Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Instituto de Matemática e Estatística

Marcio Nogueira Pereira da Silva

Estudo e Implementação de um Sistema Customizável para Controle Laboratorial para o Processo de Tipificação HLA

Rio de Janeiro
2019

Marcio Nogueira Pereira da Silva

Estudo e Implementação de um Sistema Customizável para Controle Laboratorial para o Processo de Tipificação HLA

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Ciências Computacionais, da Universidade do Estado do Rio de Janeiro.

Orientadores: Prof. Dr. Alexandre da Costa Sena
Prof. Dr. Leandro Augusto Justen Marzulo

Rio de Janeiro
2019

CATALOGAÇÃO NA FONTE

UERJ/REDE SIRIUS/BIBLIOTECA CTC/A

S586 Silva, Márcio Nogueira Pereira da.
Estudo e implementação de um sistema customizável para controle laboratorial para o processo de tipificação HLA/ Márcio Nogueira Pereira da Silva. – 2019.
109 f.: il.

Orientadores: Alexandre da Costa Sena, Leandro Augusto Justen Marzulo
Dissertação (Mestrado em Ciências Computacionais) - Universidade do Estado do Rio de Janeiro, Instituto de Matemática e Estatística.

1. Banco de dados - Teses. 2. Sistemas de informação em saúde - Teses. 2. Antígenos histocompatibilidade HLA - Banco de dados - Teses.. I. Sena, Alexandre da Costa. II. Marzulo, Leandro Augusto Justen. III. Universidade do Estado do Rio de Janeiro. Instituto de Matemática e Estatística. IV.Título.

CDU 004.6

Patricia Bello Meijinhos CRB7/5217 - Bibliotecária responsável pela elaboração da ficha catalográfica

_____ _____

Assinatura Data

Marcio Nogueira Pereira da Silva

Estudo e Implementação de um Sistema Customizável para Controle Laboratorial para o Processo de Tipificação HLA

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Ciências Computacionais, da Universidade do Estado do Rio de Janeiro.

Aprovada em 14 de março de 2019.
Banca Examinadora:

Prof. Dr. Alexandre da Costa Sena (Orientador)
Instituto de Matemática e Estatística - UERJ

Prof. Dr. Leandro Augusto Justen Marzulo (Orientador)
Instituto de Matemática e Estatística - UERJ

Prof. Dr. Dilson Silva
Instituto de Matemática e Estatística - UERJ

Prof. Dr. Diego Nunes Brandão
Centro Federal de Educação Tecnológica Celso Suckow da Fonseca - CEFET/RJ

Prof. Dr. Luis Cristóvão de Moraes Sobrino Pôrto
Policlínica Piquet Carneiro - UERJ

Rio de Janeiro
2019

DEDICATÓRIA

A minha esposa Cristiane pela paciência sem fim.

AGRADECIMENTOS

Aos meus orientadores, professor Alexandre Sena, por me acolher e ajudar nesse caminho e o professor Leandro Marzulo pela grande oportunidade de participar desse projeto.

Agradeço também a equipe do laboratório HLA pela paciência e solicitude em prover as informações necessárias para que esse trabalho fosse concluído.

Um obrigado especial também para o Luís Cristovão, responsável pelo laboratório. Sem a iniciativa dele, esse trabalho nunca teria existido.

A meus sogros Sergio e Cristina pelo apoio nesse momento difícil da vida.

Ao pessoal da equipe UERJ-BOTZ pelas palavras de incentivo nas horas de cansaço e desânimo, e pelos momentos de descontração nas horas de tensão.

RESUMO

SILVA, Marcio Nogueira Pereira da. *Estudo e Implementação de um Sistema Customizável para Controle Laboratorial para o Processo de Tipificação HLA*. 2019. 109 f. Dissertação (Mestrado em Ciências Computacionais) – Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2019.

O avanço tecnológico trouxe grandes mudanças em todos os setores da sociedade. No tocante a área médica, a evolução nos últimos 100 anos fez com que o homem repensasse os limites do seu próprio corpo. Nesse período passamos da descoberta de grupos sanguíneos a realização de cirurgias robóticas passando por importantes marcos como a descoberta das mais diversas vacinas, por exemplo. Um grande destaque foram os sucessos obtidos nos transplantes de órgãos e tecidos. Foram muitos os fracassos porém na trajetória até o primeiro transplante bem sucedido, descobriu-se o processo de compatibilidade entre o doador e o receptor de órgãos e o importante papel exercido pelo sistema imunológico. O processo de rejeição ocorre em casos onde esse sistema identifica o transplante como um corpo estranho. Estudos mostraram que todas as espécies possuem um grupamento gênico do DNA responsável pelo sistema imune. Nos humanos esse grupamento é denominado HLA e também percebeu-se que uma compatibilidade favorável entre doador/receptor desse grupamento diminuem consideravelmente a chance de rejeição. Testes laboratoriais foram desenvolvidos para identificar esse grupamento e um deles é a tipificação HLA. Essa avaliação precisa ser realizada por laboratórios especializados e nesse contexto o Laboratório de Histocompatibilidade e Criopreservação da Universidade do Estado do Rio de Janeiro (HLA-UERJ) tem posição de destaque no setor. O processo como um todo é complexo e extenso, envolvendo diversas equipes e setores do laboratório, com uma intensa troca de informações. Os dados tratados são extremamente importantes, o que demanda um rígido controle para garantir a rastreabilidade e confiabilidade dos resultados. A automação de partes do processo laboratorial é bem comum nos dias de hoje e a integração de todo processo normalmente é feita por *softwares* de gestão laboratorial. O HLA-UERJ não conta com tal sistema, e a troca de informações é realizada de forma manual por formulários ou através de pequenas soluções computacionais desenvolvidas localmente. O presente trabalho apresenta um estudo detalhado do processo e uma implementação de um sistema de controle, customizável, *open source* para o laboratório com os objetivos principais de minimizar os riscos operacionais e prover uma maior capacidade de análise e rastreabilidade dos dados gerados.

Palavras-chave: HLA. histocompatibilidade. LIS.

ABSTRACT

SILVA, Marcio Nogueira Pereira da. *Study and Implementation of a Customized Laboratory Control System for the HLA Typing Process.* 2019. 109 f. Dissertação (Mestrado em Ciências Computacionais) – Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2019.

Technology's development has brought major changes in all sectors of society. As for the medical field, evolution over the last 100 years has made man rethink the limits of his own body. In this period humanity moved from the discovery of blood groups, to the execution of robotic surgeries, passing through important milestones such as the discovery of most diverse vaccines, for example. A major highlight was the success achieved in organ and tissue transplants. There were many failures, but in the trajectory to the first successful transplant, key discoveries were made: (i) ho to determine genetic compatibility between donor and organ recipient and (ii) the importance of the immune system in this process. Transplant rejection occurs in cases where the imune system identifies the transplanted organ as a foreign body. Studies have shown that all species have a genetic grouping of the DNA responsible for the immune system. In humans, this grouping is called HLA and it has also been observed that a favorable compatibility between donor / recipient of this grouping considerably reduces the chance of rejection. Laboratory tests were developed to identify this grouping and one of them is HLA typing. These evaluations need to be performed by specialized laboratories and in this context the Laboratory of Histocompatibility and Cryopreservation of the University of the State of Rio de Janeiro (HLA-UERJ) has a prominent position in the sector. The process as a whole is complex and extensive, involving diverse teams and sectors of the laboratory, with an intense exchange of information. Data processing is extremely important and requires strict control to ensure the traceability and reliability of results. Automation of parts of the laboratory process is very common these days and integration of the whole process is usually done by laboratory software. HLA-UERJ does not have such a system, and exchange of information is done manually by forms or through small computational solutions developed locally. This work presents a detailed study of the process and an implementation of a control system, customizable, open source for the laboratory with the main objectives of minimizing operational risks and providing greater capacity for analysis and traceability of the generated data.

Keywords: HLA. histocompatibility. LIS.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

BD	*Banco de Dados*
csv	*Comma-Separeted Values*
CoC	*Convention Over Configuration*
CSS	*Cascading Style Sheets*
DNA	*Deoxyribonucleic Acid*
DRY	*Don't Repeat Yourself*
FK	*Foreign Key*
HLA	*Human Leukocyte Antigen*
HTML	*Hypertext Markup Language*
JSON	*JavaScript Object Notation*
LIMS	*Laboratory Information Management System*
MIT	*Massachusetts Institute of Technology*
MVC	*Model-View-Controller*
PCR	*Polymerase Chain Reaction*
PDF	*Portable Document Format*
PK	*Primary Key*
RoR	*Ruby on Rails*
SSO	*Sequence-Specific Oligonucleotide*
TLA	*Total Automation System*

SUMÁRIO

INTRODUÇÃO

Os avanços das ciências e da tecnologia tiveram impactos significativos na formação da sociedade. As invenções e descobertas científicas dos últimos séculos influenciaram todas as áreas do comportamento humano, moldando como as pessoas se comunicam, trabalham, locomovem, vestem, entre outras mudanças. Um grande destaque nessa evolução ocorreu na área médica. Em pouco mais de 100 anos passamos do descobrimento de grupos sanguíneos por Karl Landsteiner em 1901 a realização de cirurgias remotas (robótica) como a realizada por Jacques Marescaux em 2001 - passando nesse caminho por diversos eventos importantes como por exemplo a criação de um sem número de vacinas (catapora, rubéola, hepatite A) e pela clonagem de seres vivos como a ovelha Dolly em 1996 [1] [2].

A inserção da tecnologia na área da medicina criou campos de pesquisa multidisciplinares (como a bioinformática, por exemplo), e fez o homem repensar os limites do seu próprio corpo. As pesquisas nessa área tiveram seu início na década de 1970, porém os últimos e rápidos avanços dos últimos 20 anos foram tão expressivos que fizeram com que o século XXI ficasse conhecido como "século da biotecnologia" [3].

Um grande destaque da evolução da medicina no século XX foram os transplantes de órgãos e tecidos. Embora atualmente a prática já seja vista como relativamente comum, tanto para órgãos sólidos como também para tecidos líquidos (como a medula óssea), é um procedimento relativamente novo. Um marco considerado como o início da fase experimental dos transplantes foi o desenvolvimento da técnica de sutura vascular por Alex Carrel em Lion (1897). Esse método é o mesmo utilizado atualmente e, por seus trabalhos na área, Alex Carrel recebeu o Prêmio Nobel de Medicina em 1912.

O primeiro transplante bem sucedido que se tem conhecimento foi renal e ocorreu no final de 1954, em gêmeos idênticos, na cidade de Boston nos EUA. No Brasil, o primeiro transplante de órgãos que se tem notícia também foi renal, ocorrido em abril de 1964 no Hospital dos Servidores do Estado, no Rio de Janeiro [4].

A longa e tortuosa trajetória até esses sucessos levou a descoberta da existência de processos que regem a compatibilidade entre o doador e o receptor de órgãos. Descobriu-se então o papel fundamental do sistema imunológico no transplante. Mecanismos complexos de imunidade, que em circunstâncias normais funcionam para identificar organismos estranhos e direcionar o sistema imunológico para destruí-los, podem também representar uma barreira significativa para um transplante bem-sucedido. Percebeu-se que em certos casos o sistema imunológico identificava o transplante como um corpo estranho, desencadeando uma resposta imunológica que acabava destruindo o órgão ou tecido transplantado. Esse fenômeno ficou conhecido como rejeição [5].

Estudos mostraram que todas as espécies possuem um grupamento gênico no DNA cuja função está diretamente ligada a resposta imunológica. Essa região foi denominada

complexo de histocompatibilidade principal - MHC (do inglês *Major Histocompatibility Complex*). As proteínas codificadas por certos genes desse grupo exercem uma função importante no controle do reconhecimento de antígenos próprios e antígenos externos. Nos humanos esse grupamento é denominado HLA (do inglês *Human Leukocyte Antigen*) pelo fato desses antígenos serem expressos nos leucócitos humanos.

Embora existam outros fatores não imunológicos que devem ser levados em consideração para realização do transplante como por exemplo a urgência médica, o tempo de espera, a idade do doador entre outros, foi comprovado que uma compatibilidade HLA favorável tem um papel significante para diminuir as chances de rejeição [6].

Existem testes laboratoriais que tem por intuito identificar a possibilidade de ocorrência desse problema antes do transplante em si ocorrer, e um deles é a tipificação HLA. Esse teste identifica os antígenos do sangue que poderiam vir a atacar o órgão transplantado. Outro teste realizado é o denominado "*crossmatching*" que busca determinar se já existem anticorpos formados no corpo do receptor contra os antígenos do doador.

Todas essas avaliações precisam ser realizadas por laboratórios especializados que contam com um tipo de equipamento específico para esse fim. Nesse contexto o laboratório de Histocompatibilidade e Criopreservação da Universidade do Estado do Rio de Janeiro (HLA-UERJ), ocupa uma posição de destaque no setor. O HLA-UERJ, localizado na Policlínica Piquet Carneiro, atua nas áreas de ensino, pesquisa e assistência, realizando exames de imunogenética pertinentes aos transplantes renais e atua como colaborador no cadastramento de doadores voluntários do Registro Nacional de Doadores de Medula Óssea (REDOME) [7].

Para a realização dos exames, o laboratório aplica técnicas de sequenciamento de DNA e identificação de genótipos do complexo principal de histocompatibilidade humano (HLA). O processo como um todo é bastante complexo, pois envolve diversas etapas, mobiliza diversos equipamentos e equipes alocados em diferentes setores dentro do laboratório. Há uma intensa troca de informações o que demanda um rígido controle para garantir a rastreabilidade e confiabilidade das amostras e dados gerados, e posterior arquivamento dos resultados.

Nas última décadas, a medicina laboratorial passou por grandes transformações. Pressões de mercado para redução de custos e maior produtividade, e também a busca por serviços com maiores padrões de qualidade, fizeram com que houvesse uma busca pela automação dos processos.

Atualmente, em um típico laboratório de análises clínicas, a maioria do volume processado é realizada por equipamentos automatizados [8]. Já até existem laboratórios que são completamente automatizados, ou seja, há pouca ou nenhuma interferência humana no processo. Embora esse tipo de automação esteja se tornando mais comum, o alto custo dos equipamentos ainda é impeditivo para que seja adotado como solução padrão. Dessa forma são mais utilizadas automações modulares, onde diversas etapas são realizadas por

equipamentos especializados, porém outras são realizadas por técnicos qualificados. [9].

Para integração de todo o sistema laboratorial e facilitar a difusão da informação, é comum a utilização de sistemas informatizados especializados. Essa forma de gestão laboratorial é conhecida como LIMS (do inglês *Laboratory Information Management System*) e consiste de um software que serve como um integrador de processos, englobando todo o fluxo de informações, dados dos usuários, informações das amostras, integração com os equipamentos, análise de dados, geração de relatórios, entre outros [10].

Embora tenha uma posição importante no setor laboratorial, o HLA-UERJ não possui algo similar a esse tipo de aplicação. Com o crescente volume de análises, o laboratório começou a sentir a necessidade de implementar melhorias na área de TI. O laboratório possui equipamentos modulares especialistas bastante avançados porém ainda estava faltando um agente para integrar as informações oriundas dos diversos equipamentos.

Dessa observação surgiu em 2016 uma parceria do HLA-UERJ com o LabCAD UERJ (Laboratório de Computação de Alto Desempenho da UERJ) para que fossem realizados diversos trabalhos que envolvessem a interação entre medicina e computação.

Um dos estudos consistia no desenvolvimento e implementação de um sistema customizado que viesse a auxiliar na gerenciamento das análises. Em 2017, Ribeiro [11] conduziu a primeira etapa, que foi realizar um mapeamento do exame de tipificação HLA, mostrando diversos detalhes e, identificando questões que deveriam ser abordadas no sistema. A autora apresentou uma possível estrutura de aplicação que atendesse ao mapeamento realizado.

Desse modo, esse trabalho aprofunda consideravelmente o estudo apresentado em Ribeiro [11] identificando e mapeando novas etapas. Além disso, este trabalho apresenta a implementação completa do sistema de controle para exames (especialmente os exames de tipificação HLA), assim como o início do processo de implantação do sistema no laboratório. Um importante destaque é que o sistema desenvolvido é customizável para que não só atenda as necessidades atuais do laboratório, mas que também possa ser expandido com outras funcionalidades de forma prática. Essa abordagem não estava planejada inicialmente e teve o objetivo de deixar a aplicação com o escopo mais amplo, podendo atender outros tipos de laboratórios.

Para o desenvolvimento, optou-se por construir uma aplicação *web*, ou seja, o usuário irá interagir com a aplicação a partir de um *browser* (*Firefox* ou *Google Chrome*, por exemplo), não necessitando que seja instalado nada nos computadores. A aplicação será toda gerida por um servidor contendo o código e o banco de dados necessário. Outra vantagem dessa abordagem é que com poucos ajustes é possível adaptar todo a aplicação para uso em dispositivos móveis (*tablets*, por exemplo).

Espera-se que com o produto do presente trabalho o laboratório HLA-UERJ possa atuar de forma ainda mais eficiente, pois terá uma poderosa ferramenta de controle de

processos laboratoriais.

Objetivos

O objetivo deste trabalho é apresentar uma proposta de um sistema computacional customizável para gerenciamento de processos laboratoriais. A motivação principal foi integrar as informações das atividades relacionadas ao processo de tipificação HLA, acompanhando todo o ciclo das análises realizadas e suas subetapas.

Percebeu-se durante o desenvolvimento do trabalho que o laboratório realiza outras atividades (por exemplo, exames moleculares referentes às hepatites virais – hepatite B e hepatite C), e por esse motivo optou-se por uma abordagem que deixa a implementação um pouco mais complexa porém flexível para inclusão de novas funcionalidade e de fácil atualização. Assim, o sistema proposto tem potencial para atender outros laboratórios, servindo como uma solução customizável e *open source* para a integrar as atividades laboratoriais.

Espera-se que a utilização desse sistema minimize os riscos operacionais e amplie a capacidade de análise e a rastreabilidade dos dados gerados.

Para realizar essa tarefa os seguintes objetivos específicos foram estabelecidos:

- Compreender conceitos básicos de biologia molecular necessários para o entendimento das atividades exercidas pelo laboratório;
- Estender o mapeamento básico do processo de tipificação HLA realizado por Ribeiro [11];
- Obter mais informações e detalhar o processo de tipificação HLA;
- Pesquisar sistemas laboratoriais similares disponíveis que pudessem atender aos requisitos funcionais do HLA-UERJ;
- Pesquisar e implementar boas práticas de desenvolvimento de *software* para tornar o sistema robusto e flexível;
- Pesquisar quais abordagens e ferramentas computacionais seriam necessárias para criação de um sistema customizável;
- Incluir no desenvolvimento do sistema características que o deixem apto a ser utilizado também em dispositivos móveis;
- Desenvolver e detalhar o sistema computacional de controle laboratorial.

Metodologia

Um obstáculo inicial que precisava ser transposto é o fato de duas áreas tão distintas estarem criando uma interface comum: a área da medicina laboratorial com a área da computação. Para facilitar a comunicação com a equipe local, percebeu-se que seria importante a compreensão de conceitos teóricos básicos sobre o trabalho que é realizado no laboratório. Por esse motivo foi conduzido um breve estudo sobre a biologia molecular envolvida nos processos. O material de base foram livros e artigos científicos, porém a colaboração da equipe do laboratório, sanando eventuais dúvidas, foi muito importante.

Em seguida foi preciso confirmar o mapeamento de processos do laboratório apresentado por Ribeiro [11] e obter mais informações para o detalhamento final do sistema. Para tal foi necessário um envolvimento com a equipe local e uma imersão na rotina de trabalho para compreensão do funcionamento do sistema interno.

Dentre as metodologias de pesquisa disponíveis, utilizou-se uma que estimula o envolvimento do pesquisador com o ambiente de pesquisa e com o problema a ser estudado. Nesse caso foi adotada a pesquisa participante.

> Na observação participante, o observador não é apenas um espectador do fato que está sendo estudado, ele se coloca na posição e ao nível dos outros elementos humanos que compõem o fenômeno a ser observado. Se o pesquisador está empenhado em estudar as aspirações, interesses ou rotina de trabalho de um grupo de operários, na forma de observação participante, ele terá de se inserir nesse grupo de operários como se fosse um deles. [12]

Toda a rotina laboratorial para realização de diversos exames foi observada algumas vezes, e os relatos dos funcionários sobre o processo em si, sugestões de melhorias e dificuldades foram anotados para posterior utilização.

Findada essa etapa, uma pesquisa foi realizada buscando sistemas comerciais disponíveis que pudessem atender aos requisitos de funcionamento do laboratório HLA-UERJ. Para isso buscou-se *softwares* laboratoriais específicos para área de HLA. Como fonte foi utilizada a internet.

