementação, é fundamental verificar se as funcionalidades foram implementadas de maneira correta. Neste capítulo serão descritos os testes realizados, bem como os seus resultados.

5.1 CRIAÇÃO DOS TESTES

Inicialmente, com o objetivo de se verificar quais testes deveriam ser criados, foi gerado o relatório por meio do *coverage* a fim de verificar quais partes ainda necessitavam ser validadas, uma vez que o próprio Django já possui ferramentas de teste. O resultado geral do *coverage* pode ser visto na Figura 5.

Figura 5: Relatório preliminar do *coverage*.



Coverage report: 68%

coverage.py v7.2.7, created at 2023-09-06 22:03 -0300

Module	statements	missing	excluded	coverage
Digital_Logbook\asgi.py	4	4	0	0%
pw118\models.py	176	17	0	90%
pw118\signals.py	80	63	0	21%
pw118\views.py	3	1	0	67%
Total	263	85	0	68%

Fonte: O próprio autor (2023).

Observa-se que já existe uma porcentagem do código testada. Isso porque o Django já possui testes padrão, principalmente para os *models.py*, que são as entidades do banco de dados. Sendo assim, para este módulo basta realizar o teste da função que retorna o nome do objeto.

O foco principal de testes está nas funcionalidades, desenvolvidas no módulo *signals.py*. Para este módulo foram criados os testes, a fim de garantir o correto funcionamento da lógica da aplicação. Estes testes serão discutidos nas seções seguintes.

5.1.1 Teste 1 – Limite de motores por aeronave

O padrão de dados relacional do Django não é suficiente em termos de limitação de relacionamentos. Não é possível estabelecer diretamente, por meio de uma chave externa, um relacionamento dois-para-um, como é o caso dos motores e aeronave, no qual cada aeronave pode ter dois motores instalados, no máximo.

Por isso foi feito um filtro para quando um dado de motor for salvo no banco de dados. Basicamente é verificado o número de motores já instalados na aeronave para a qual se está atribuindo o motor. Caso esse número seja dois, a aplicação retornará um erro de validação.

A fim de testar essa funcionalidade foi desenvolvido o código mostrado na Figura 6:

Figura 6: Código de teste da função limite_motores.

```python
class LimiteMotoresTestCase(TestCase):

    def setUp(self) -> None:
        self.aeronave = mommy.make('Aeronave')
        self.motores = mommy.make('Motor', _quantity=3)

    def test_limite_motores(self):
        installed = 0
        for motor in self.motores:
            motor.anv = self.aeronave
            try:
                motor.save()
                installed += 1
            except ValidationError:
                self.assertEquals(installed, 2)
```

Fonte: O próprio autor (2023)

No teste, são criados 4 objetos aleatórios: um objeto Aeronave e 3 objetos Motor. A função *test_limite_motores* atribui à aeronave os motores, um a um. No momento de adicionar o terceiro motor, a função original *limite_motores* deve retornar um erro de validação, e o número de motores atribuídos na aeronave deve ser igual a dois.

5.1.2 Teste 2 – Limite de LRU e LLP por motor

Da mesma forma que existe a limitação de motores por aeronave, também há um limite de itens a serem instalados no motor, tanto LRU quanto LLP. Como o tratamento é análogo, será utilizado para fins de explanação os itens LRU.

Foi visto nas seções 4.1.3 e 4.1.4 que foram criadas duas entidades para os itens LRU: PnLRU e LRU. Entretanto, para controlar a quantidade de itens instalados no motor faz-se necessário a criação da entidade PosicaoMotor. Esta classe é composta pelas posições disponíveis de instalação do motor, e é fixa, ou seja, o usuário não terá acesso, nem poderá criar, alterar ou deletar qualquer objeto dessa classe, apenas atribuir uma das posições a algum PN criado, obrigatoriamente.

Cada motor poderá ter vários itens LRU, ou LLP, instalados, e cada um destes itens só poderá estar instalado em um motor. Este caso já está incluso no relacionamento entre as entidades LRU, LLP e Motor. Mas um motor só poderá ter um item instalado por posição, por isso foi desenvolvida a função limite_LRU, limitando a quantidade de itens instalado por posição do motor.