A última etapa foi feita a confecção de um sistema utilizando o *framework* gratuito *Ruby on Rails*. Com intuito de se familiarizar melhor com a ferramenta foi conduzido um amplo estudo utilizando-se principalmente cursos *online* e tutoriais disponíveis na internet. Diversos protótipos foram concebidos testando algumas diferentes abordagens para construção e bibliotecas do próprio *framework*. Optou-se então pela construção de um sistema modular, de modo a ser facilmente expandido e atualizado [13].

Com intuito de garantir que o produto final fosse robusto e confiável, uma pesquisa sobre boas práticas de engenharia de *software* foi conduzida e, por fim, foi adotado o desenvolvimento guiado por testes. Um estudo foi feito sobre o assunto e pesquisou-se também como implementar a técnica no *framework* escolhido. A base para toda esse consulta foram livros e artigos disponíveis *online*.

Organização do Trabalho

O presente trabalho está estruturado da seguinte forma:

Na introdução foram apresentadas, a motivação para realização do trabalho, os objetivos e a metodologia utilizada. Cada tópico desse está em uma seção individual.

O capítulo 1, intitulado "O Complexo Principal de Histocompatibilidade", apresenta uma breve introdução ao mundo da biologia molecular envolvida nos processos laboratoriais estudados e contém somente a seção 1.1 O Sistema HLA.

O capítulo 2, intitulado "Processo de Tipificação HLA do HLA-UERJ" tem como intuito apresentar o laboratório HLA-UERJ, e o processo de tipificação HLA, descrevendo as etapas envolvidas. Está dividido nas seguintes seções: 2.1 Os exames realizados pelo laboratório HLA-UERJ, 2.2 Detalhamento e mapeamento do processo atual e 2.3 Identificação de oportunidades de melhoria.

No terceiro capítulo, intitulado "Sistemas Informatizados Laboratoriais", são discutidos os impactos da inserção da tecnologia nos laboratórios e apresentadas algumas soluções comercias de gestão de processos laboratoriais. Está dividido na seguinte seção: 3.1 Sistemas Comerciais de Gestão de Laboratórios.

No quarto capítulo, intitulado "Proposta e Implementação do Sistema Customizado para Controle Laboratorial para o Processo de Tipificação HLA" são apresentadas as ferramentas e métodos computacionais que foram utilizados na construção da aplicação e também a descrição do sistema construído, discutindo questões da implementação e apresentando telas da interface gráfica. Está divido nas seguintes seções: 4.1 Os Módulos da Aplicação, 4.2 A linguagem *Ruby* e o *Framework Rails*, 4.3 Por que *Ruby on Rails*?, 4.4 Modularidade, 4.5 Desenvolvimento Guiado por Testes e 4.6 Descritivo dos Módulos da Aplicação.

Por fim é apresentado um fechamento no capítulo intitulado "Conclusões", onde são apresentadas também sugestões para trabalhos futuros.

1 O COMPLEXO PRINCIPAL DE HISTOCOMPATIBILIDADE

O ser humano vive em um ambiente bastante hostil pois o coabita com uma grande variedade de organismos infecciosos como vírus, bactérias e fungos. Caso algum destes consiga infectar um indivíduo, e não haja algum mecanismo de controle sobre sua ação e multiplicação, não tardaria para que o organismo infeccioso consega levar ao colapso o ser infectado.

Felizmente existe um conjunto de células e moléculas responsáveis pela defesa contra infecções para evitar esse tipo de ocorrência, que constituem o sistema imune ou imunológico. A principal função desse sistema é a de discernir antígenos estranhos (aloantígenos) de antígenos próprios (auto-antígenos) dos tecidos corporais. Dessa forma, mecanismos podem ser ativados quando forem necessárias defesas contra organismos infecciosos [14].

Um efeito indesejável do sistema imunológico é a rejeição a enxertos e transplantes realizados entre pessoas incompatíveis imunologicamente. Nesse caso as proteínas celulares do doador são reconhecidas como estranhas ao receptor e dessa forma ocorre a rejeição do órgão doado [5].

1.1 O Sistema HLA

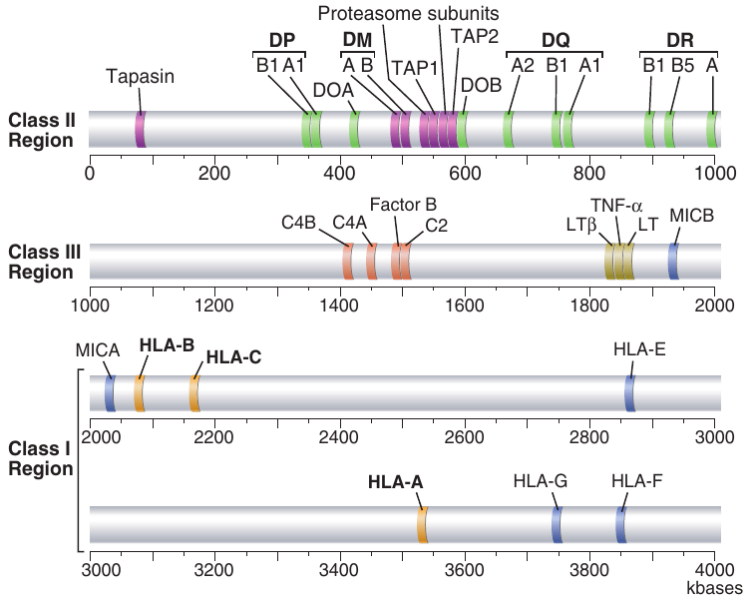
O Complexo Principal de Histocompatibilidade (MHC - do inglês *Major Histocompatibility Complex*) é uma região genômica que codifica proteínas que podem ser encontradas na superfície das células que auxiliam o sistema imunológico a reconhecer substâncias estranhas. As proteínas codificadas pelo MHC são encontradas em todos os vertebrados superiores. Nos seres humanos, esse complexo é denominado de HLA (do inglês *Human Leukocyte Antien*).

Existem dois tipos principais de moléculas de proteína do MHC, divididos didaticamente em Classe I e Classe II. A região de Classe I, composta, principalmente pelos *loci* HLA-A, -B e -C, codificam moléculas que conseguem atravessar a membrana de praticamente qualquer célula do organismo, enquanto a região de Classe II, que engloba os *loci* HLA-DR, -DP e -DQ (cada um desses ainda contêm genes separados denominados A e B), codificam moléculas que estão restritas a células do sistema imunológico, denominadas macrófagos e linfócitos. Em humanos essas moléculas são codificadas por vários genes, todos agrupados na mesma região, localizada no braço curto do cromossomo 6 (na posição 6p21.3). Cada gene tem um número muito grande de alelos (formas alternativas de um gene que produz formas alternativas de proteínas). Como resultado, é muito raro que dois indivíduos tenham o mesmo conjunto de moléculas de MHC. O MHC contém também

uma variedade de genes que codificam outras proteínas, como proteínas do sistema complemento, citocinas (mensageiros químicos) e enzimas, que são chamadas de moléculas de MHC de classe III.

A Figura 1, apresenta a região do *loci* do MHC humano, destacando a localização das classes e dos *loci* importantes mencionados anteriormente.

Figura 1 - *Loci* to MHC humano



Fonte: [15]

As moléculas de MHC são componentes importantes do sistema imune porque permitem que os linfócitos T (tipo de glóbulo branco) detectem células, tipo macrófagos que ingeriram microrganismos infecciosos. Quando um macrófago engolfa um microrganismo, digere-o parcialmente e exibe fragmentos peptídicos do micróbio em sua superfície, ligados a moléculas de MHC. O linfócito T reconhece o fragmento estranho ligado à molécula do MHC e se liga a ele, estimulando uma resposta imune. Em células saudáveis não infectadas, a molécula de MHC apresenta peptídeos de sua própria célula (peptídeos próprios), aos quais a células T normalmente não reagem.

As moléculas de MHC foram inicialmente definidas como antígenos que estimulam a resposta imunológica de um organismo aos órgãos e tecidos transplantados. Na década de 1950, experimentos de enxerto de pele realizados em camundongos mostraram que a rejeição do enxerto era uma reação imune criada pelo organismo hospedeiro contra tecidos estranhos. O hospedeiro reconheceu as moléculas do MHC nas células do enxerto como antígenos estranhos e as atacou. Assim, o principal desafio em um transplante bem-sucedido é encontrar um hospedeiro e um doador com tipos tão semelhantes quanto possível [15].

Estudos já comprovaram a grande importância da compatibilidade HLA entre doador e receptor. Em transplantes de rim, um estudo conduzido entre 1987 e 2013 concluiu que a incompatibilidade de somente um HLA já aumentava em 13% o risco de ocorrer rejeição, enquanto 6 incompatibilidades chegavam a aumentar em até 64% esse risco [16].

Com relação ao transplante de médula óssea, um estudo conduzido no Japão pelo Programa de Doação de Medula Óssea local usou uma população de 7898 pacientes que receberam transplante de doadores japoneses para medir os riscos associados a incompatibilidade em cada um dos *locus* do HLA e o impacto da incompatibilidade em múltiplos *locus*. Das várias conclusões, percebeu-se um elevado risco de rejeição aguda quando há incompatibilidade em HLA-A, -B, -C ou -DPB1. O estudo ressalta também que a taxa de mortalidade se eleva consideravelmente quando ocorrem incompatibilidades em HLA-DRB1 e -DQB1 simultaneamente [17].

2 PROCESSO DE TIPIFICAÇÃO HLA DO HLA-UERJ

O Laboratório de Histocompatibilidade da Universidade do Estado do Rio de Janeiro (HLA-UERJ) iniciou sua história no final de 1994. Na época ocupava algumas salas do Departamento de Histologia e Embriologia do Instituto de Biologia da UERJ (IBRAG-UERJ) [7]

Em 1996, o HLA-UERJ foi credenciado pela Associação Brasileira de Histocompatibilidade (ABH) quando passou também a ser responsável pelos exames de imunogenética relacionados aos transplantes renais efetuados no Hospital Universitário Pedro Ernesto (HUPE-UERJ) e também começou a atuar como colaborador no cadastramento de doadores voluntários do Registro Nacional de Doadores de Medula Óssea (REDOME) [7]

Em 2003, o laboratório foi transferido para a Policlínica Piquet Carneiro em função da incorporação de equipamentos e da realização de novos testes, como os exames de genotipagem do vírus da Hepatite C. A partir de 2007, passou a se chamar laboratório de Histocompatibilidade e Criopreservação, estendendo suas atividades para o uso de células e tecidos preservados [7].

Atualmente, o laboratório ocupa, juntamente com o Laboratório de Diagnóstico por DNA (LDD-UERJ) e o Laboratório de Reparo Tecidual (LRT-UERJ), o Pavilhão José Roberto Feresin Moraes, localizado na Policlínica Piquet Carneiro. Possui uma equipe com cerca de 25 colaboradores, entre servidores, prestadores de serviço, pesquisadores e alunos de pós-graduação, voltados para a realização de exames de compatibilidade na área de transplante de órgãos sólidos e células-tronco hematopoéticas, exames moleculares referentes às hepatites virais – hepatite B e hepatite C – e criopreservação de células-tronco hematopoéticas [7].

As atividades atuais estão distribuídas entre a realização de exames laboratoriais, a participação em campanhas para cadastramento de doadores voluntários de medula óssea e a oferta de cursos de formação e aperfeiçoamento, além da participação em importantes projetos de pesquisa nestas áreas [7].

O HLA-UERJ possui acreditação do Instituto Nacional de Metrologia, Qualidade e Tecnologia (INMETRO), concedida com base no cumprimento dos requisitos das normas e em documentos normativos [7].

2.1 Os exames realizados pelo laboratório HLA-UERJ

A Medicina Laboratorial tem como enfoque primordial a busca por novas formas de diagnóstico e terapêutica de doenças de grande relevância social. O Laboratório de

HLA da UERJ possui um grupo multidisciplinar de pesquisadores e doutores, tendo como foco estudos de associação de alelos do Complexo Principal de Histocompatibilidade com doenças, principalmente nas áreas de Reumatologia e Dermatologia, como por exemplo a febre reumática (15-18.000 novos casos/ano [18]) e em pesquisas de incidência familiar. O laboratório tem contribuído também na pesquisa em transplantes renais. Uma das fronteiras científicas nesta área é a detecção e prevenção de rejeições crônicas renais através de exames laboratoriais que permitam acompanhar a evolução de pacientes no pós-transplante e influir na terapêutica empregada visando a escolha do melhor esquema terapêutico para o paciente e a minimização dos custos. O serviço de Histocompatibilidade é oferecido à comunidade e investe no aperfeiçoamento de técnicas diagnósticas e no estudo da resposta humoral ao enxerto [7].

Os seguintes exames são realizados no laboratório:

- **Tipificação HLA de Receptor de transplante de células-tronco hematopoéticas** - Este exame permite a tipificação HLA com a identificação dos alelos HLA-A, HLA-B e HLA-C, no nível de baixa ou média resolução, e HLA-DRB1 e HLA-DQB1 em alta resolução, dos pacientes candidatos a transplante de células-tronco hematopoéticas.

- **Tipificação HLA de Doador Aparentado de células-tronco hematopoéticas** - Este exame permite a tipificação HLA com a identificação dos alelos HLA-A, HLA-B e HLA-DRB1 no nível de baixa ou média resolução dos candidatos à doação aparentada de células-tronco hematopoéticas.

- **Tipificação HLA de Doador Voluntário de células-tronco hematopoéticas – 1ª fase** - Este exame permite a tipificação HLA com a identificação dos alelos HLA-A, HLA-B e HLA-DRB1, no nível de baixa resolução, dos doadores voluntários cadastrados pelo REDOME.

- **Tipificação HLA de Doador Voluntário de células-tronco hematopoéticas – 2ª fase** - Este exame permite a complementação da tipificação HLA de doadores voluntários, potencialmente compatíveis com pacientes candidatos a transplante de células-tronco hematopoéticas. Consiste na identificação dos alelos HLA-C e HLA-DRB1, no nível de alta resolução, de doadores voluntários previamente selecionados.

- **Tipificação HLA Confirmatória de Doador Voluntário de células-tronco hematopoéticas** - Este exame permite a confirmação da tipificação HLA de doadores voluntários candidatos ao transplante de células-tronco hematopoéticas; consiste na identificação dos alelos HLA-A, HLA-B, HLA-C, HLA-DRB1 e HLA-DQB1 no nível de alta resolução.

- **Prova Cruzada (Crossmath) em Doador Vivo de Rim** - Este exame permite avaliar a presença de anticorpos anti-HLA contra células do doador no soro do receptor de órgão sólido e está indicado para todos os pacientes que serão submetidos a transplante renal.

- **Avaliação de Reatividade contra Painel (PRA) de Receptor de Rim** - Este exame permite a pesquisa de anticorpos anti-HLA no soro do receptor e está indicado para pacientes candidatos a um transplante de rim.

- **Tipificação HLA de Receptor de Rim** - Este exame permite a tipificação HLA com a identificação dos alelos HLA-A, HLA-B e HLA-DRB1, no nível de baixa ou média resolução, dos candidatos a transplante renal.

- **Tipificação HLA de Doador Vivo de Rim – 1ª fase** - Este exame permite a tipificação HLA com a identificação dos alelos HLA-A, HLA-B e HLA-DRB1 no nível de baixa ou média resolução dos candidatos a doação para transplante renal.

2.2 Detalhamento e mapeamento do processo atual

Ribeiro [11] realizou em 2017 um primeiro estudo da logística envolvida na realização do processo de tipificação HLA no laboratório HLA-UERJ. Com base em suas observações, a autora propôs um diagrama que detalha de forma bastante clara o fluxo de informações do processo de tipificação HLA.

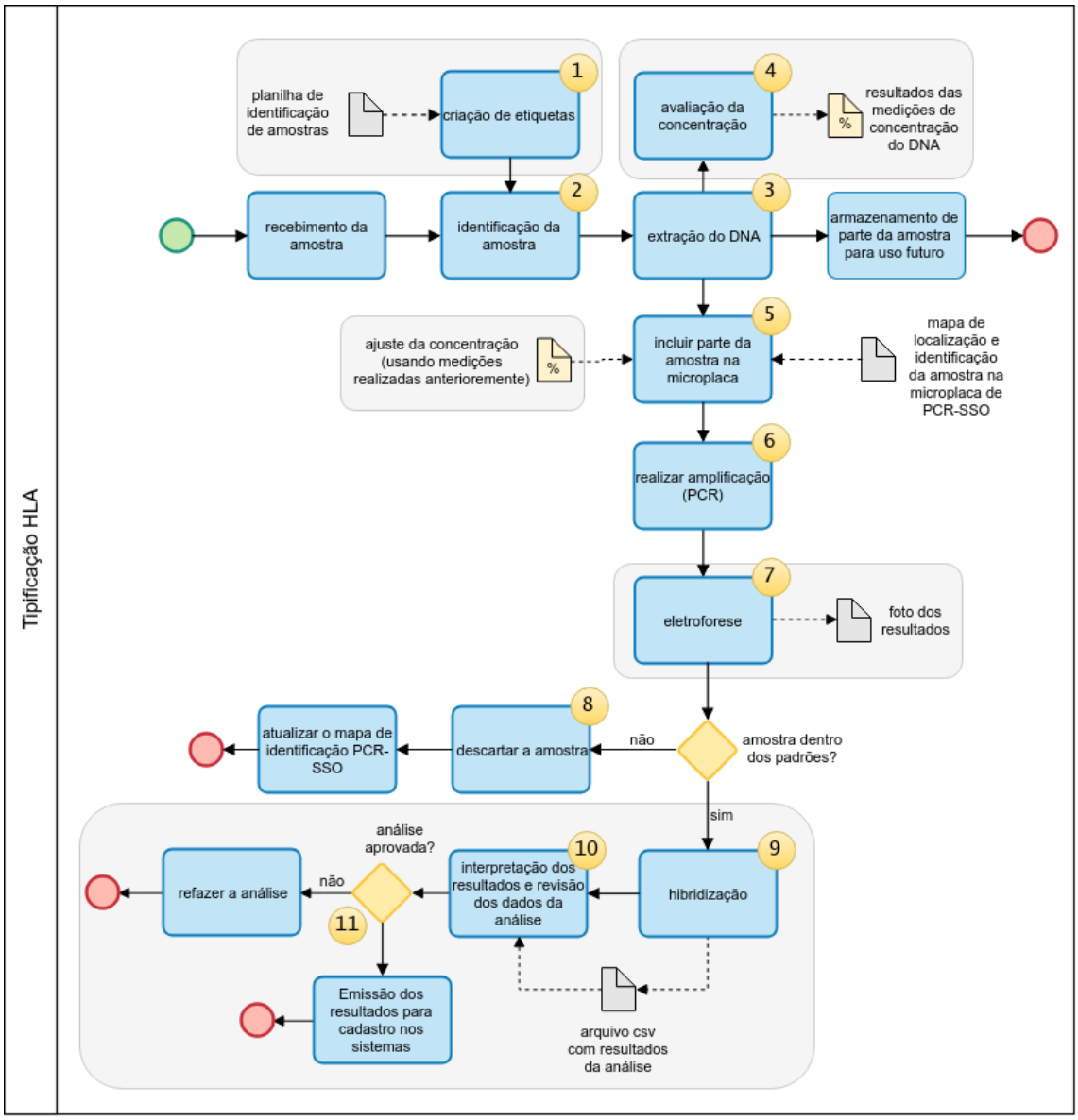
Utilizando como base essa primeira abordagem, o autor acompanhou a rotina laboratorial e fez uma releitura de todo o processo com intuito de confirmar o exposto pela autora e acrescentar mais informações quando necessário. Essas visitas ocorreram em paralelo com a criação de protótipos diversos para estudo de como melhor abordar o problema do ponto de vista computacional. Durante a implementação, surgiram ainda algumas questões, que exigiram outras visitas para obtenção de ainda mais detalhes.

O diagrama do processo apresentado por Ribeiro [11] foi alterado e o resultado é apresentado na Figura 2. As partes sombreadas foram as áreas onde ocorreram mudanças.

A seguir é apresentado um breve resumo das atividades observadas. A numeração apresentada ($\textcircled{x}$, com $x = 1, 2, 3, ...$) na Figura 2 tem como intuito destacar algumas partes do processo ao longo da descrição.

Logo após o recebimento das amostras, existe uma importante etapa administrativa de verificação de todo o material recebido. A solicitação de análise que chega ao laboratório vem acompanhada de uma lista com informações sobre os doadores obtidas no momento da coleta e as amostras em si. Cabe então ao laboratório fazer uma verificação completa das informações para filtrar alguns poucos casos de erro, como pessoas que

Figura 2 - Fluxo do processo de tipificação HLA realizado no HLA-UERJ



Fonte: O autor, 2018. Baseado em Ribeiro [11] e anotações próprias

já haviam doado em campanhas anteriores (resultados já disponíveis) ou amostras não recebidas.

Uma parte dessa verificação consiste em etiquetar os tubos das amostras para fácil identificação posterior. Essa etiqueta é afixada nos tubos da amostras (denominados *eppendorf* - Figura 3) e as informações apresentadas são oriundas da lista recebida com dados dos doadores. A Figura 4 apresenta um exemplo de etiqueta. Essa etapas correspondem aos passos (1) e (2) na Figura 2. Atualmente as etiquetas são geradas utilizando o *software* comercial da empresa Pimaco®.

Figura 3 - Tubo utilizado para armazenamento das amostras (eppendorf)

Fonte: [19]

Figura 4 - Exemplo de Etiqueta

0000000 C0000 PEDRO ALVARES CABRAL **22/04/1500 - 123**	**código DNA** **nº interno** nome do doador **data da coleta - nº da campanha**

Fonte: O autor, 2018

Nessa etapa ocorre também um processo de cadastro interno das amostras. Este é feito em uma planilha Excel® (Microsoft®). Fazem parte deste registro algumas informações como nome do doador, endereço, médico ou instituição que requiriu os exames e um código identificador interno, denominado "número de DNA" (número inteiro, sequencial e único).

A primeira etapa do processo de tipificação HLA começa no setor do laboratório denominado "Sala de Extração". Nesse laboratório são realizadas as primeiras etapas do processamento da amostra, que consiste na separação das células que possuem DNA dos demais componentes presentes, e posterior extração do DNA do núcleo da célula.

Esse processo é composto por etapas físicas (utilizando uma centrífuga para acelerar o processo de decantação) e químicas (utilizando *kits* de reagentes comerciais que fazem a lise de membranas celulares). Esse processo corresponde a etapa (3).

Para finalizar, é preciso mensurar a concentração de DNA em cada amostra. Essa informação é útil para realizar cálculos de ajuste de concentração em processos futuros. Isso é feito em um outro laboratório e os resultados são anotados em uma planilha denominada "Controle de Quantificação de amostras de DNA" (Etapa (4)).

Uma parte do produto dessa manipulação é separado para dar continuidade no processo e o restante é armazenado em câmaras frigoríficas para uso futuro, como por exemplo no caso da necessidade de exames mais detalhados.

Em seguida ocorre um processo de amplificação do DNA extraído através da técnica PCR-SSO [20]. Porém antes é preciso preparar as amostras para o procedimento e isto é feito em outro laboratório, denominado "Sala de Preparo de Reações Pré-PCR".

Nesse momento o DNA extraído anteriormente é transferido para uma microplaca com 96 poços (Figura 5), arranjados em 8 linhas e 12 colunas. As linhas são marcadas com letras (A - H) e as colunas com números (1 -12). É preciso identificar apropriadamente o conteúdo de cada poço, relacionando-o com a amostra colocada nele. Isso é feito preenchendo um formulário denominado "Mapa de Trabalho para Tipificação HLA por PCR-SSO" (Figura 6). Nesse mapa, cada espaço referente a um poço é preenchido com o número interno criado anteriormente ("número de DNA") e o tipo de exame que será realizado. Também são incluídos reagentes importantes para a etapa seguinte e são feitos ajustes que levam em consideração dados de concentração de DNA já obtidos. Informações sobre estes reagentes são preenchidas no mapa de trabalho (lote e data de validade).

Figura 5 - Exemplo de placa utilizada para amplificação do DNA

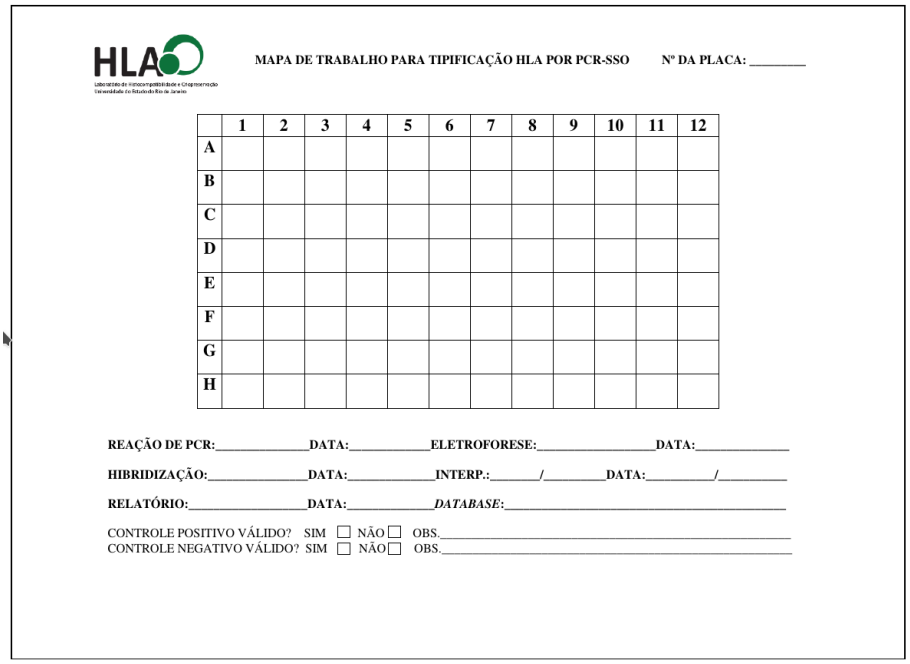
Fonte: [21]

As microplacas recebem uma numeração interna, que consiste de um número inteiro, sequencial e único. Toda essa manipulação corresponde a etapa (5) na Figura 2.

Agora com as amostras selecionadas e o mapa preenchido a próxima etapa é realizada na "Sala Pós PCR". A microplaca agora é submetida a um equipamento denominado termociclador para realização do processo de amplificação do DNA pelo método PCR-SSO [20]. No mapa de trabalho são registradas a data de realização do procedimento e o nome do funcionário que o realizou (etapa (6) na Figura 2).

Para garantir que houve êxito no processo anterior é utilizada a técnica denominada

Figura 6 - Mapa de localização



HLA

MAPA DE TRABALHO PARA TIPIFICAÇÃO HLA POR PCR-SSO Nº DA PLACA: _____

	1	2	3	4	5	6	7	8	9	10	11	12
A												
B												
C												
D												
E												
F												
G												
H												

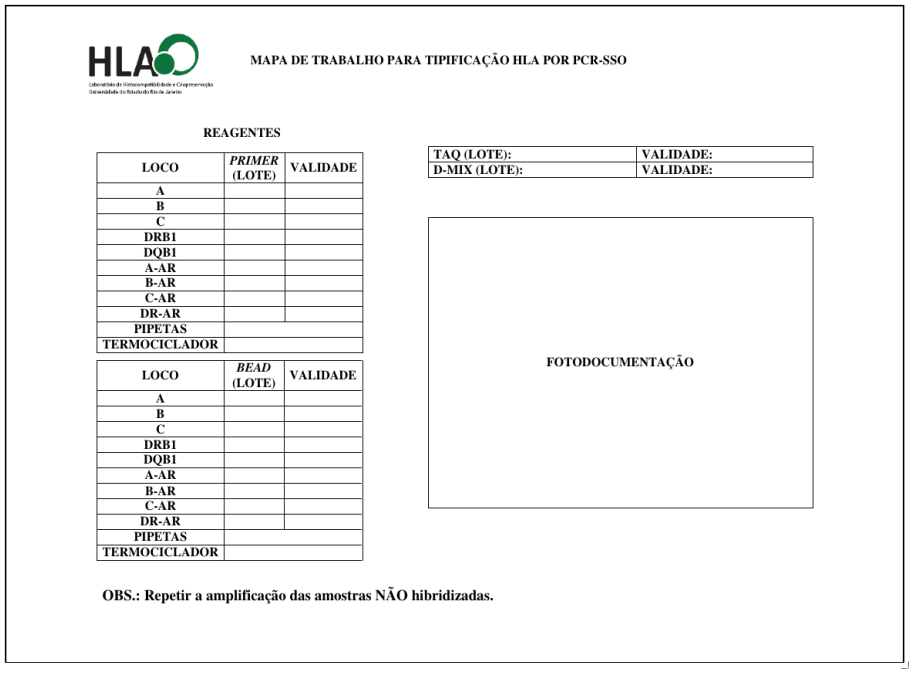
REAÇÃO DE PCR:_____DATA:_____ELETROFORESE:_____DATA:_____

HIBRIDIZAÇÃO:_____DATA:_____INTERP.:_____/_____DATA:_____/_____

RELATÓRIO:_____DATA:_____*DATABASE*:_____

CONTROLE POSITIVO VÁLIDO? SIM ☐ NÃO ☐ OBS._____
CONTROLE NEGATIVO VÁLIDO? SIM ☐ NÃO ☐ OBS._____

(a)



HLA

MAPA DE TRABALHO PARA TIPIFICAÇÃO HLA POR PCR-SSO

REAGENTES

LOCO	*PRIMER* (LOTE)	VALIDADE
A		
B		
C		
DRB1		
DQB1		
A-AR		
B-AR		
C-AR		
DR-AR		
PIPETAS		
TERMOCICLADOR		

LOCO	*BEAD* (LOTE)	VALIDADE
A		
B		
C		
DRB1		
DQB1		
A-AR		
B-AR		
C-AR		
DR-AR		
PIPETAS		
TERMOCICLADOR		

TAQ (LOTE):	VALIDADE:
D-MIX (LOTE):	VALIDADE:

FOTODOCUMENTAÇÃO

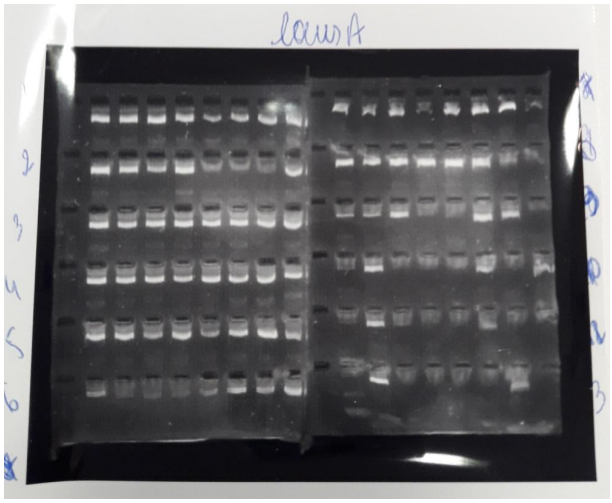
OBS.: Repetir a amplificação das amostras NÃO hibridizadas.

(b)

Fonte: HLA-UERJ - obtido no laboratório: (a) Frente; (b) Verso

"eletroforese em gel". O resultado é uma imagem similar a apresentada na Figura 7, que o técnico utiliza como base para avaliação de quais amostras estão dentro dos padrões. Essa imagem é impressa e anexada no mapa de trabalho como evidência e para fins de arquivo. Também é registrada a data da avaliação e o nome do funcionário que realizou o trabalho (etapa (7) na Figura 2).

Figura 7 - Resultado da eletroforese em gel aplicada em amostras



Fonte: O autor, 2017

As amostras reprovadas são descartadas e não seguirão para a etapa seguinte. Esse fato também é registrado no mapa de trabalho (caminho iniciado na etapa em (8) na Figura 2). A próxima etapa consiste na hibridização do DNA com Oligonucleotídeos de Sequência Específica (SSO). Nesse momento a metodologia a ser utilizada dependerá do tipo de análise que foi solicitada - média ou alta resolução. Para essa análise é utilizado um equipamento denominado citômetro de fluxo da empresa Luminex® (Figura 8).

As amostras precisam passar por um pré-processamento para ajuste da concentração conforme especificações do fabricante do equipamento. Isso é feito utilizando um *kit* de reagentes e a exata quantidade necessária é calculada com auxílio de uma planilha em Excel®.

O computador acoplado ao equipamento possui um software próprio que permite a configuração da máquina de acordo com a análise a ser realizada. Os resultados são fornecidos em um arquivo com extensão *csv* e são copiados do computador para etapa seguinte de interpretação (via porta USB, pois o computador não pode ser conectado a rede por recomendação do fabricante).

Esses processo correspondem a etapa (9) na Figura 2.

Os arquivos são levados para o escritório e são lidos em um *software* específico para esse fim (mais detalhes em [11]).

Figura 8 - Máquina Luminex onde são realizas as análises

Fonte: [22]

Nesse ponto são extraídas informações sobre testes de controle de qualidade do processo realizado (denominados controle positivo e controle negativo) e as informações são preenchidas no mapa de trabalho.

Em seguida são realizadas interpretações dos resultados por dois funcionários diferentes. Ambos colocam a data que realizaram a avaliação e seus nomes no mapa de trabalho.

Uma última verificação de todo o processo é feita por um funcionário supervisor (Etapa (10) na Figura 2).

Estando todas as informações como esperado, um relatório final é então gerado (Etapa (11) na Figura 2), cujos dados seguirão para cadastro em dois sistemas distintos. Um é o sistema laboratorial denominado BiosLab/Sil e o outro é o Redome. Caso contrário, é necessário repetir todo o procedimento.

Ao fim, todo o material escrito é arquivado seguindo normas internas e o processo é dado como encerrado.

2.3 **Identificação de oportunidades de melhoria**

A forma como os processos laboratoriais são conduzidos atualmente é bastante dependente da troca de extensos formulários preenchidos manualmente pelos funcionários e passados de um setor para outro. Informações importantes são digitalizadas. Porém a falta de um agente integrador fez com que cada setor desenvolvesse pequenas soluções próprias para armazenamento/ manuseio de dados, utilizado para isso softwares disponíveis, como por exemplo o Excel® da Microsoft®. Dessa forma, as informações acabam ficando fragmentadas em diversos arquivos dificultando uma rápida recuperação, controle de redundância e colocando em dúvida até mesmo a integridade dos dados. A ausência

de uma gerência central ainda dificulta o acompanhamento do progresso.

A utilização de formulários ainda levanta importantes questões relativas ao manuseio, arquivamento e escrita. Atualmente os papéis são armazenados em pastas arquivos guardados em armários, o que ocupa um grande espaço físico no laboratório. A tarefa de consulta é difícil e demorada pois, com o passar dos anos, um grande número de arquivos se acumulou. A simples existência física do papel já supõe um risco de que este pode ser perdido ou danificado acidentalmente ou até pela ação do tempo. A caligrafia também pode ser confusa ou até ilegível, pois depende do funcionário que preencheu e a limitação de espaço para escrever pode fazer com que informações importantes deixem de ser registradas.

Observou-se também que um formulário em particular, nomeado "Mapa de trabalho para tipificação HLA por PCR-SSO" (Figura 6) demanda um tempo de preenchimento bastante extenso dada a quantidade informações necessárias.

Dessa forma também é praticamente impossível obter dados estatísticos, que seriam interessantes para estudo do desempenho das atividades, como por exemplo, o tempo de execução de cada etapa e onde ocorrem os maiores retrabalhos.

Uma grande questão observada, que é impeditiva da implementação do sistema no momento, é a ausência de computadores em todos os laboratórios e a falta de uma estrutura de rede adequada. Sem isso, restringe-se bastante a utilização da aplicação, embora ainda possa ser de bastante utilidade. Para um completo funcionamento, seriam necessários computadores em todos os laboratórios e, em alguns, se possível, telas com o recurso de *touch screen* ou *tablets*, pois o teclado não é de fácil utilização dependendo do procedimento que está sendo realizado.

3 SISTEMAS INFORMATIZADOS LABORATORIAIS

A crescente evolução do tecnologia está impactando de forma drástica as mais diversas áreas da sociedade e, como não podia deixar de ser, a área da saúde também está passando por grandes reestruturações para se adaptar a essa nova realidade. A pressão do mercado por excelência e maior eficiência nos processos para redução de custos, também está atingindo os laboratórios clínicos, e isso está levando a busca por uma automatização de processos.

A primeira automação laboratorial documentada foi realizada pelo Dr. Masahide Sasaki na Escola Médica de Kochi no Japão no início da década de 80. Utilizando a tecnologia disponível na época, Sasaki e sua equipe realizaram diversas modificações em equipamentos existentes para adequar as suas necessidades. Eles adaptaram esteiras rolantes para carregar as amostras até estações de trabalho analíticas, onde então eram manipuladas por braços robóticos para realização de tarefas mais complexas. Todas as estações podiam operar sem a intervenção humana e o processo inteiro era controlado e monitorado por computadores. Dessa forma foi possível realizar todos os testes laboratoriais de um hospital de 600 leitos com uma equipe de apenas 19 funcionários. Comparativamente, nessas condições, normalmente seriam necessários até cerca de 10 vezes mais técnicos especializados. Esse tipo de automação ficou conhecido como TLA (do inglês *Total Automation System*) [23].

Apesar das claras vantagens que um modelo TLA apresenta, os altos custos para implementação desse tipo de tecnologia tornaram-a pouco atraente para laboratórios de médio/pequeno porte. Para poder disponibilizar a automatização para todos os laboratórios, os vendedores desses equipamentos optaram por desmembrar o sistema completo em pedaços, assim produzindo pequenos módulos especialistas. Hoje em dia praticamente todos os laboratórios existentes possuem algum grau de automação.

Uma grande questão surgiu dessa separação em módulos: como gerenciar o controle do processo como um todo? No modelo completo, o TLA, fazia parte do projeto do laboratório um ponto de controle central de todos os processos. Nessa automatização parcial, cada módulo tem o seu próprio controle e disponibiliza os resultados somente do que é responsável, sem integração com o resto do laboratório.

A demanda por uma solução para essa questão, aliada ao crescente volume de informações (gerada pela facilidade de realizar as análises por causa da modularização de processos), fez como que surgissem sistemas de gerenciamento de informações que tinham o intuito de administrar todo o fluxo de informações de um laboratório, independente de como ele funcionasse. Esse tipo de sistema ficou conhecido como LIMS (do inglês *laboratory information management system*) [24]. Um LIMS eficaz acompanha todo o ciclo de vida das amostras e análises no processo laboratorial, permite integrar instrumentos e

gerencia dados do laboratório em si permitindo o controle e a avaliação de informações de maneira rápida e segura.

Algumas características podem ser destacadas como importantes para o sistema [25]:

- Gestão da auditoria: controlar e manter registros das auditorias realizadas;
- Atribuição de responsabilidades: permitir a criação de regras para controle de acesso aos registros de dados específicos;
- Conformidade: seguir as normas reguladoras necessárias;
- Gestão de documentos: processar documentos e gerir a forma como os documentos são distribuídos e acessados;
- Calibração de instrumentos e manutenção: manter um registro detalhado sobre a calibração e manutenção dos instrumentos do laboratório;
- Entrada de dados de forma manual ou eletrônica: fornecer interfaces intuitivas e confiáveis para que informações possam ser inseridas manualmente ou eletronicamente;
- Relatórios: gerar relatórios em um formato específico, conforme demanda do laboratório;
- Rastreabilidade: permitir a rápida auditoria e/ou verificação de estado de amostras/ análises;
- Fluxos de trabalho: permitir o acompanhamento do ciclo de vida dos processos sendo executados.

Um grande obstáculo enfrentando para implementação desse tipo de sistema é a variação existente no funcionamento dos laboratórios. Embora os processos realizados sejam semelhantes, podem haver alterações devido ao tipo de equipamento adotado, disposição dos mesmos nos recintos, procedimentos utilizados, entre outros fatores, fazendo com que a adoção de um *software* de controle de processos seja difícil. Se o aplicativo não for flexível, irá gerar a necessidade de realização de alterações em rotinas do laboratório para ajustar-se a ele, o que em muitos casos não é viável.

3.1 **Sistemas Comerciais de Gestão de Laboratórios**

Para tentar comparar os recursos disponíveis para o sistema desenvolvido frente a soluções comerciais, foi feita uma busca por sistemas laboratoriais similares. A intenção era testá-las e avaliar se estas se adequariam a realidade do laboratório HLA-UERJ.

A pesquisa revelou que já existem algumas soluções comerciais para gerenciamento de processos laboratoriais, porém quando restringimos a procura por alguma que esteja mais próxima da realidade de um laboratório de HLA, não restam muitas opções. Os sistemas encontrados foram:

- mTilda (<http://www.mtilda.com/>);
- Orpheus (<http://www.hlasoft.com/index.php/hla-laboratory-management-software>);

3.1.1 Descritivo dos Sistemas Comerciais

Nenhum dos sistemas possuía uma versão demo para avaliação de suas características, portanto não foi possível realizar um estudo detalhado das funções de modo a avaliar se os mesmos se adequariam a rotina do laboratório HLA-UERJ. Por esse motivo as descrições apresentadas a seguir foram obtidas das informações disponíveis no *site* das empresas desenvolvedoras.

O sistema "mTilda" fornece uma descrição bastante simples em sua página, ressaltando como pontos fortes o fato de já serem adotados por diversos laboratórios nos Estados Unidos (não são mencionados quais) e o fato de que existe um grande número de expansões disponíveis para a aplicação, desenvolvidas ao longo dos anos de acordo com os comentários dos usuários.