Para verificar o seu funcionamento foi aplicado o teste conforme o código da Figura 7.

Figura 7: Código de teste da função limite_lru.

```
class LimiteLruTestCase(TestCase):

    def setUp(self) -> None:
        self.motor = mommy.make('Motor')
        self.pos = mommy.make('PosicaoMotor', _quantity=9)
        self.pnlru = mommy.make('PnLRU', _quantity=20)
        self.lrus = mommy.make('LRU', _quantity=100)


    def test_limite_lru(self):
        result = {}
        for pn in self.pnlru:
            pn.pos = self.pos[randint(0, 8)]
            pn.save()
            result[str(pn.pos)] = 0
        for lru in self.lrus:
            lru.pn = self.pnlru[randint(0, 19)]
            try:
                lru.motor = self.motor
                lru.save()
                result[str(lru.pn.pos)] += 1
            except ValidationError:
                self.assertEquals(result[str(lru.pn.pos)], 1)
        for pos in result:
            self.assertLessEqual(result[pos], 1)
```

Fonte: O próprio autor (2023).

Primeiramente, foram criados os seguintes objetos: 1 Motor, 9 Posições, 20 *Part Numbers* e 100 Seriais, aleatórios. Em seguida, é atribuído para cada PN, aleatoriamente, uma das posições de motor criadas, e para cada serial um dos PNs criados, também de forma aleatória. Por fim, é feito um loop para atribuir os seriais ao motor. Cada vez que um item é atribuído a uma posição já preenchida é retornado um erro de validação. E ao final, todas as posições que tiveram um PN atribuído devem estar preenchidas.

5.1.3 Teste 3 – Atualização de operação

Uma das funcionalidades mais importantes a ser implementada é a contagem dos parâmetros de controle de itens, por meio do lançamento de dados de operação da aeronave. Quando algum dado de operação de uma aeronave é salvo no banco de dados, deve ser adicionado também a mesma operação para os motores e seus respectivos itens, de acordo com os parâmetros e fatores de ciclos específicos.

Com o objetivo de garantir esta funcionalidade de controle, foi desenvolvido um teste que verifica todos os lançamentos de operação. Como se trata de vários parâmetros a serem atualizados, este teste foi dividido em blocos, a fim de facilitar a detecção de falhas nos códigos. Como nos testes anteriores, foram criados objetos aleatórios conforme mostra a Figura 8.

Figura 8: Configuração do teste de operação.

```
class OperacaoTestCase(TestCase):

    def setUp(self) -> None:
        self.anv = mommy.make('Aeronave')
        self.anv2 = mommy.make('Aeronave')
        self.modelo = mommy.make('Modelo')
        self.motores = mommy.make('Motor',
                                  anv=self.anv,
                                  modelo=self.modelo,
                                  _quantity=2)
```

Fonte: O próprio autor (2023).

O primeiro teste verifica se os dados de aeronave e de seus motores foi atualizada corretamente. A Figura 9 mostra o código utilizado para esta verificação.

Figura 9: Código teste da função atualiza_tsn.

```
def test_atualiza_tsn(self):
    for index in range(10):
        tsn_anv = self.anv.tsn
        pousos_anv = self.anv.pousos
        tsn_mot1 = self.motores[0].tsn
        tso_mot1 = self.motores[0].tso
        csn_mot1 = self.motores[0].csn
        tsn_mot2 = self.motores[1].tsn
        tso_mot2 = self.motores[1].tso
        csn_mot2 = self.motores[1].csn
        op = mommy.make('Operacao', matricula=self.anv)
        hv = op.hv
        pousos = op.pousos
        partidas = op.partidas
        self.motores[0].refresh_from_db()
        self.motores[1].refresh_from_db()
        self.assertEquals(tsn_anv+hv, self.anv.tsn)
        self.assertEquals(tsn_mot2+hv, self.motores[1].tsn)
        self.assertEquals(tso_mot1+hv, self.motores[0].tso)
        self.assertEquals(tso_mot2+hv, self.motores[1].tso)
        self.assertEquals(pousos_anv+pousos, self.anv.pousos)
        self.assertEquals(csn_mot1+partidas, self.motores[0].csn)
        self.assertEquals(csn_mot2+partidas, self.motores[1].csn)
```

Fonte: O próprio autor (2023).