Dois módulos aparecem em destaque, indicando o que parecem ser produtos independentes. Um deles, nomeado "*VXmatch*" realiza testes de compatibilidade (*crossmatch*) e o outro, nomeado "*3Ring*" faz o serviço de uma biblioteca virtual de documentação.

A aplicação está disponível somente para o sistema operacional *Windows*®.

Não são apresentados aspectos do funcionamento da estrutura de *back-end*, ou seja, como fica estruturada a comunicação com o servidor, como é estruturado o banco de dados, entre outras informações.

A Figura 9 apresenta uma tela parcial da aplicação, única disponibilizada pela empresa.

Embora o "Orpheus" também não ofereça opção de testes, a página da aplicação é mais detalhada. O sistema parece ser bastante robusto e conta com uma grande gama de funcionalidades como cadastro de pacientes/ doadores, acompanhamento de análises, gerenciamento de usuários, interface para parte de cobrança, entre outros. Também está disponível somente para o sistema operacional *Windows*®.

De forma análoga ao anterior não estão disponíveis informações sobre o funcionamento da parte de *back-end*.

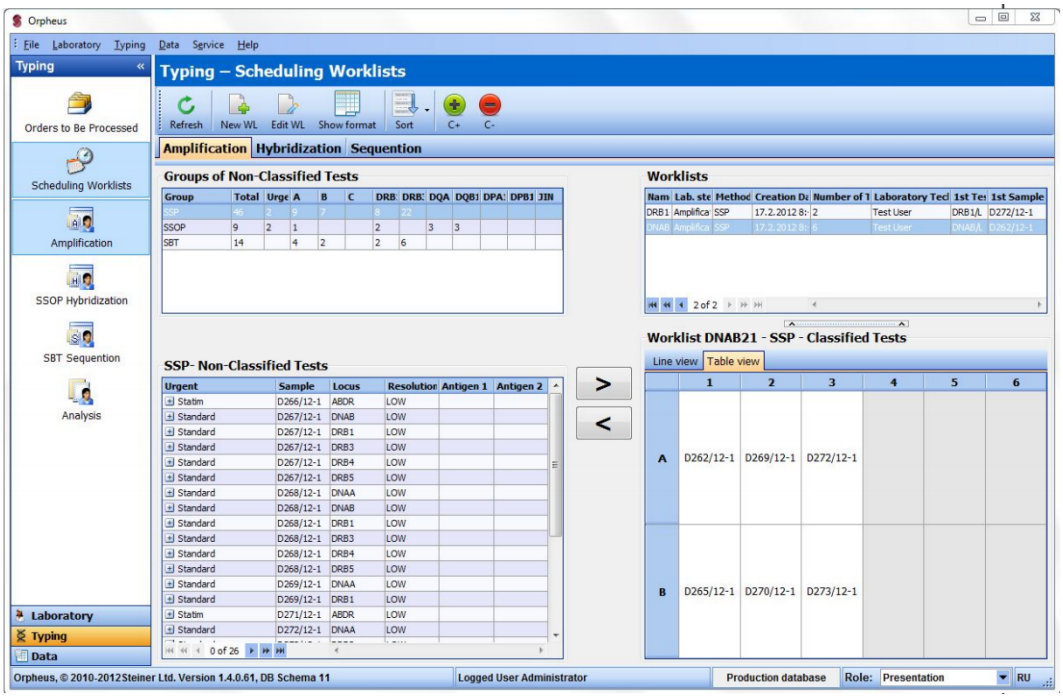
A Figura 10 apresenta uma tela da aplicação disponibilizada pela empresa.

Figura 9 - Vista parcial do *software* mTilda (única disponível no *site*)

Fonte: <http://www.mtilda.com/>

Comparativamente, o sistema desenvolvido tem muitas funcionalidades análogas as do "*Orpheus*", sendo que este possui algumas outras não incluídas por hora, como interface para cobrança, por exemplo.

Figura 10 - Tela do *software* Orpheus



Fonte: <http://www.hlasoft.com/>

4 PROPOSTA E IMPLEMENTAÇÃO DO SISTEMA CUSTOMIZADO PARA CONTROLE LABORATORIAL PARA O PROCESSO DE TIPIFICAÇÃO HLA

O sistema desenvolvido teve como objetivos sanar os riscos e vulnerabilidades encontrados no processo atual inicialmente descritos por Ribeiro [11] e confirmados durante o mapeamento *in situ* (Seção 2.3).

A aplicação permite um controle centralizado sobre o processo de tipificação HLA do HLA-UERJ e conta com um grande número de funcionalidades. Dentre as diversas disponíveis, estão presentes a possibilidade de criar usuários e atribuir controle de acesso (administrador ou não), criar análises, incluir amostras, cadastrar doadores e realizar o cadastro de hemocentros e/ ou médicos (solicitantes das análises).

Como o laboratório realiza outros serviços (não considerados nesse primeiro momento), e comumente agrega novas funcionalidades, uma outra característica identificada como importante e implementada, foi a possibilidade de expansão do sistema de forma fácil e rápida sem que fosse preciso uma grande reformulação de toda a programação. Para conseguir obter essa característica utilizou-se um conceito de engenharia de software denominado modularização, conforme já descrito anteriormente.

O sistema foi desenvolvido com um módulo central, que é um simples repositório de submódulos. Dessa forma basta que sejam seguidos um certo conjunto de regras para criação de novos módulos, que se integrarão facilmente sem alterar a dinâmica do que já está em uso.

Para que possam ser realizadas atualizações e também permitir a criação/ alteração de funcionalidades, instruções estão disponíveis nos Apêndices A, B e C. Um sumário de todas gems utilizadas pode ser visto no Apêndice E.

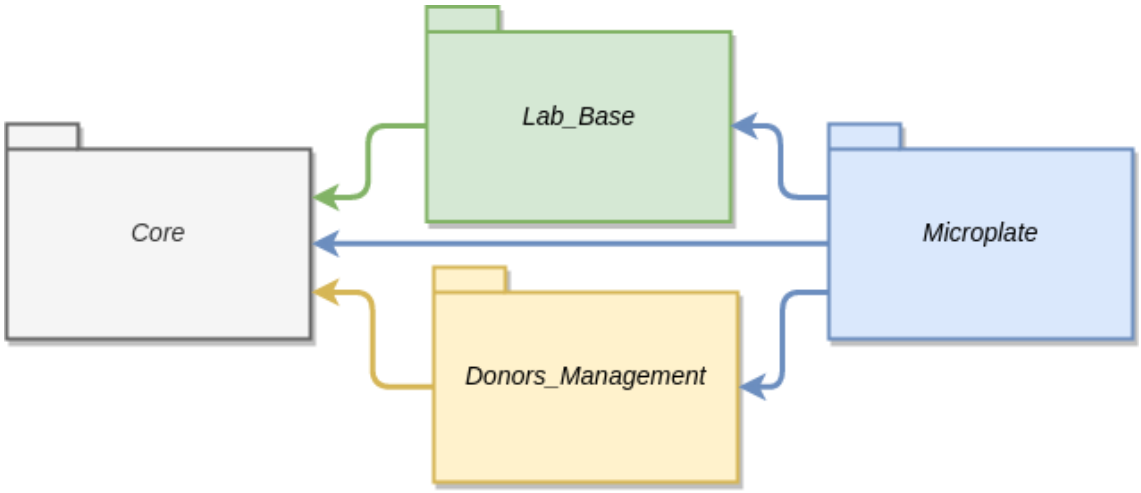
4.1 Os Módulos da Aplicação

Após extensa análise do fluxo de informações que ocorre no laboratório, e com base na ideia de modularização, optou-se por subdividir a aplicação em 4 partes (Figura 11).

O módulo nomeado *Core* é o responsável pela parte de autenticação e autorização de usuários. Como o sistema precisa lidar com informações sigilosas, como dados de pessoas e exames, é preciso restringir o acesso de pessoas não autorizadas. Esse módulo exerce justamente essa função, só permitindo a entrada de pessoas previamente aprovadas e com um *login* e senha cadastrados. Outra função desse módulo é impedir o acesso as páginas de administração por usuários comuns.

A base da interface gráfica também é toda fornecida por este módulo e teve como

Figura 11 - Módulos da Aplicação (as setas representam as dependências)



Fonte: O autor, 2018

ponto de partida um *template* gratuito [26]. Dessa forma os demais módulos só precisam implementar suas funcionalidades e copiar os padrões de interface disponíveis nesse módulo. Outra importante característica é que toda a interface foi desenvolvida de forma que seja possível acessá-la tanto de um computador, quando de um dispositivo móvel, ou seja, os diversos componentes da aplicação (por exemplo *menus* e tabelas), ajustam-se automaticamente de acordo com o tamanho da tela.

Toda rotina laboratorial foi implementada no módulo nomeado *Lab_Base*. Neste são gerenciadas a criação de tipos de análises e suas etapas e também todo o caminho percorrido pelas amostras ao longo das diversas etapas no laboratório.

Como o gerenciamento dos doadores é uma tarefa bastante extensa, e independe do fluxo das análises em si, optou-se por criar um módulo separado para este fim. No módulo *Donors_Management* é possível realizar todas as operações básicas de cadastro de pessoas, tais como consultas, alterações, inclusões, entre outras. Também foi incluída a possibilidade de realizar essas mesmas operações em lote, utilizando para isso uma planilha com o formato *csv* que pode ser aberta em *softwares* já utilizados pelo laboratório como o Excel® ou com aplicações livres como o *Calc*® do pacote *LibreOffice*®.

Uma etapa bastante importante dentro da análise, consiste na disposição das amostras em microplacas para realização de diversos procedimentos. Toda a manipulação das informações necessárias para esse fim, ficaram separadas em um módulo nomeado *Microplate*. Este depende de informações geradas em todos os outros módulos, visto que as amostras são de doadores (cadastrados no módulo *Donors_Management*) e a microplaca em si é parte de uma análise (criada no módulo *Lab_Base*). De modo a deixar esse módulo flexível, foi incluída a possibilidade de criar tipos de microplacas diferentes da do padrão utilizado pelo laboratório (com 96 poços). Desse modo é possível também criar

tipos para outros exames que necessitem de outras placas disponíveis no mercado, por exemplo com 6, 12, 24, 48, 384 ou 1536 [19][21], e até mesmo placas fora do padrão, se necessário.

Como nenhuma parte do sistema fica aberta para visualização sem autenticação (somente as pessoas cadastradas poderão interagir com a aplicação), todos os módulos dependem do módulo *Core* (cor cinza na Figura 11), que é o gerenciador dos usuários.

Com a combinação de todos os módulos, é possível acompanhar todo o processo do fluxograma apresentado na Figura 2, porém individualmente eles atuam representando diferentes etapas. Na Figura 12 é possível ver onde as ferramentas disponíveis por cada módulo se encaixam no processo. O módulo *Core* (cor cinza) gerencia o acesso aplicação assim como as permissões de cada usuário, desse modo ele faz-se presente em todas etapas. Por sua vez, o módulo *Lab_Base* (cor verde) controla o fluxo de informações de todo o processo, ditando quais etapas estão ocorrendo, quais já ocorreram e armazenando arquivos gerados ao longo do caminho, dessa forma esse modo tem atuação em toda a aplicação. As etapas que utilizam a microplaca fazem uso do módulo *Microplate* (cor azul). Essas etapas estão associadas a uma análise gerenciada pelo módulo *Lab_Base*. Por fim, é preciso associar doadores as microplacas, e todo o trabalho de cadastro e gerenciamento de informações das pessoas é feito pelo módulo *Donors_Management*.

Soluções computacionais para criação de sistemas complexos como o proposto, requerem a utilização de uma ampla gama de ferramentas para garantir que o produto final não só atenda aos requisitos funcionais necessários, como também seja facilmente atualizável e permita modificações quando necessário.

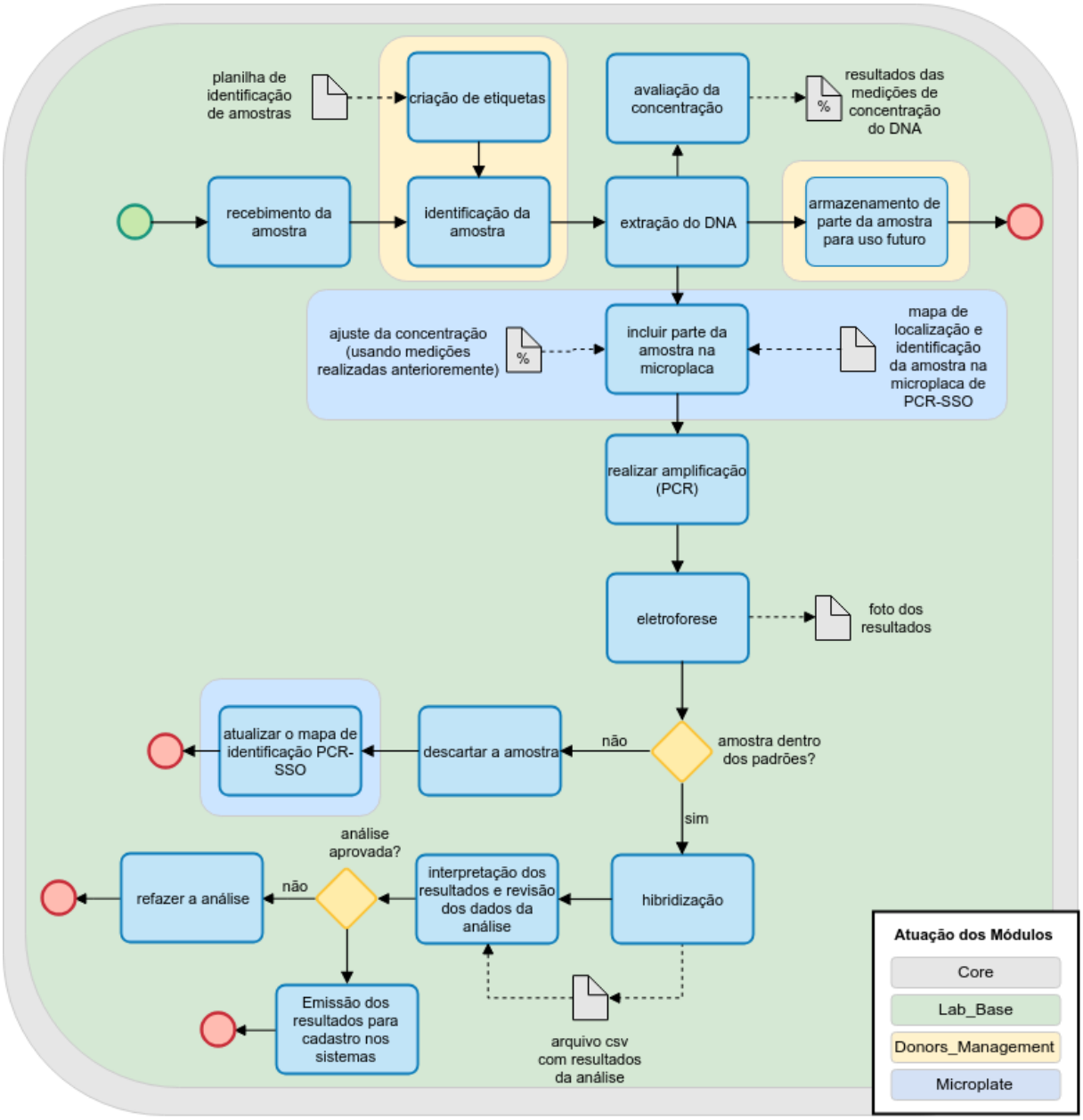
Dentre as muitas opções disponíveis para o desenvolvimento, optou-se por utilizar o *framework Ruby on Rails* e metodologias como modularização e desenvolvimento guiado por testes, com intuito de deixar o sistema completo e robusto.

4.2 **A linguagem Ruby e o Framework Rails**

A linguagem Ruby foi criada por Yukihiro "Matz" Matsumoto, e é uma mistura de várias linguagens que eram as suas favoritas (*Perl*, *Smalltalk*, *Eiffel*, *Ada* e *Lisp*). A intenção era criar uma nova linguagem que aproveitasse o melhor de dois paradigmas: a programação funcional e a programação imperativa. Yukihiro queria criar algo que fosse mais simples e natural que as linguagens até então disponíveis [27].

Lançada em 1995 de forma completamente gratuita (a licença permite a utilização, cópia, modificação e distribuição sem qualquer ônus ao usuário), sua adoção foi pouco a pouco crescendo ao longo dos anos. Em 2006 ocorreu um aumento expressivo do número de usuários por causa da enorme popularidade atingida pelo *framework Ruby on Rails* que tem como base a linguagem Ruby.

Figura 12 - Atuação de cada módulo no processo



Fonte: O autor, 2018

A Ruby é uma linguagem interpretada e por esse motivo necessita a instalação de um interpretador antes da execução de qualquer programa. Nos primórdios da linguagem, existia apenas um interpretador simples escrito em C pelo próprio Matz que tinha um gerenciador de memória limitado e não possuía características dos interpretadores mais atuais como a compilação em tempo de execução. A partir da versão 1.9, essas questões foram sanadas com um robusto interpretador baseado em uma máquina virtual com recursos mais avançados (conhecida como YARV - do inglês *Yet Another Ruby VM*).

Hoje é possível encontrar versões do interpretador para todos os sistemas operacionais. Para detalhes sobre a instalação/ configuração basta consultar o site principal da linguagem [27].

Uma importante característica é que a linguagem possui um gerenciador de pacotes bastante robusto, flexível e eficiente denominado *RubyGems* [28]. As assim chamadas *gems* são como bibliotecas reutilizáveis de código Ruby, que possuem algum tipo específico de funcionalidade, assim como qualquer outro arquivo relacionado. As *Gems* podem conter algum código nativo se necessário (em C, Java, .Net). Fazendo uma analogia com outras linguagens, as *gems* são como os *jars* no ambiente Java, ou os *assemblies* do .Net.

Essa organização em *Gems* ajuda a resolver alguns problemas bem comuns no desenvolvimento de aplicações extensas, pois fornece uma estrutura padrão para distribuição das bibliotecas e ferramentas Ruby, e também facilita o gerenciamento de dependências.

A comunidade Ruby disponibiliza um amplo repositório gratuito de bibliotecas disposta no site *RubyGems.org* [28]. Lá é possível obter mais detalhes sobre a criação, distribuição e pesquisar por *gems*.

O *Ruby on Rails* (RoR) é uma *gem* que fornece um framework livre de desenvolvimento Web escrito na linguagem de programação Ruby. A estrutura inicial foi desenvolvida por David Heinemeier Hansson e posteriormente foi ampliada pelo esforço conjunto de uma equipe de vários contribuintes. O RoR permite que sejam desenvolvidos aplicativos Web de forma rápida, minimizando as etapas, o tempo e o código envolvido na programação. O RoR teve sua primeira versão lançada em julho de 2004 e é distribuído sob a licença MIT [29].

A arquitetura utilizada pelo RoR é denominada MVC (do inglês *Model-View-Controller*), ou Modelo-Visão-Controlador. Esse padrão de arquitetura modulariza as diversas iterações do programa em entidades independentes.

O modelo é onde estão os dados da aplicação, validações, associações entre modelos e transações. Cada modelo representa (na maioria das vezes) uma tabela no banco de dados e herda de uma classe do *rails* recursos como recuperar, salvar, editar e excluir dados da tabela do banco de dados. Objetos de modelo são utilizados como uma camada entre a aplicação e o banco de dados.

O controlador é um componente que fica no servidor e que responde a solicitações externas e organiza os dados que serão enviados para modelo, bem como traduz dados

vindos de modelos em visões que serão passados a um navegador para visualização.

Já as visões são uma apresentação dos dados em um formato específico (pode ser *PDF*, *HTML*, *JSON*, entre outros). O resultado final de uma *view* provavelmente será a interface do usuário. Na grande maioria das páginas na *web*, essas *views* terão formato *HTML* com folhas de estilo em *CSS* e talvez algum *script* em *javascript*.

O *Rails* tem como fundamento dois princípios importantes que ajudam no desenvolvimento de uma plataforma robusta que são o DRY - *Dont't Repeat Yourself* e o CoC - *Convention over Configuration*.

O conceito de DRY é um princípio que visa a redução de repetição de código, estimulando a substituição destes trechos repetidos por abstrações. O conceito de CoC auxilia o desenvolvedor, reduzindo o número de pequenas configurações/ decisões que seriam necessárias já pré-configurando o *framework*. Assim o desenvolvedor só precisa especificar algo caso saia do padrão.

A partir da versão 3.1 do *Rails* foi criado um conceito para facilitar a extensão de uma aplicação chamado *Rails Engine*. Esse conceito é extensamente utilizado no presente trabalho e consiste em uma mini aplicação *Rails* cujas funcionalidades estão isoladas e podem ser reaproveitadas em qualquer outra aplicação Rails de uma forma estruturada. Uma vez finalizado o desenvolvimento, é possível exportar essas aplicações na forma de *gems*.

As engines podem ser adicionadas a uma aplicação por meio de um arquivo de configuração padrão denominado *Gemfile*, que é criado com a aplicação Rails e é utilizado para descrever todas as *gems* e suas versões, e irão então fazer parte da aplicação global adicionando funcionalidades.

4.3 Por que *Ruby on Rails?*

A escolha do *framework* RoR não foi por acaso. Atualmente existem uma gama muito grande de possibilidades que realizam tarefas similares utilizando as mais diversas abordagens. Segue uma compilação de fatos obtida de uma grande gama de artigos de desenvolvedores na *internet* [30] [31] [32] [33] [34].

O *Ruby on Rails* é uma ferramenta madura e já está no mercado há mais de 10 anos. Mesmo com o passar dos anos a equipe de desenvolvimento se mantêm sólida e lançando novas atualizações frequentemente. A prova disso é que grandes empresas adotaram o *framework* no desenvolvimento de suas aplicações. Podemos citar como exemplo, empresas como *GitHub*, *Airbnb*, *Twitch*, *Basecamp*, *Kickstarter* e muitas outras dos mais diversos setores.

O *Ruby on Rails* é uma ferramenta completa (*full-stack*). Ao contrário da maioria das outras linguagens, o *RoR* oferece suporte tanto para o desenvolvimento de

front-end (parte visível da aplicação), quanto para a parte de *back-end* (parte do servidor). Sendo assim um único desenvolvedor pode construir uma aplicação completa sem precisar depender de outra pessoas para construir, ou o *back-end*, ou o *front-end* para ele. No caminho provavelmente ainda vai aprender um pouco de *javascript*, *HTML/CSS* e *Ruby*.

O *Ruby on Rails* possui uma das comunidades mais ativas dentre as linguagens. Por causa da ampla utilização em diversos nichos, é muito fácil encontrar pessoas dispostas a ajudar. São milhares de fóruns, conferências no mundo todo e uma gama incontável de tutoriais na internet. É muito pouco provável que alguma dúvida fique sem resposta. Isso é excelente para iniciantes.

O *Ruby on Rails* "esconde" muita coisa que o desenvolvedor não precisa saber. Por exemplo, não é necessário conhecer SQL para utilizar um banco de dados em uma aplicação criada com o *framework*. O RoR simplesmente assume que o desenvolvedor quer seguir um conjunto comum de configurações a menos que seja especificado o contrário. Esse fato é ótimo para iniciantes que não precisam conhecer a fundo todos os detalhes e customizações necessárias da aplicação para poder começar. Isso vale também para o lado da segurança, sem que o desenvolvedor faça nenhuma configuração, alguns casos típicos de falhas já exploradas são tratadas por padrão, como *sql-injection*, *cross-site scription*, *session hijacking*, entre outros.

O *Ruby on Rails* "força" a utilização de boas práticas. Por causa das suposições feitas pelo *framework*, parte do já citado lema *Convenction Over Configuration*, os desenvolvedores seguem sempre um padrão de desenvolvimento que estimula o uso de boas práticas de desenvolvimento. Como a estrutura de diretórios e arquivos é bastante similar entre aplicações, é muito simples que desenvolvedores consigam entender sistemas que não foram criados por eles. A instalação básica de uma aplicação já conta até com uma estrutura para realização de testes.

O *Ruby on Rails* tem uma biblioteca muito extensa de ferramentas e funções. Existem *gems* para praticamente qualquer funcionalidade. Basta procurar no repositório *Rubygems* [28] que provavelmente irá ter algo que atenda as necessidades do desenvolvedor. Mesmo que não exista algo, é muito pouco provável que não seja encontrado algo próximo, assim evitando que o trabalho seja iniciado do zero.

E por último, mas não menos importante, **o *Ruby on Rail* é divertido de ser utilizado**. A combinação de fatores citados, faz com que a ferramenta seja prazerosa de se trabalhar. Parece algo fútil mas se levarmos em consideração que em muitos casos os desenvolvedores ficam trabalhando por cerca de 40 horas por semana e por anos no desenvolvimento de um sistema, ter uma ferramenta que causa tranquilidade faz muita diferença.

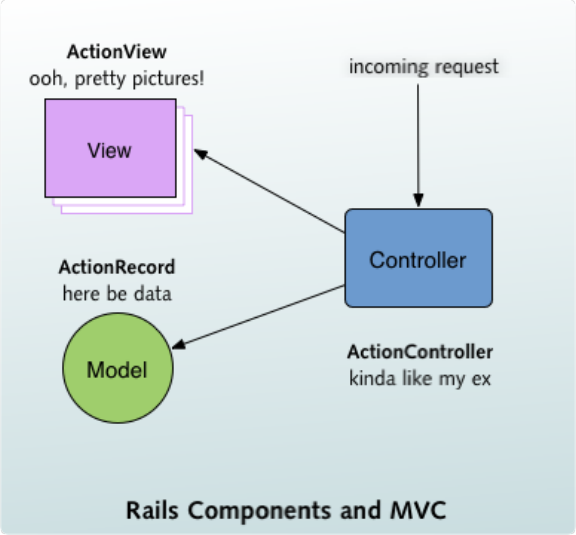
4.4 **Modularidade**

O conceito de modularidade é uma especialização de um importante princípio da engenharia de software, que é o da Separação de Conceitos - SoC (do inglês *Separation of Concerns*).

O princípio da SoC é um conceito que sugere que qualquer problema complexo pode ser resolvido de forma mais eficiente se forem identificadas a priori os diferentes tipos de conceitos envolvidos, e então usar esse conhecimento para subdividi-lo em pedaços que possam ser resolvidos e/ ou otimizados independentemente.

Podemos considerar como "conceito" uma funcionalidade, uma certa responsabilidade ou qualquer ponto de foco do problemas que possa ser agrupado para formar uma unidade. Em uma escala mais ampla podemos usar o modelo MVC (*Model-View-Controller*)(figura 13) utilizado pelo *Rails* como exemplo, que separa os conceitos em três camadas com distintas funcionalidades. São elas a camada de apresentação (*View*), a de lógica (*Controller*) e a de dados (*Model*). Se pensarmos em escalas menores, os componentes podem ser as classes ou métodos.

Figura 13 - *Model-View-Controller* no Ruby on Rails



Fonte: http://rubyonrails.org/

A utilização do princípio da separação de conceitos traz consigo inúmeras vantagens. O particionamento do *software* torna a manutenção mais fácil pois permite com que seja mais simples garantir a falta de duplicação e singularidade de propósito das partes individuais. Dessa forma o sistema também tende a ser mais estável, subproduto da facilidade de manutenção. A dissociação das funcionalidades leva a criação de componentes especialistas, o que leva a componentes que são mais facilmente reutilizáveis em outros sistemas, ou até mesmo em outros contextos dentro do mesmo sistema [35].

Em um ambiente laboratorial, a utilização desse conceito permite a criação de aplicativos flexíveis que possam se adaptar a realidade dos procedimentos utilizados naquele local (as alterações necessárias serão modulares e não globais), bem como a inclusão de novas funcionalidades (novos módulos) caso o laboratório comece a realizar outros tipos de exames.

4.5 **Desenvolvimento Guiado por Testes**

De forma resumida, podemos dizer que o desenvolvimento guiado por teste (TDD) (do inglês *Test-Driven Development*) é uma técnica de desenvolvimento de *software* que tem como premissa base a criação de testes antes do código principal, i.e., antes de qualquer mudança no código de produção o programador deve ter escrito testes falhos. O passo seguinte consiste na escrita do mínimo de código funcional para que o teste seja atendido e no fim realizar uma etapa de refatoração, com intuito de eliminar qualquer duplicata no código [36].

Embora o nome sugira que o TDD seja um método de teste, um estudo mais aprofundado do tema nos mostra que na verdade o assunto é um pouco mais complexo.

Além de realizar os testes propriamente ditos, o TDD envolve a escrita de testes automáticos das unidades individuais de um programa. Uma unidade individual pode ser entendida como o menor componente de *software* possível de ser testado. Não existe um consenso sobre o que exatamente constitui uma "unidade individual de software". Mesmo no mundo da programação orientada a objetos, tanto a classe quanto os métodos são sugestões de possíveis unidades individuais apropriadas. De um modo geral, entretanto, o método acaba sendo o menor componente de software passível de ser testado [37].

A execução dos testes pode ser um processo manual ou automático e pode ser realizada pelos próprios desenvolvedores ou por pessoas designadas especificamente para esse fim. A atividade de testes envolve a escrita de códigos de teste unitários e a colocação destes em ambientes próprios para esse fim como *JUnit* no caso do Java ou *RSpec* no caso do *Rails*. Esses *frameworks* de testes diminuem consideravelmente o esforço requerido para executar o teste pois muitas vezes permitem que se executem todos eles com um simples comando ou apenas apertando um botão. Em contraste, a execução de testes manuais deve ser feita individualmente e o esforço será então proporcional ao número de testes especificados.

Tradicionalmente, os testes unitários eram executados após o desenvolvimento. Essa tarefa poderia levar desde alguns minutos até mesmo meses. Os testes poderiam ser escritos pelo próprio programador ou por um profissional designado somente para esse fim. Com o TDD, o programador escreve os testes antes do código que será testado. Como resultado, o programador pode executar os testes assim que eles forem escritos [36].

A técnica já provou ser bastante efetiva quando aplicada em empresas desenvolvedoras de *softwares* comerciais. Na prática o TDD faz com os desenvolvedores reflitam melhor em como o sistema deve ser organizado [38].

Em resumo o TDD poder ser descrito com as premissas abaixo, segundo Martin, mais conhecido no meio por "*Uncle Bob*" [39]:

- Não é permitida a escrita de qualquer código de produção a menos que já tenha sido escrito um teste que falhou para ele;
- Um teste deve ter código somente o suficiente para falhar, e falhas de compilação são falhas;
- Não é permitido escrever mais código de produção além do suficiente para que o teste passe.

Para o desenvolvimento de sistemas complexos, onde existem dependências entre as partes, a atualização de uma funcionalidade pode vir a criar problemas em outras partes da aplicação. Na aplicação desenvolvida foram escritos mais de 100 testes de modo a garantir que mudanças futuras não afetem o funcionamento dos módulos já feitos.

O *Ruby* tem uma ampla gama de *gems* que facilitam muito a criação e execução dos testes individualmente ou em lote. É possível testar desde componentes internos como modelos e controladores, até aspectos de interação do usuário com as interfaces gráficas, via um simulador de *browser*. Assim é possível inclusive verificar o preenchimento de formulários e suas consequências após o usuário o submeter, verificar também a existência de elementos importantes em *views* e até mesmo testar acessos a páginas restritas, etc.

Detalhes técnicos sobre as *gems* utilizadas podem ser encontrados no Apêndice B.5. Exemplos de testes escritos utilizando as ferramentas disponibilizadas pelas bibliotecas podem ser vistos no Apêndice D.

4.6 **Descritivos dos Módulos da Aplicação**

A seguir são apresentados os módulos em detalhes. Como o intuito não é criar um "tutorial" da linguagem *rails*, não são apresentados todos detalhes técnicos de construção quando estes puderem ser realizados por alguma outra fonte (por exemplo, as páginas das *gems* são ricas em instruções de como instalá-las, realizar configurações básicas e manipulações de dados diversos na aplicação). Nesse caso serão indicadas fontes de consulta para que o leitor possa se informar. Serão discutidas as configurações e decisões consideradas importantes para o desenvolvimento básico e que afetam a aplicação como um todo.

Para estudo sobre o básico de *Ruby on Rails*, recomenda-se o livro gratuito de Michael Hartl [40].

4.6.1 O banco de dados

O termo banco de dados (BD) é usado para referir-se a uma coleção ou respositório de dados que possui um certa estrutura. Os dados podem ser organizados de forma tabular ou não ter uma estrutura definida. Para gerenciar os bancos de dados são utilizados aplicativos especialmente projetados para esse fim, conhecidos como DBMS (do inglês *Database Management System*) [41]. Esse tipo de *software* permite que os dados sejam gerenciados, recuperados e armazenados de forma otimizada e organizada, além de ser o responsável por garantir a integridade do sistema.

De um modo geral podemos separar os tipos de banco de dados atuais em duas categorias, que são os relacionais e os não relacionais. Os bancos de dados relacionais são baseados em tabelas e possuem uma estrutura pré-definida enquanto os não-relacionais são baseados em pares valores-chave e possuem estrutura dinâmica, i.e., podem ser alteradas mais facilmente. Pelo fato de ter uma estrutura fixa, as buscas mais complexas nos BD do tipo relacional são mais eficientes, sendo portanto mais indicados quando o volume de dados a ser armazenado é muito grande.

Na presente aplicação, temos estruturas de dados bem definidas, como por exemplo os dados dos usuários e doadores podem ser facilmente implementados como um tabela com campos fixos como nome, sobrenome, endereço, etc. Além disso as relações entre dados também são bem estabelecidas (por exemplo, cada análise tem um responsável, cada microplaca está relacionada com um certo número de poços, sendo que cada poço por sua vez contém o exame de um doador, etc), fazendo com que um BD do tipo relacional seja o mais indicado.

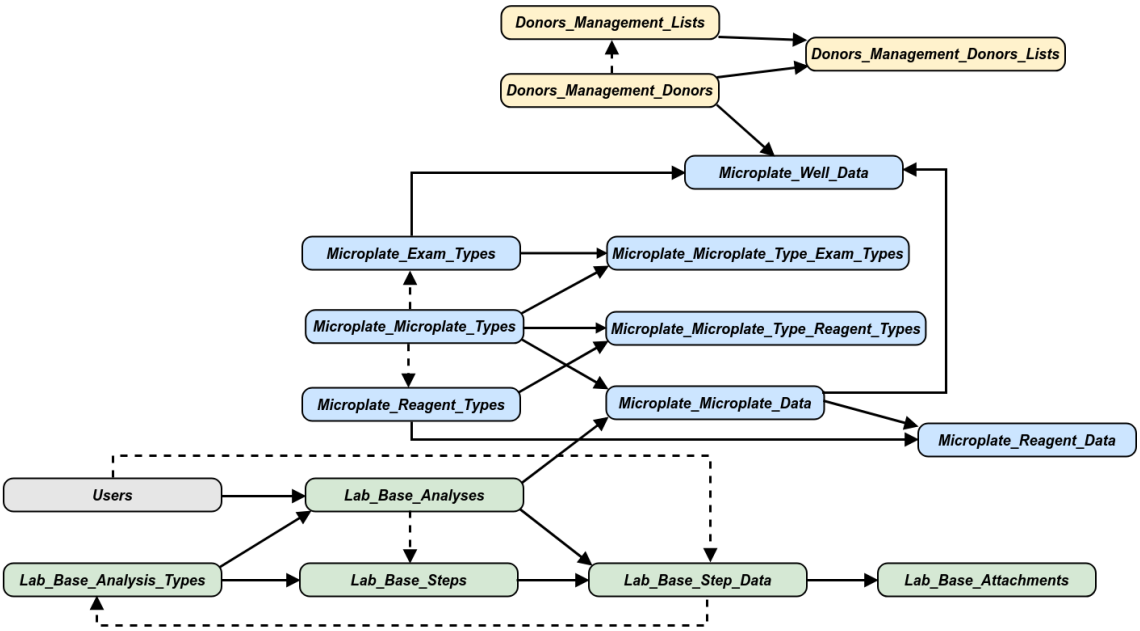
Dos diversos DBMS disponíveis, no sistema apresentado foi utilizado o *PostgreSQL*. Esse tipo de banco de dados é considerado um dos mais robustos e avançados disponíveis atualmente, além de ser *open source* [41] [42].

O *RoR* conta nativamente com abstrações para utilização desse banco de dados, fazendo com que o desenvolvedor não precise se preocupar com os detalhes do funcionamento, ficando livre para se preocupar somente com a organização das informações que precisa salvar.

4.6.2 Relações entre as tabelas do banco de dados

As relações entre as tabelas do banco de dados podem ser vistas graficamente na Figura 14. No gráfico, as linhas simples representam associações do tipo um-para-muitos e as muitos-para-muitos que usam uma tabela de junção são representadas por setas pontilhadas. Mais informações sobre associações pode ser visto no RailsGuides [43], na seção *Active Record Associations*.

Figura 14 - Relações entre as tabelas



Fonte: O autor, 2018

4.6.3 Módulo *Core*

O primeiro módulo tem como intuito gerenciar o acesso ao sistema. Para a aplicação, existem dois tipos distintos de usuários: comum e administrador. O usuário administrador tem acesso a todas as funcionalidades do sistema, enquanto o usuário comum poderá somente acessar dados de análises.

Para controle do tipo de usuário, foi adicionado um campo denominado *role* que utiliza um tipo de dado denominado *enumerate*. Esse tipo habilita uma variável que pode ser configurada com constantes pré-definidas.

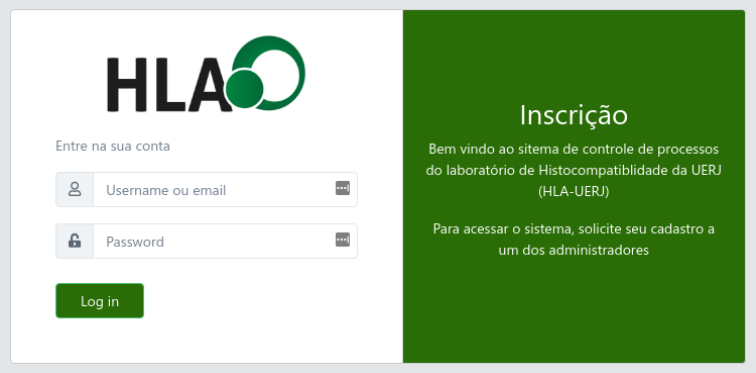
Segue a declaração da variável feita no modelo *user.rb*.

```
1 enum role: [:user, :admin]
```

Para a parte de autenticação foi utilizada uma *gem* denominada *Devise* [44]. Essa *gem* fornece um robusto controle de acesso, já provendo todo o código e um *template* das telas necessárias para acesso de usuário. A segurança é garantida por um algoritmo de criptografia (*bcrypt* [45]), que cria um *hash* das senhas antes de armazenar no banco de dados. Caso um usuário comum esqueça sua senha, deverá solicitar a outro que tenha permissões de administrador para criar uma nova. A Figura 15 apresenta a tela de *login*.

Após a instalação do módulo, é preciso cadastrar um administrador para acessar o sistema e dar início ao resto da configuração. Isso deve ser feito no *console* do *Rails*. Basta acessar via comando a tabela do banco de dados relativa aos usuários e cadastrar diretamente essa primeira entrada, conforme abaixo:

Figura 15 - Tela de login



Fonte: O autor, 2018

```
rails c
# console do Rails
2.5.1 :001> Dat::User.create(email: "admin@temp.com", username: "admin",
   password: "123456", password_confirmation: "123456", role: :admin)
# para sair do console: ctrl+d
```

Recomenda-se fortemente que, depois de feita toda a configuração, seja criado um outro usuário administrador já com informações de um funcionário e este apagado também via console.

Outro aspecto importante é que a aplicação foi configurada de modo que não seja possível apagar completamente registros importantes. Essa abordagem tem como intenção evitar que informações sejam perdidas acidentalmente e para manter um registro de todas as operações realizadas. Para obter tal comportamento foi utilizada uma *gem* nomeada *Discard* [46].

Para o funcionamento dessa *gem* é preciso criar um campo na tabela cujos registros não podem ser apagados, denominado *discarded_at*. Esse campo é do tipo *datetime* e será preenchido com a data/hora que o registro foi apagado. Utilizando métodos providos pela biblioteca é possível filtrar os registros que estão ativos dos que estão arquivados.

O usuário com acesso de administrador tem acesso a uma área especial da aplicação denominada “Arquivo” onde poderá visualizar todos os registros que foram arquivados e recuperá-los se assim desejar. Na Figura 16 é mostrada uma tela com um usuário administrador logado. Nela é possível ver a seção “Arquivos” na barra lateral com um link para a página que mostrará os usuários arquivados. Mais detalhes sobre o funcionamento e configurações pode ser visto nas instruções que se encontram na página da *gem* [46].

Também é preciso limitar o acesso não autorizado as diversas páginas da aplicação. Se nenhuma configuração for feita, um usuário comum poderá acessar as páginas de administração se souber o endereço das mesmas. Para limitar esse acesso é utilizada uma *gem* denominada *CanCanCan* [47]. O funcionamento normal dessa *gem* consiste em

Figura 16 - Tela após ao login - Visão do administrador (somente o módulo *Core* ativo)

Fonte: O autor, 2018

configurar as permissões de acesso em um arquivo denominado *abilities.rb* e indicar nos controladores que eles devem, antes de qualquer operação, verificar as permissões nesse arquivo.

Da forma como a *gem* funciona, a cada novo módulo incluído, o desenvolvedor seria obrigado a editar o módulo *Core* para incluir as permissões que fossem necessárias no arquivo *abilities.rb*. O que viola o princípio de separação de conceitos e obrigaria a uma revisão desse módulo sempre que um novo fosse criado.