Acompanhando o código, foram simulados o lançamento de 10 operações, criadas de forma aleatória. Os dados da aeronave e dos motores são então comparados, antes e depois do lançamento, sendo que o valor de cada parâmetro deve ser o valor do parâmetro anterior mais o acréscimo da operação.

O segundo teste, mostrado na Figura 10, é mais simples e somente verifica se uma condição de restrição, a qual não permite o lançamento de operação para uma aeronave que não esteja com 2 motores instalados no sistema.

Figura 10: Código teste do limite de motores ao atribuir uma operação à uma aeronave.

```
    def test_motores_instalados(self):
        with self.assertRaises(ValidationError):
            op = mommy.make('Operacao', matricula=self.anv2)
```

Fonte: O próprio autor (2023).

O terceiro e último teste de operação verifica a atualização dos parâmetros dos itens instalados nos motores aplicados à aeronave em voo. São criados então, aleatoriamente, um item do tipo LRU e um do tipo LLP. Em seguida, é feito o mesmo procedimento do primeiro teste: uma operação é simulada e os parâmetros dos itens são comparados, de modo a verificar o acréscimo da operação. A Figura 11 mostra o código utilizado neste teste.

Figura 11: Código para teste da função atualiza_itens

```
def test_atualiza_itens(self):
    pos_llp = mommy.make('PosicaoMotor')
    pn_llp = mommy.make('PnLLP', pos=pos_llp)
    fator = mommy.make('FCF', modelo=self.modelo, llp=pn_llp)
    llp = mommy.make('LLP', pn=pn_llp, motor=self.motores[0])
    pos_lru = mommy.make('PosicaoMotor')
    pn_lru = mommy.make('PnLRU', pos=pos_lru)
    lru = mommy.make('LRU', pn=pn_lru, motor=self.motores[1])
    for index in range(10):
        tso_lru = lru.tso
        acc_llp = llp.acc
        op = mommy.make('Operacao', matricula=self.anv)
        hv = op.hv
        partidas = op.partidas
        lru.refresh_from_db()
        llp.refresh_from_db()
        self.assertEquals(tso_lru + hv, lru.tso)
        self.assertEquals(acc_llp + partidas*fator.fator, llp.acc)
        op.delete()
        lru.refresh_from_db()
        llp.refresh_from_db()
        self.assertAlmostEqual(tso_lru, lru.tso, delta=0.001)
        self.assertAlmostEqual(acc_llp, llp.acc, delta=0.001)
```

Fonte: O próprio autor (2023).

5.2 RESULTADOS DOS TESTES

Após a programação dos testes, o *coverage* é utilizado novamente para efetivamente realizar os códigos e emitir o relatório final. A Figura 12 mostra o resultado dos testes. Cada ponto significa um teste realizado com êxito, evidenciando que todas as funcionalidades implementadas estão corretas.

Figura 12: Log dos resultados dos testes aplicados.

```
(venv) C:\Users\guigs\OneDrive\PAMALS\208. CESLOG\TCC\Projeto_Django>coverage run manage.py test
Found 15 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
...............
----------------------------------------------------------------------
Ran 15 tests in 15.957s

OK
Destroying test database for alias 'default'...
```

Fonte: O próprio autor (2023).

Para confirmar que todo o código foi testado, foi emitido o relatório do *coverage*. O resultado é apresentado na Figura 13. Todos os módulos desenvolvidos da aplicação, a saber *models.py* e *signals.py* aparecem com 100% do código testado.

Figura 13: Relatório do *coverage* após realização dos testes.

Coverage report: 98%

coverage.py v7.2.7, created at 2023-09-06 22:47 -0300

Module	*statements*	*missing*	*excluded*	*coverage*
Digital_Logbook\asgi.py	4	4	0	0%
pw118\models.py	160	0	0	100%
pw118\signals.py	80	0	0	100%
pw118\views.py	3	1	0	67%
Total	**247**	**5**	**0**	**98%**

Fonte: O próprio autor (2023).