Felizmente é possível modificar esse arquivo, criando alguns métodos dentro da classe, que permitem a adição de novas permissões oriundas de outros módulos. Para isso cria-se um parâmetro de classe (*:abilities*) e métodos que permitem acesso a esse atributo (*register_ability* e o *remove_ability*). Quando o objeto *user* é inicializado, o código de *loop* irá percorrer o conteúdo de todas os atributos *abilities* e instanciar cada elemento passando o objecto *user*. Em seguida é possível obter as regras (utilizando um método disponível na *gem*) e adicionar as permissões ao arquivo de *ability* principal.

Em seguida, nos demais módulos, basta ao criar esse arquivo *ability.rb* utilizar o método *register_ability* para incluir as permissões no arquivo principal [13].

Segue o código utilizado no módulo *Core* implementando o exposto. As instruções para incluir as permissões nos outros módulos, de modo que elas se juntem ao arquivo principal, encontram-se no Apêndice C.

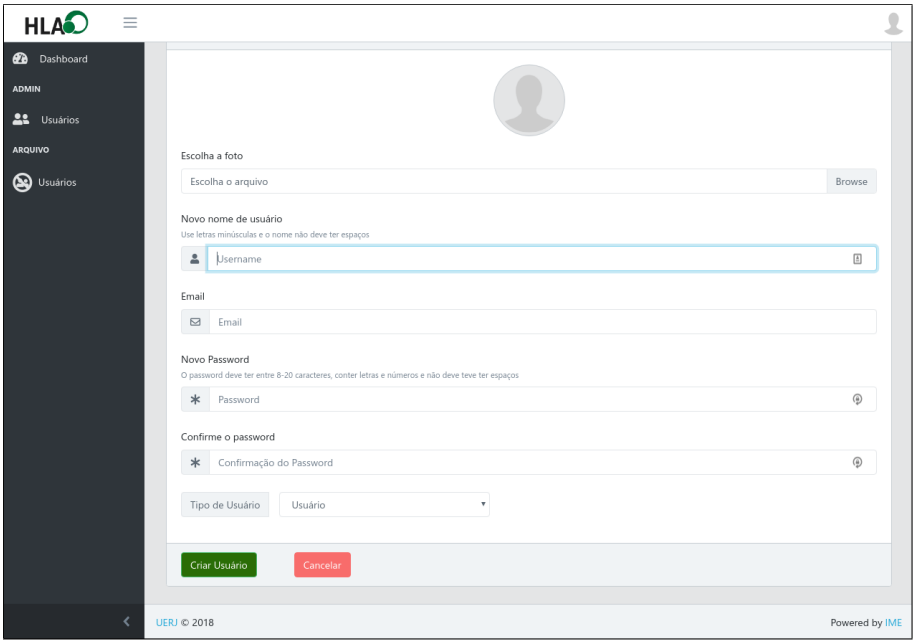
```ruby
module Dat
  class Ability
    include CanCan::Ability

    class_attribute :abilities
    self.abilities = Set.new

    def self.register_ability(ability)
      self.abilities.add(ability)
    end

    def self.remove_ability(ability)
      self.abilities.delete(ability)
    end

    def initialize(user)
      Ability.abilities.each do |klass|
        ability = klass.send(:new, user)
        merge(ability)
      end

      if user.admin?
        can :manage, :all
      else
        can :read, :dashboard
        can :manage, Dat::User, id: user.id
      end
    end
  end
end
```

Para finalizar, são apresentadas 3 telas importante do módulo.

A Figura 17 é a tela de adição de usuários, nesta pode ser visto o formulário com as informações básicas necessárias para o cadastro. A Figura 18 apresenta a tela de gerenciamento de usuários. Nela é possível ver uma lista com todos os usuários cadastrados ativos e algumas informações gerais como o "*user name*" (nome único no sistema usado para identificação) do usuário e se ele é cadastrado como administrador ou não. Somente usuários administradores tem acesso a essa parte da aplicação.

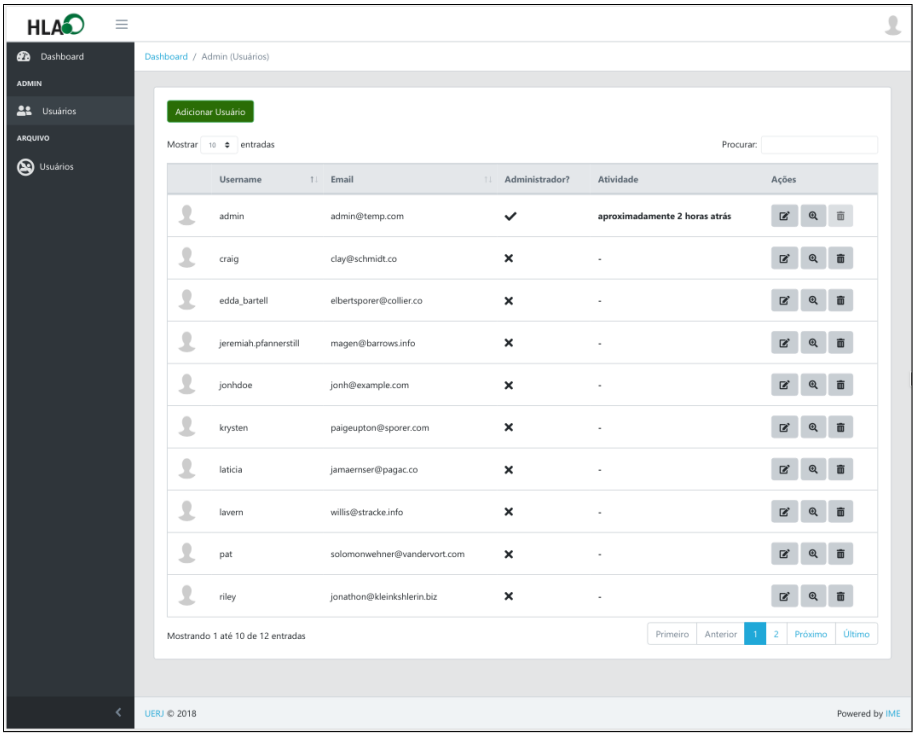
Na Figura 19 é apresentada a tela de edição de perfil. Todos os usuários tem acesso a essa tela. Nela é possível trocar a senha e incluir uma foto como avatar (identidade virtual do usuário).

Figura 17 - Tela de adição de usuários (somente os administradores tem acesso)



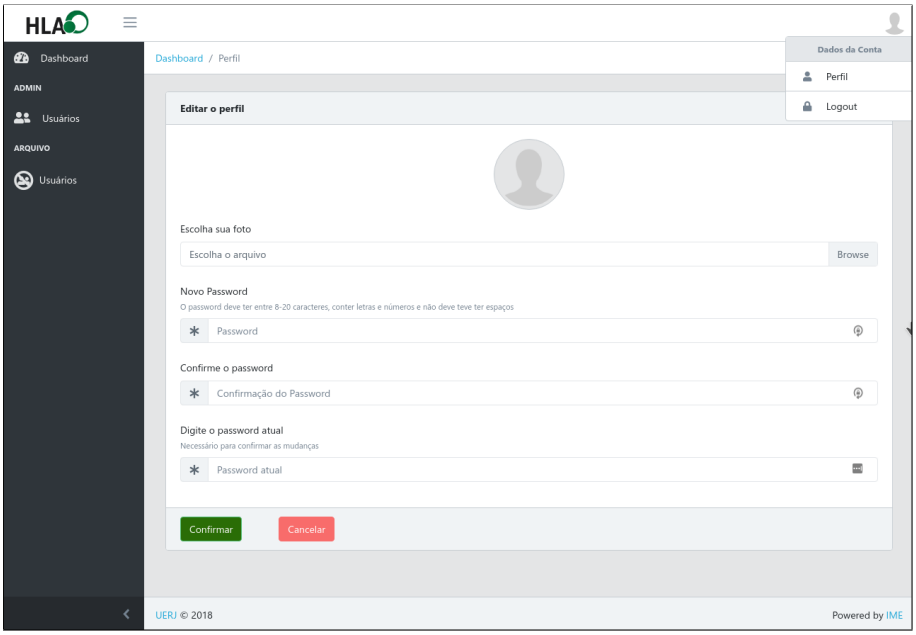
Fonte: O autor, 2018

Figura 18 - Tela de gerenciamento de usuários (somente os administradores tem acesso)



Fonte: O autor, 2018

Figura 19 - Tela de edição do perfil



Fonte: O autor, 2018

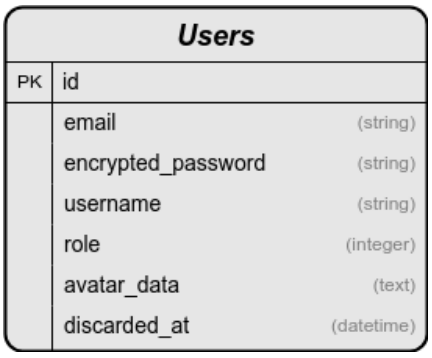
4.6.4 Tabela do modelo *User*

A Figura 20 mostra as principais entradas da tabela *Users*. A *gem Devise* já cria, em seu processo de configuração, uma tabela básica com diversos campos para controle de acesso. Alguns outros campos foram adicionados com intuito de prover funcionalidades extras.

Segue um descritivo dos principais campos da tabela *Users*.

- ***id***: campo de chave primária (*PK*) de auto incremento. Gerada automaticamente na criação do modelo,
- ***encrypted_password***: campo gerado automaticamente pela *Gem Devise*. Quarda o *hash* da senha do usuário,
- ***username***: campo para o apelido do usuário,
- ***role***: campo para selecionar o tipo de usuário,
- ***avatar_data***: campo que guarda o caminho relativo de um arquivo de imagem (foto) que o usuário pode usar para identificar seu perfil,
- ***discarded_at***: campo que irá guardar a data/hora quando um registro for arquivado. Caso contrário será igual a *NULL*.

Figura 20 - Tabela do banco de dados - *Users*



Fonte: O autor, 2018

4.6.5 Módulo *Lab_Base*

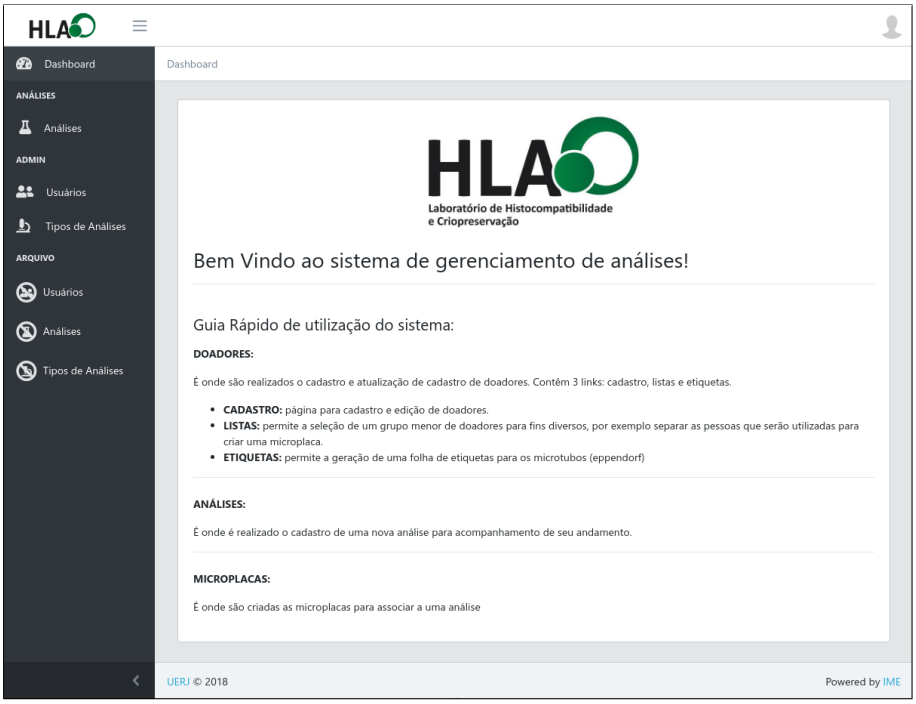
Esse módulo visa controlar toda a vida da análise ao longo dos processos realizados no laboratório. A Figura 21 apresenta a tela do administrador após o *login*, nela é possível ver, no menu lateral, as opções na área do administrador para cadastro de tipos de análises e também a área de arquivo para visualizar os tipos de análises e as análises arquivadas.

Antes de começar qualquer tipo de análise, é necessário que o tipo dela seja criado. Essa tarefa cabe a pessoas com a função de administrador, visto que é uma tarefa que não será realizada com frequência e não deve se alterar ao longo da vida da aplicação. Cada análise tem as suas respectivas etapas, que precisam ser incluídas também.

De acordo com o mapeamento realizado das atividades de tipificação HLA, uma sugestão seria o seguinte esquema:

- Tipo de Análise: “Tipagem HLA - Média Resolução”
  - **Fase 1:** “Cadastro das amostras”
  - **Fase 2:** “Extração de DNA”
  - **Fase 3:** “Criação de Mapa de Identificação PCR”
  - **Fase 4:** “Amplificação”
  - **Fase 5:** “Verificação do Padrão das Amostras”
  - **Fase 6:** “Atualização do Mapa de Identificação”
  - **Fase 7:** “Hibridização”
  - **Fase 8:** “Inclusão dos Dados nos Sistemas”
  - **Fase 9:** “Emissão de Laudo”

Figura 21 - Tela após login - Visão do administrador (módulos *Core* e *Lab_Base* ativos)



Fonte: O autor, 2018

Para o cadastro de um tipo novo são necessários dois parâmetros: o nome e uma breve descrição. Com relação as etapas é preciso indicar o nome, uma breve descrição e também a ordem das etapas. A Figura 22 apresenta a visualização de uma lista com todos os tipos de análise cadastrados (somente administradores tem acesso a essa parte da aplicação). A Figura 23 apresenta a tela onde é possível cadastrar os tipos de análises e suas respectivas etapas.

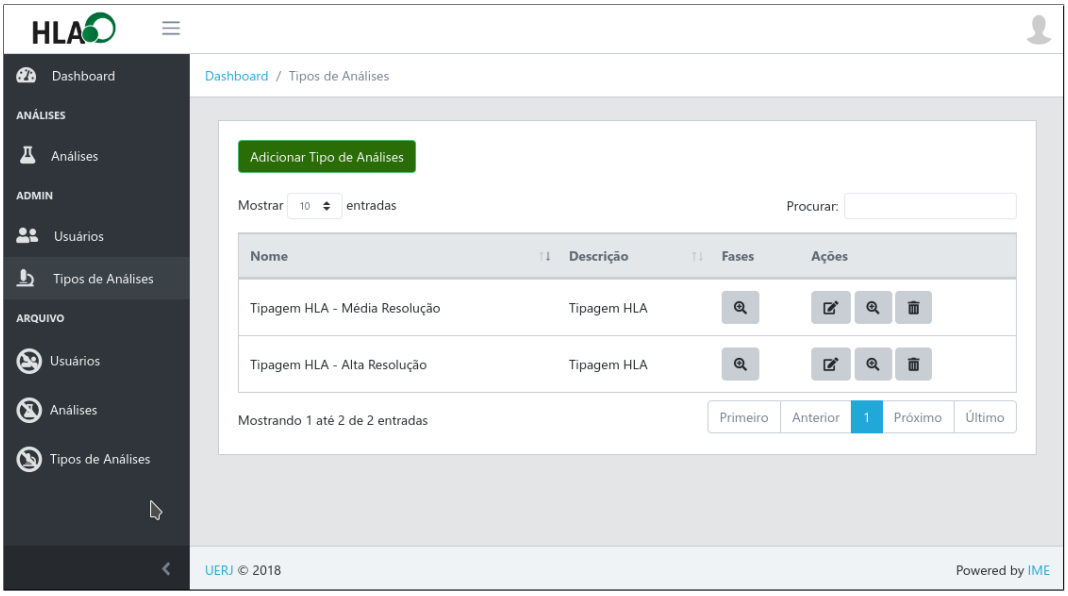
Findada esse etapa, é possível então iniciar uma análise em si. A Figura 24 apresenta a visualização de tela que mostra todas as análises não arquivadas. Escolheu-se um código de cor para facilitar a visualização do andamento de cada uma. A legenda aparece no menu lateral logo que o usuário entra nessa tela. É possível também filtrá-las por andamento ou ordenar por data de criação, código e tipo de análise. Essa área já é comum para qualquer tipo de usuário.

Assim como no módulo *Core*, esse código de cores é guardado em uma variável do tipo *enumerate* com as seguintes constantes pré-definidas (arquivo *analysis.rb*):

```
1 enum role: [:created, :on_going, :finished, :suspended, :cancelled]
```

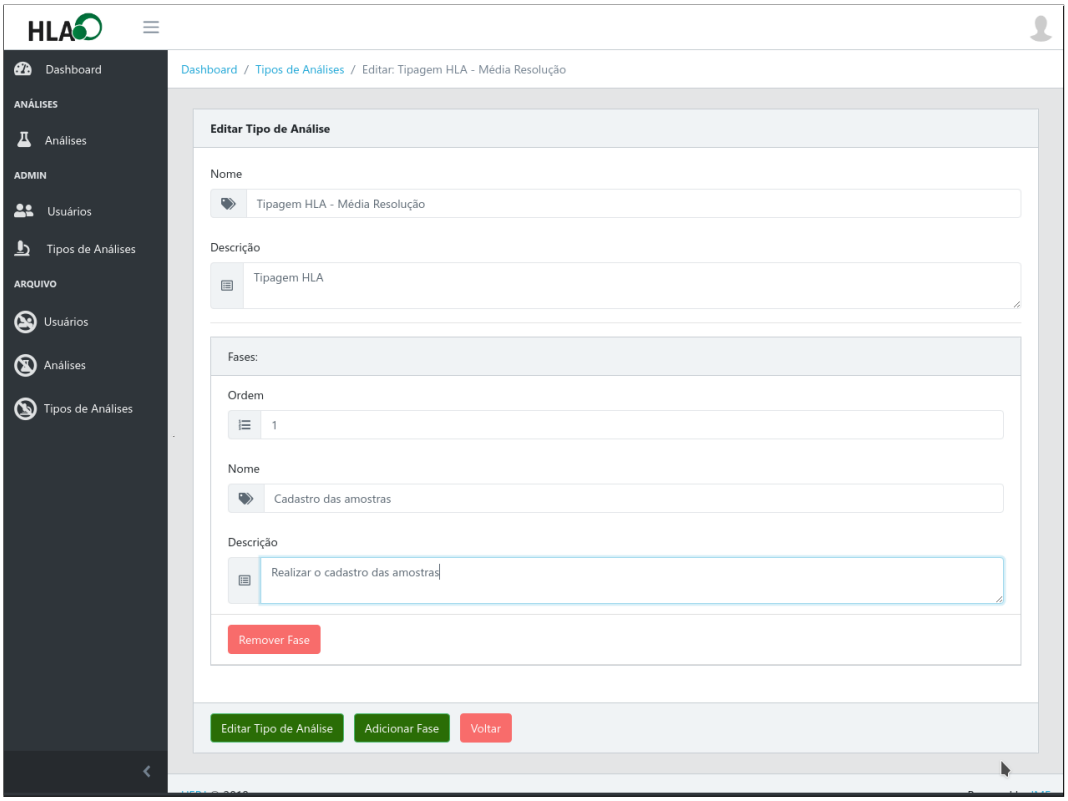
A Figura 25 apresenta a tela para criação de uma análise. É preciso entrar com o código, descrição, e selecionar o tipo de análise (o campo de observações é opcional).

Figura 22 - Tela que mostra todas os tipos de análises cadastradas (somente os administradores tem acesso)



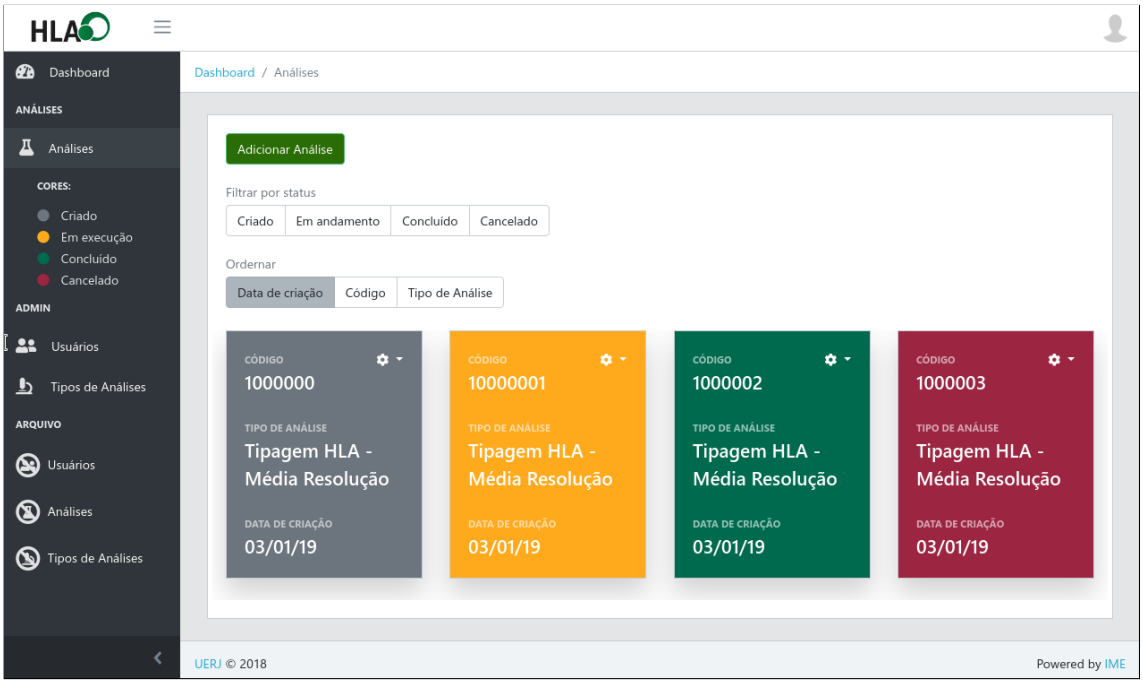
Fonte: O autor, 2018

Figura 23 - Tela que mostra o cadastro de um tipo de análise e suas fases (somente os administradores tem acesso)



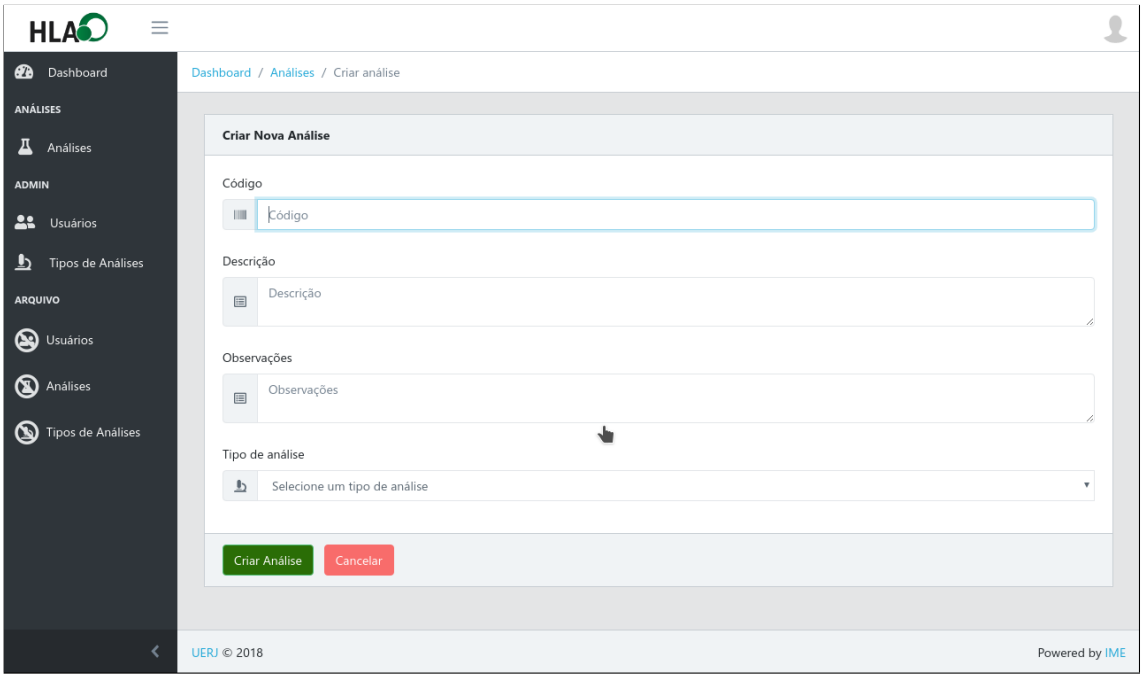
Fonte: O autor, 2018

Figura 24 - Tela que mostra todas as análises não arquivadas



Fonte: O autor, 2018

Figura 25 - Tela que mostra o cadastro de uma análise

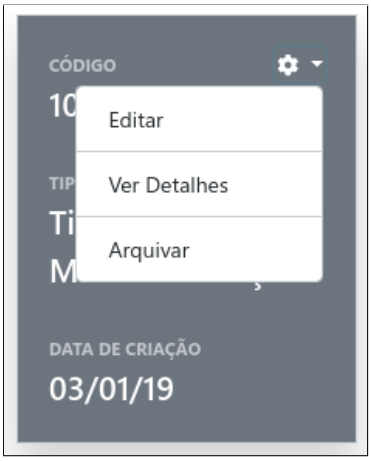


Fonte: O autor, 2018

Após essa etapa a análise criada irá aparecer na tela de análises. No servidor é criada um registro para a nova análise na tabela *Lab_Base_Analyses* mas também são criadas automaticamente entradas para salvar as informações para cada etapa referente ao tipo de análise escolhido (Na tabela *Lab_Base_Step_Data*).

Para visualizar as etapas e o andamento das mesmas basta clicar no ícone em formato de engrenagem no canto superior direito do bloco representativo da análise, e depois em "ver detalhes" no menu que aparecerá (Figura 26).

Figura 26 - Tela que mostra o menu para ver detalhes da análise



Fonte: O autor, 2018

A tela resultado dessa ação é a apresentada na Figura 27. Novamente é utilizado um código de cores para indicar o andamento de cada etapa, que é igual aos das análises apresentado anteriormente.

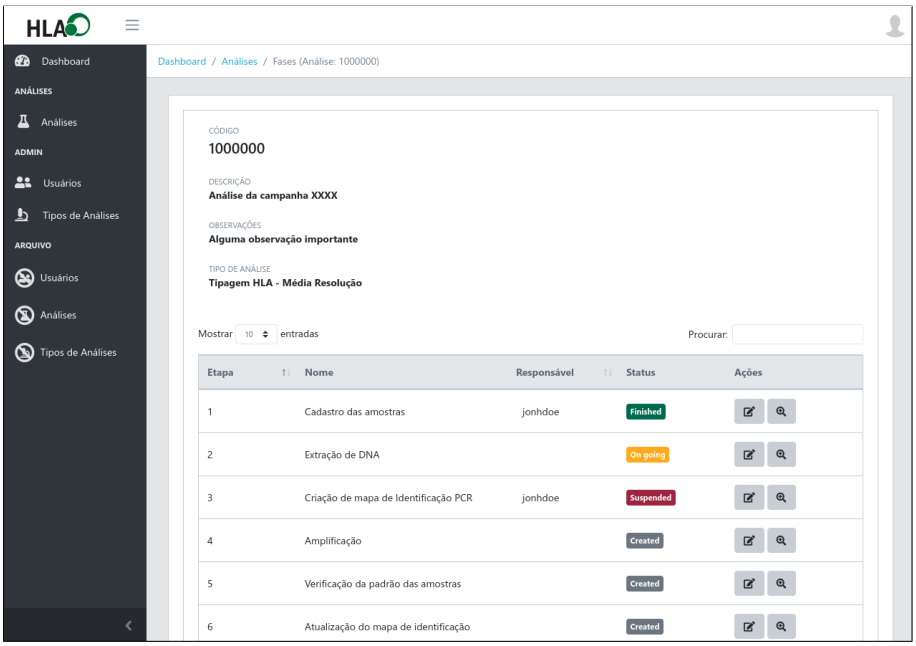
Para editar cada passo basta clicar no ícone correspondente a edição na coluna "Ações". A tela que aparecerá é a apresentada na Figura 28. Nesta tela é possível fazer observações, anexar um ou mais arquivos de qualquer formato e mudar o estado da fase.

Quando o estado da fase é mudado para algum ponto terminal (finalizado, suspenso ou cancelado), o nome de usuário da pessoa que realizou tal mudança ficará indicado e a etapa ficará bloqueada para edições, podendo ser desbloqueada somente por uma usuário administrador. Essa marcação visa identificar o usuário responsável pela realização da etapa (coluna "Responsável" da tabela apresentada na Figura 27).

Quando o usuário edita algumas desses campos e salva, é criado uma entrada na tabela *Lab_Base_Attachments* associada a essa etapa com as informações.

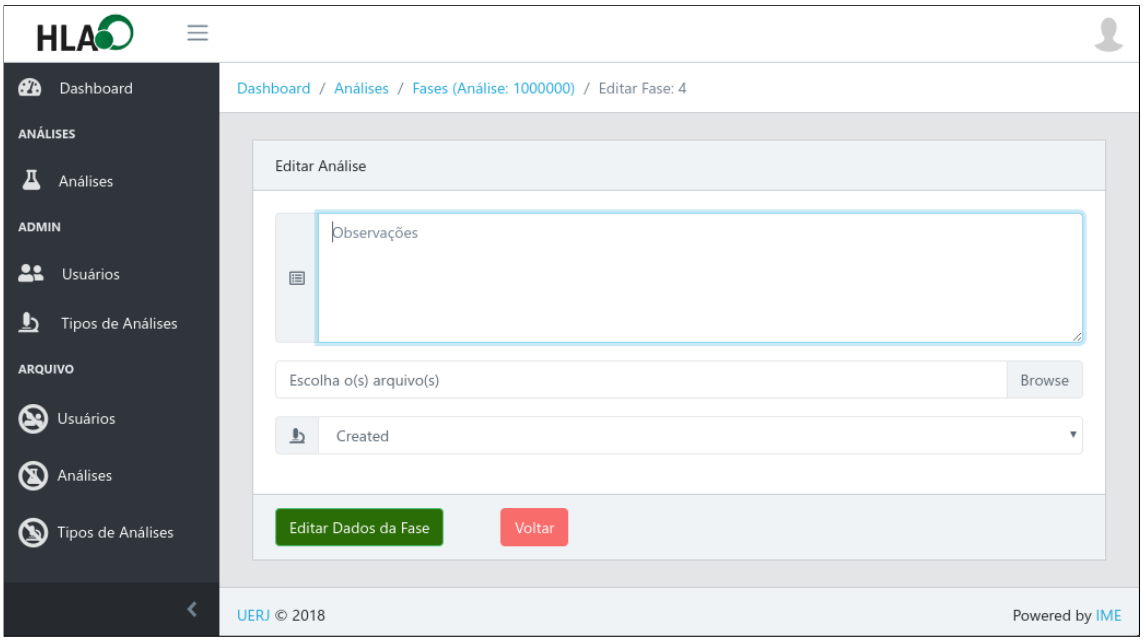
Para realizar essas operações de manipulação de arquivos, foi utilizada a *gem Shrine* [48]. Ela permite fácil configuração de todos os parâmetros necessários para manipulação de arquivos, fornecendo até métodos para verificação de formato ou tamanho de arquivo enviado, por exemplo.

Figura 27 - Tela que mostra as etapas de uma análise



Fonte: O autor, 2018

Figura 28 - Tela que mostra as etapas de uma análise



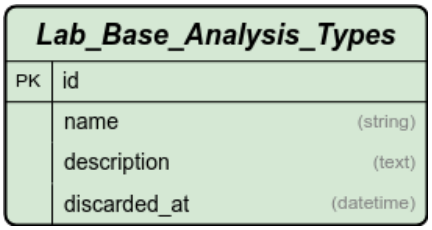
Fonte: O autor, 2018

Esse módulo está separado do módulo da microplaca com intenção de deixá-lo aplicável a qualquer situação que requeira acompanhamento. Caso o laboratório precise criar algum processo em que só interesse saber o andamento das análises e cadastrar registros de cada etapa, poderá fazê-lo simplesmente adicionando um tipo de análise novo e suas etapas.

4.6.6 Tabela do modelo *Lab_Base_Analysis_Type*

A Figura 29 mostra as principais entradas da tabela *Lab_Base_Analysis_Type*. Essa tabela tem como intuito salvar informações sobre os tipos de análise criados. Segue um descritivo dos campos da tabela.

Figura 29 - Tabela do banco de dados - *Lab_Base_Analysis_Types*



Fonte: O autor, 2018

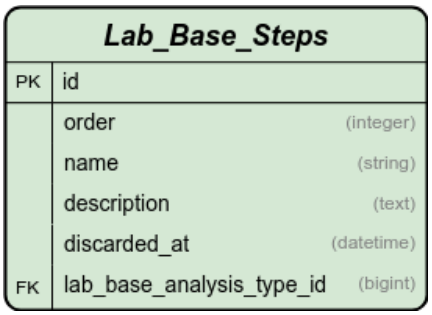
- ***id***: campo de chave primária (*PK*) de auto incremento. Gerada automaticamente na criação do modelo,
- ***name***: campo para o nome do tipo de análise,
- ***description***: campo para uma breve descrição do tipo de análise,
- ***discarded_at***: campo que irá guardar a data/hora quando um registro for arquivado. Caso contrário será igual a *NULL*.

4.6.7 Tabela do modelo *Lab_Base_Step*

A Figura 30 mostra as principais entradas da tabela *Lab_Base_Steps*. Essa tabela tem como intuito salvar informações sobre os etapas de um tipo de análise. Segue um descritivo dos campos da tabela.

- ***id***: campo de chave primária (*PK*) de auto incremento. Gerada automaticamente na criação do modelo,

Figura 30 - Tabela do banco de dados - *Lab_Base_Steps*



Fonte: O autor, 2018

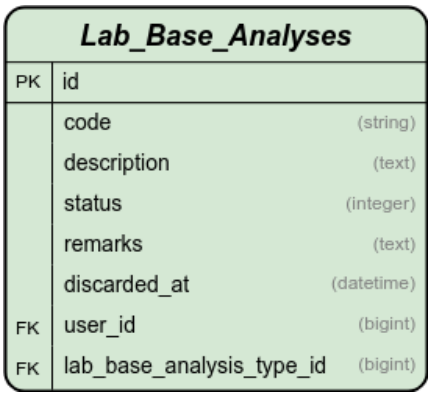
- ***order***: campo para armazenar qual o número de sequencia da fase,
- ***name***: campo para o nome da fase,
- ***description***: campo para uma breve descrição da fase,
- ***discarded_at***: campo que irá guardar a data/hora quando um registro for arquivado. Caso contrário será igual a *NULL*,
- ***lab_base_analysis_type_id***: campo de chave estrangeira (*FK*), referência para o modelo *Lab_Base_Analysis_Type*.

4.6.8 Tabela do Modelo *Lab_Base_Analysis*

A Figura 31 mostra as principais entradas da tabela *Lab_Base_Analysis*. Essa tabela tem como intuito salvar informações sobre as análises. Segue um descritivo dos campos da tabela.

- ***id***: campo de chave primária (*PK*) de auto incremento. Gerada automaticamente na criação do modelo,
- ***description***: campo para uma breve descrição da análise,
- ***status***: campo para identificar o estado da fase da análise,
- ***remarks***: campo incluir algum comentário sobre a análise, se necessário,
- ***discarded_at***: campo que irá guardar a data/hora quando um registro for arquivado. Caso contrário será igual a *NULL*,
- ***user_id***: campo de chave estrangeira (*FK*), referência para o modelo *User*,

Figura 31 - Tabela do banco de dados - *Lab_Base_Analyses*



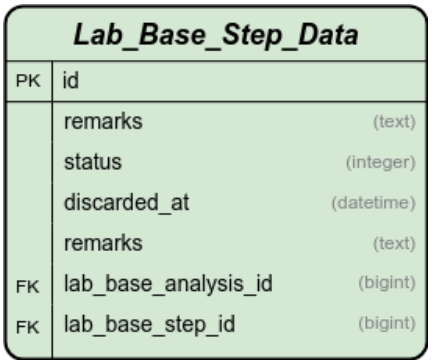
Fonte: O autor, 2018

- ***lab_base_analysis_type_id***: campo de chave estrangeira (*FK*), referência para o modelo *Lab_Base_Analysis_Type*.

4.6.9 Tabela do modelo *Lab_Base_Step_Datum*

A Figura 32 mostra as principais entradas da tabela *Lab_Base_Step_Data*. Essa tabela tem como intuito salvar informações sobre cada etapa de um tipo de análise associado a uma análise. Segue um descritivo dos campos da tabela.

Figura 32 - Tabela do banco de dados - *Lab_Base_Step_Data*



Fonte: O autor, 2018

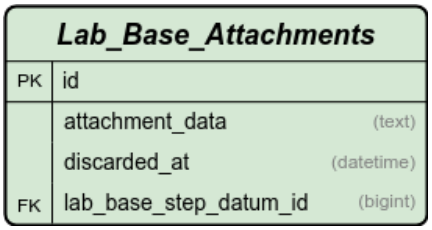
- ***id***: campo de chave primária (*PK*) de auto incremento. Gerada automaticamente na criação do modelo,
- ***remarks***: campo incluir algum comentário sobre a fase, se necessário,

- ***discarded_at***: campo que irá guardar a data/hora quando um registro for arquivado. Caso contrário será igual a *NULL*,
- ***lab_base_analysis_id***: campo de chave estrangeira (*FK*), referência para o modelo *Lab_Base_Analysis*,
- ***lab_base_step_id***: campo de chave estrangeira (*FK*), referência para o modelo *Lab_Base_Step*.

4.6.10 Tabela do modelo *Lab_Base_Attachment*

A Figura 33 mostra as principais entradas da tabela *Lab_Base_Attachments*. Essa tabela tem como intuito salvar informações sobre anexos enviados pelo usuário relativos a uma etapa de uma análise. Segue um descritivo dos campos da tabela.

Figura 33 - Tabela do banco de dados - *Lab_Base_Attachments*



Fonte: O autor, 2018

- ***id***: campo de chave primária (*PK*) de auto incremento. Gerada automaticamente na criação do modelo,
- ***attachment_data***: campo que guarda o caminho relativo do arquivo anexado,
- ***discarded_at***: campo que irá guardar a data/hora quando um registro for arquivado. Caso contrário será igual a *NULL*,
- ***lab_base_step_datum_id***: campo de chave estrangeira (*FK*), referência para o modelo *Lab_Base_Step_Datum*.

4.6.11 Módulo *Donors_Management*

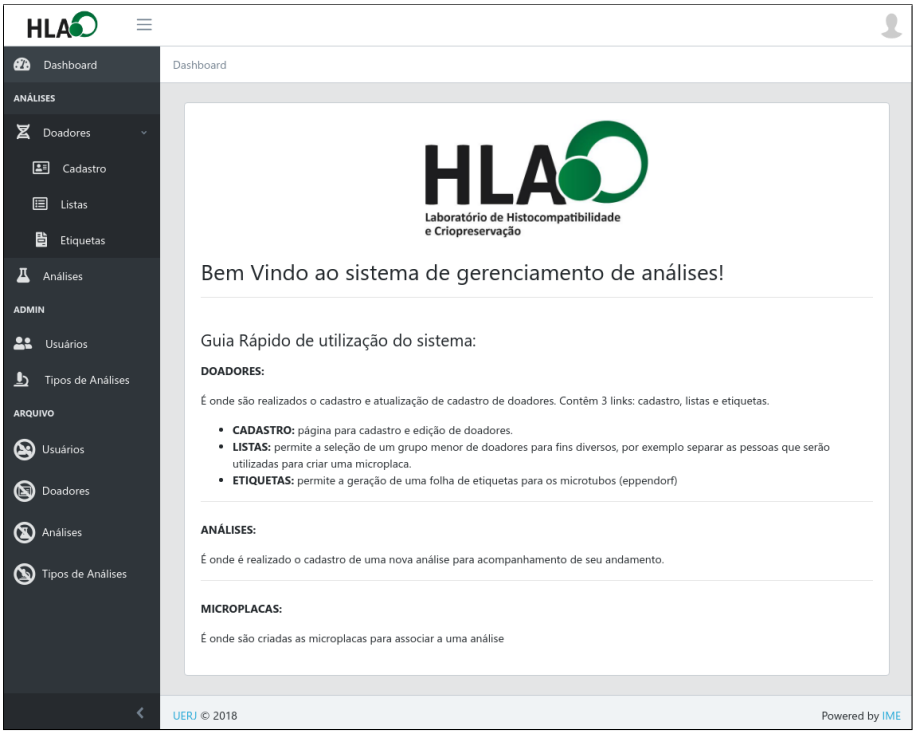
Esse módulo provê ferramentas para o gerenciamento do cadastro de doadores. A intenção é que este seja uma tabela do banco de dados que substitua as planilhas feitas

com o *software* Excel® pelo laboratório. Com os recursos implementados fica muito mais fácil a inclusão e alteração de dados, bem como a busca - tarefa bastante corriqueira no cotidiano do laboratório e pouco prática de ser executada com o Excel®.