5.3 INTERFACE COM O USUÁRIO

O desenvolvimento tratado neste trabalho é prioritariamente *back-end*, ou seja, está preocupado principalmente com a criação e funcionamento do banco de dados, de maneira segura e confiável. Todavia, a interface com o usuário, ou *front-end*, faz parte de qualquer projeto web e é necessário para que seja possível acessar o banco de dados de maneira rápida e fácil.

Uma das vantagens deste trabalho ter sido desenvolvido no Django é que este possui, por padrão, um site administrativo de *front-end* que permite a manipulação de dados por usuários cadastrados. Além disso, por se tratar de um *framework*, é possível desenvolver uma aplicação *front-end* integrada ao banco de dados no mesmo projeto.

Além da possibilidade de desenvolver uma interface personalizada, o banco de dados elaborado neste trabalho é integrável com o SILOMS, que é também uma ferramenta *front-end.*

6 CONSIDERAÇÕES FINAIS

Neste momento, é oportuno retomar o objetivo deste trabalho, o qual foi propor uma metodologia de desenvolvimento de *logbook* digital para os motores PW118 da FAB, a fim de substituir os *logbooks* físicos. Para esse propósito, foi realizada pesquisa bibliográfica acerca do *logbook* na aviação, da tecnologia de banco de dados relacional e de bibliotecas da linguagem Python capazes de implementar o *logbook* na versão digital. Destacou-se o *framework* Django como uma ferramenta de desenvolvimento de dados e aplicação *web*, a qual unifica na mesma plataforma os desenvolvimentos *back-end* e *front-end*, além de interagir com bibliotecas de teste e cobertura. Por conseguinte, após pesquisa documental nos *logbooks* dos motores PW118, realizou-se a modelagem dos dados no framework Django.

Verificou-se que o modelo criado em Django, utilizando a linguagem Python, é capaz de armazenar todos os dados contidos no logbook físico de forma eficiente e segura, e com uma melhor acessibilidade, uma vez que se trata de uma aplicação web. Além disso, por meio das funcionalidades criadas foi possível melhorar a inclusão de dados, diminuindo o número de campos necessários a serem preenchidos e integrando dados de operação e parâmetros de controle de itens, bem como criando alertas que impedem preenchimentos incorretos.

Além disso, o armazenamento digital dos dados traz facilita a possibilidade de estudos futuros e *insights* acerca dos motores e dos itens da frota, como controle de giro, estudos de confiabilidade e monitoramento de performance, em virtude da padronização dos dados e da possibilidade de se armazenar a informação com segurança por um longo período.

Este trabalho requer um maior aprofundamento no que diz respeito à interação com o usuário e o desenvolvimento de *front-end*, uma vez que o foco foi a modelagem *back-end* do *logbook*. Além disso, o modelo requer ainda uma validação no que diz respeito à utilização real, ou seja, a inclusão e acompanhamento de dados reais de motores para o levantamento de possíveis discrepâncias.

Como sugestão para trabalhos futuros, podem ser exploradas outras ferramentas de banco de dados que porventura possam ser aplicadas ao objeto em questão, como por exemplo tecnologias como *blockchain*, ou a utilização de outras linguagens como *Ruby* e *JavaScript*.

REFERÊNCIAS

ALESHI, Ahrash; SEKER, Remzi; BABICEANU, Radu F. Blockchain model for enhancing aircraft maintenance records security. **2019 IEEE International Symposium on Technologies for Homeland Security (HST)**. Woburn, MA, 2019, pág. 1-7, DOI: 10.1109/HST47167.2019.9032943.

AGÊNCIA NACIONAL DE AVIAÇÃO CIVIL- **Regulamento brasileiro da aviação civil (RBAC) nº 91**. Brasília, DF, 12 de fev. 2021. Disponível em: https://www.anac.gov.br/assuntos/legislacao/legislacao-1/rbha-e-rbac/rbac/rbac-91/@@display-file/arquivo_norma/RBAC91EMD01.pdf. Acesso em: 17 de maio de 2023.

BRASIL. Ministério da Defesa. Comando da Aeronáutica. **MCA 66-7: Manual de Manutenção Doutrina, Processos e Documentação de Manutenção.** Comando da Aeronáutica. Estado Maior da Aeronáutica. Brasília, DF, 24 de abril de 2017.

COVERAGE. **Coverage.py 7.3.1 Documentation**. [S.I.], [2023?]. Disponível em: https://coverage.readthedocs.io/en/7.3.1/. Acesso em: 29 de set de 2023.

DJANGO SOFTWARE FOUNDATION. **Django Overview**. [S.I.], [2023?]. Disponível em: https://www.djangoproject.com/start/overview/. Acesso em: 29 de set. de 2023.

ELMASRI, Ramez; NAVATHE, Shamkant B. **Fundamentals of database systems**. 7ª ed. Pearson, 2015.

GHIMIRE, Devndra. **Comparative study on Python web frameworks: Flask and Django**. Trabalho de Conclusão de Curso (Bacharelado em Engenharia) - Metropolia University of Applied Sciences, Helsínquia, Finlândia, 2020.

HIETANEN, Veli-ville. **CI report tracking solution**. Häme university of applied sciences, Information and Communication Technology. Finlândia, 2019. Disponível em: https://www.theseus.fi/handle/10024/161640. Acesso em: 15 de out. de 2023.

HORA, Andre. Excluding code from test coverage: practices, motivations, and impact. **Empirical Software Engineering**, [S.I.], v. 28, n. 1, p. 16, 2023.

MARTINI, Joana. **Using Django and JavaScript to implement Model-Based Performance Testing as a Service with the MBPeT tool**. Tese (mestrado) - Åbo Akademi University, Finlândia, 2019.

O'GRADY, Stephen. **Language Rankings: 1-23**. Maine, EUA, 2023. Disponível em: https://redmonk.com/sograd y/2023/05/16/language-rankings-1-23/. Acesso em: 29 de set. de 2023.

PAMA-LS – Parque de Material Aeronáutico de Lagoa Santa. **Capacitação em inglês e linguagem de programação**. Lagoa Santa, MG, 2023. Disponível em: http://www.pamals.intraer/index.php/2014-12-11-17-51-57/2185-capacitacao-em-ingles-e-linguagem-de-programacao. Acesso em: 20 de out. de 2023.

PAMA-SP – Parque de Material Aeronáutico de São Paulo. **BT SP20 655 C-97 001 Itens Controlados dos Motores PW118, PW118A e PW118B.** São Paulo, SP, 2020a.

PAMA-SP – Parque de Material Aeronáutico de São Paulo. **BT SP20 665 C-97 002 Plano de Manutenção dos Motores PW118, PW118A e PW118B.** São Paulo, SP, 2020b.

PWC – Pratt & Whitney Canada. **Maintenance Manual No. 3034622**. Rev No. 61.1. 2018

SANTIAGO, Cynthia Pinheiro et al. Desenvolvimento de sistemas Web orientado a reuso com Python, Django e Bootstrap. **Sociedade Brasileira de Computação.** Porto Alegre, RS, 2020. Disponível em: https://sol.sbc.org.br/livros/index.php/sbc/catalog/download/48/219/457-1?inline=1. Acesso em: 19 de out. de 2023.

SCANFONE, Leila; VASQUES, Letícia. **Guia de Estudos da disciplina Metodologia da Pesquisa.** Centro Universitário do Sul de Minas - UNIS. Varginha, MG, 2018

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN. **Database System Concepts**. 6ª ed. Nova Iorque: McGraw-Hill, 2009.

SILVA, Diandra. **Como Funciona a Arquitetura MTV do Django**. 2020. Disponível em: https://diandrasilva.medium.com/como-funciona-a-arquitetura-mtv-django-86af916f1f63. Acesso em: 29 de set. de 2023

SOUSA, Jefferson Rodrigues. **Avaliação da Alteração do programa de manutenção das aeronaves A-29 da Força Aérea Brasileira.** Trabalho de Conclusão de Curso – Curso de Especialização em Logística, Guarulhos, 2022.

THE DJANGO PROJECT**. Django 4.2 Documentation - Databases**. [S.I.], [2023?]. Disponível em: https://docs.djangoproject.com/en/4.2/ref/databases/. Acesso em: 29 de set de 2023.