Os campos presentes foram obtidos com a equipe do laboratório e visam atender a vários setores. A equipe administrativa utiliza alguns, enquanto a equipe de laboratório outros, etc. O preenchimento só é obrigatório para poucos campos comuns a todas as atividades.

A Figura 34 apresenta a tela vista pelos usuários quando se logam no sistema com o módulo *Donors_Management* ativo. É possível ver no menu lateral os links relativos a parte de manipulação de dados de doadores. Para os usuários administradores aparece também uma entrada na área *Arquivo* do menu.

Figura 34 - Tela após ao login com o menu de doadores



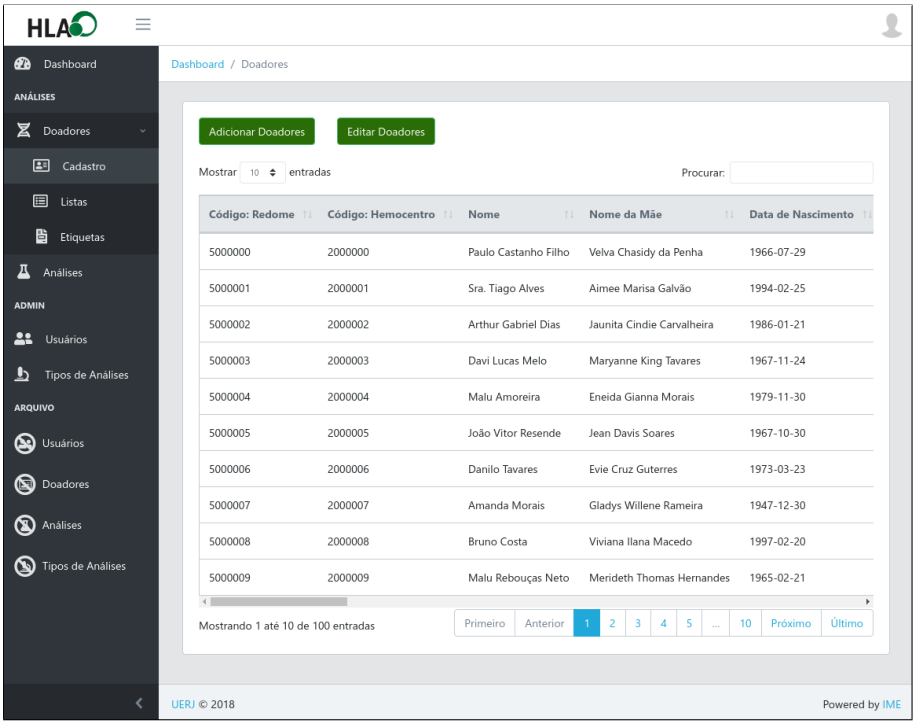
Fonte: O autor, 2018

Ao clicar na opção “Cadastro”, o usuário é levado para uma tela onde é apresentada uma lista com todos os usuários cadastrados no sistema (Figura 35). Na tabela apresentada é possível realizar buscas e ordenar as entradas por ordem crescente ou descrente.

A tela na Figura 36 mostra como é feita a adição de doadores. Como são muitos os parâmetros a serem preenchidos, e o laboratório recebe essas informações em uma planilha, optou-se por fazer essa operação em lote e não individualmente.

Não é preciso que todos os parâmetros sejam preenchidos de uma vez. O usuários pode escolher quais campos pretendem preencher marcando-os nos *checkboxes* disponíveis.

Figura 35 - Tela que mostra todos os doadores cadastrados



Fonte: O autor, 2018

Em seguida, ao clicar no link “Criar Arquivo”, será baixada uma planilha com formato *csv* com os campos escolhidos, pronta para preenchimento. Após preenchê-la, basta que o usuário a envie de volta clicando no botão de seleção de arquivos disponível e por fim, clique no botão *Cadastrar doadores*. Instruções detalhadas estão disponíveis na própria página.

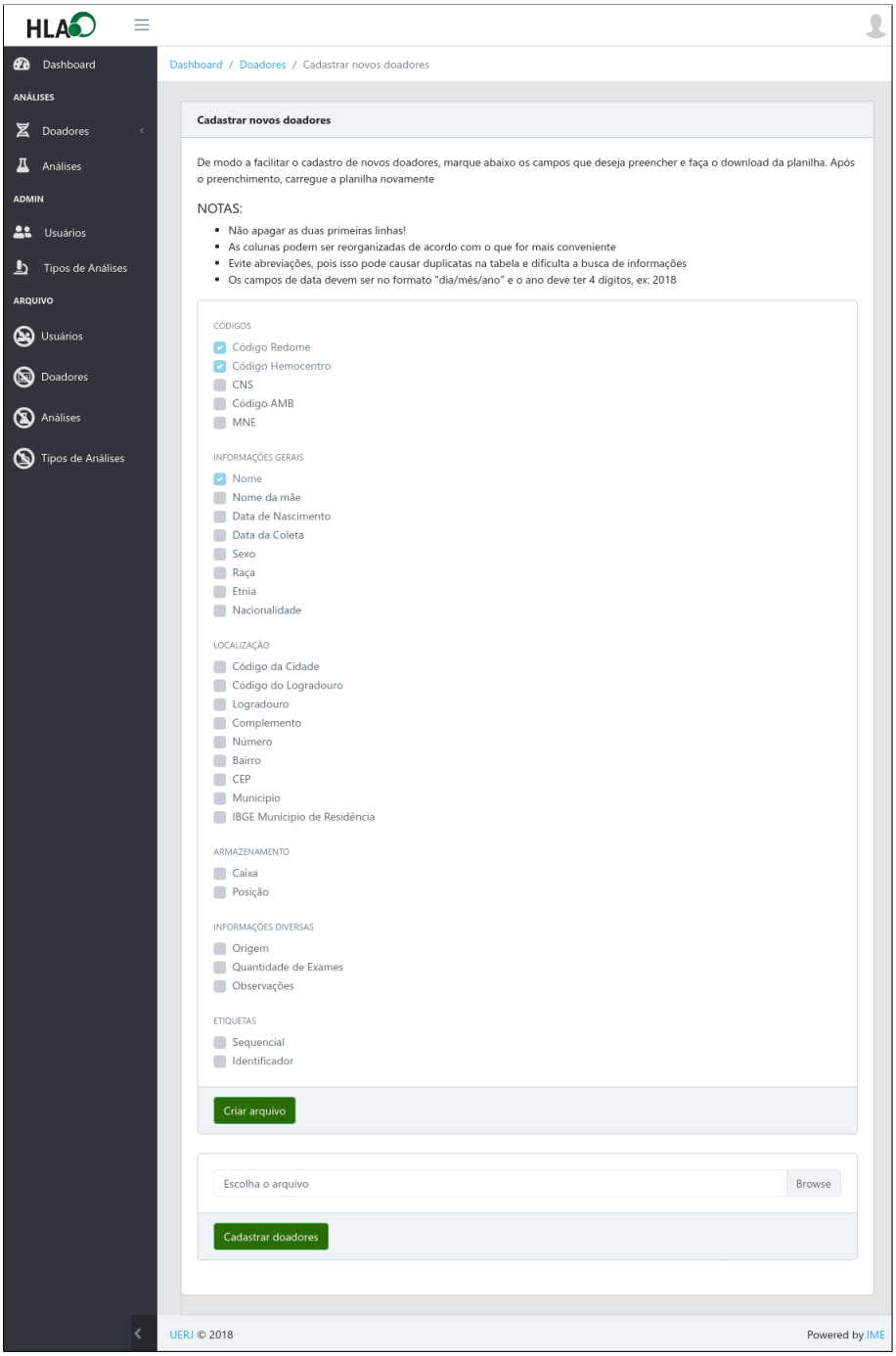
A edição pode ser feita de duas formas. Em lote, onde o usuário seleciona uma lista criada previamente com os nomes dos doadores que deseja editar (a criação de listas é apresentada a seguir) ou, se a mudança for pontual, basta clicar na célula que pretende corrigir na tabela com as informações. Essa tela está apresentada na Figura 37.

A criação de listas tem o intuito de criar uma pequeno conjunto dentro do cadastro geral de doadores para facilitar manipulações futuras, por exemplo, para criar uma microplaca. Assim não é preciso selecionar os doadores do cadastro geral, que pode vir a se tornar bastante extenso com o tempo.

A tela de criação pode ser vista na Figura 38. O usuário seleciona na tabela de baixo (cadastro geral) as pessoas que deseja incluir na lista e, em seguida, clica no botão com a seta para cima. Essas pessoas serão adicionadas a tabela de cima, que é a lista. Basta então dar um nome para lista e clicar no botão “Criar Lista”.

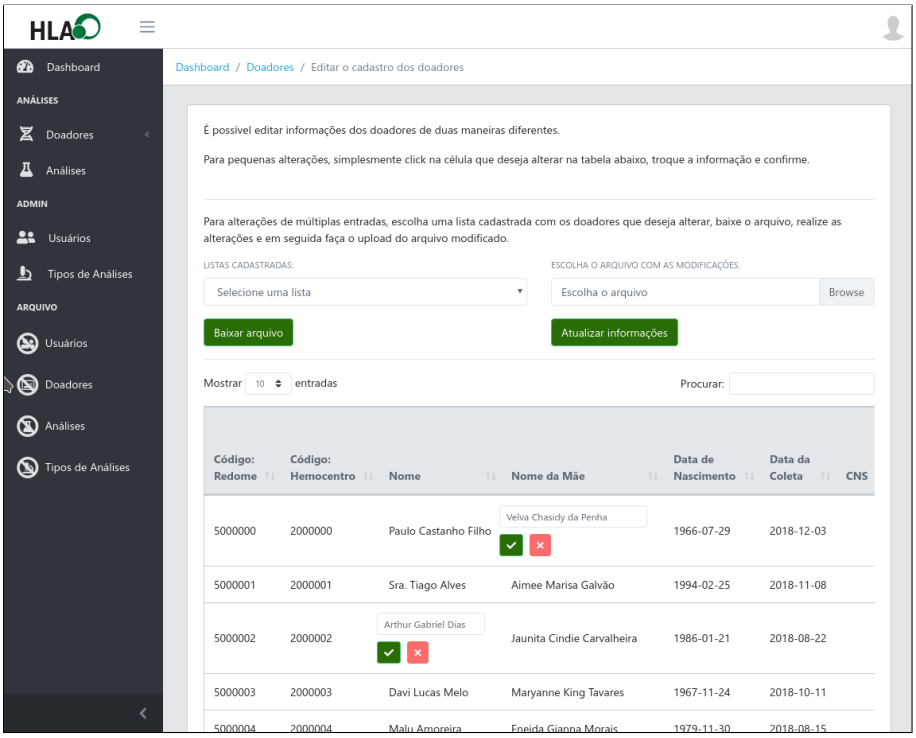
A criação de etiquetas é feita utilizando uma lista. Basta selecionar a lista com os nomes que irão para etiqueta que será criado um documento no formato *pdf* com a folha

Figura 36 - Tela para adição de doadores



Fonte: O autor, 2018

Figura 37 - Tela para edição de informações dos doadores



Fonte: O autor, 2018

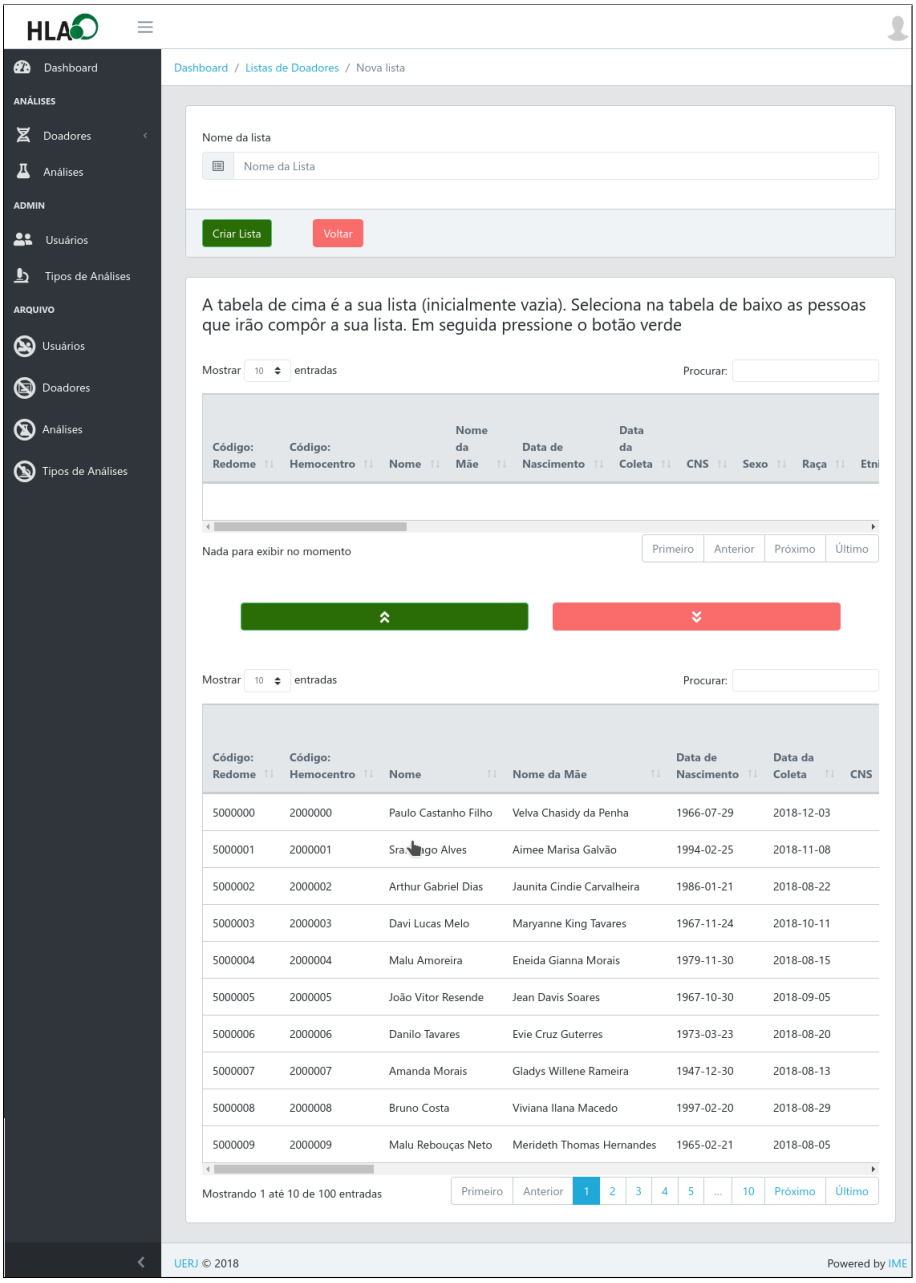
de etiquetas. As informações apresentadas tiveram como base a etiqueta apresentada na Figura 4. As dimensões e disposição das etiquetas na folha seguem o padrão da folha A5Q 1226 da empresa Pimaco®, o mesmo que é utilizado no laboratório. A Figura 39 apresenta a tela de criação de etiquetas.

4.6.12 Tabela do Modelo *Donors_Management_Donor*

A Figura 40 mostra as principais entradas da tabela *Donors_Management_Donors*. Essa tabela tem como intuito salvar informações sobre os doadores. É uma tabela bastante extensa dada a quantidade de informações necessárias. Segue um descritivo dos campos da tabela.

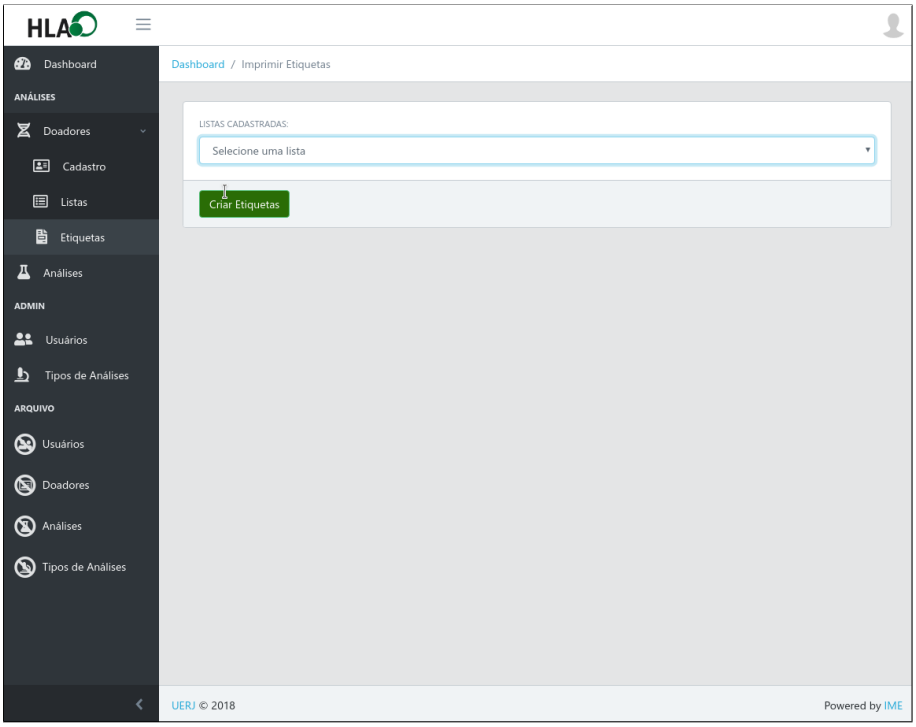
- ***id***: campo de chave primária (*PK*) de auto incremento. Gerada automaticamente na criação do modelo,
- ***redome_code***: campo para o código do redome,
- ***blood_center_code***: campo para o código interno do laboratório "número de DNA",

Figura 38 - Tela para criação de listas



Fonte: O autor, 2018

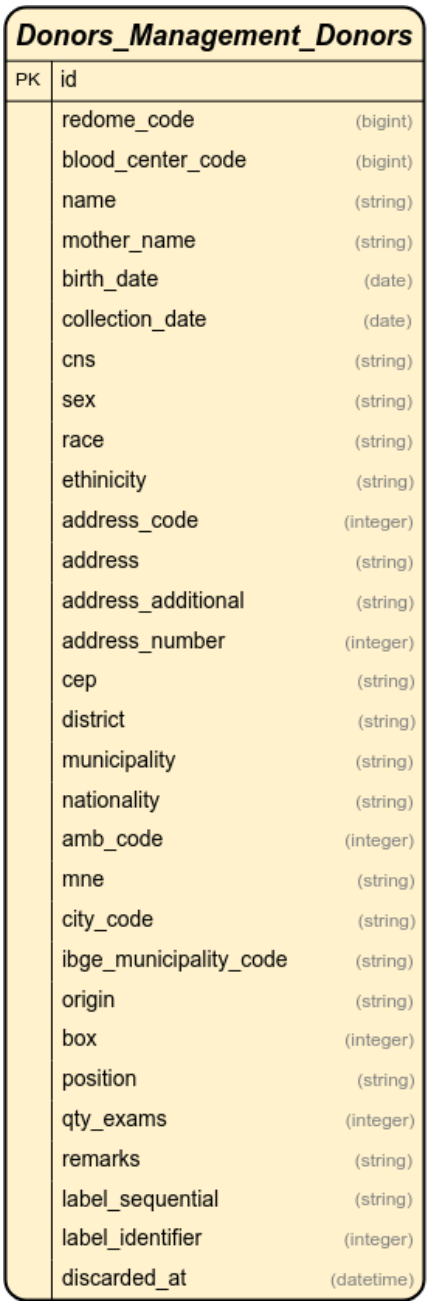
Figura 39 - Tela para geração da folha de etiquetas



Fonte: O autor, 2018

- ***nome***: campo para o nome do doador,
- ***mother_name***: campo para o nome da mãe do doador,
- ***birth_date***: campo para a data da coleta da amostra,
- ***collection_date***: campo para a data de nascimento do doador,
- ***cns***: campo para o código nacional de saúde,
- ***sex***: campo para identificação do sexo do doador,
- ***race***: campo para identificação da raça do doador,
- ***ethinicity***: campo para identificação da etnia do doador,
- ***address_code***: campo para o código do do endereço do doador,
- ***address***: campo para o endereço do doador,
- ***address_additional***: campo para dados adicionais do endereço do doador,
- ***address_número***: campo para número do endereço do doador,
- ***cep***: campo para cep do doador,

Figura 40 - Tabela do banco de dados - *Donors_Management_Donors*



Fonte: O autor, 2018

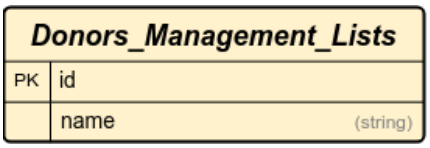
- ***district***: campo para o bairro do doador,
- ***municipality***: campo para o município do doador,
- ***nationality***: campo para a nacionalidade do doador,
- ***amb__code***: campo para código amb,
- ***mne***: campo para o código mne,
- ***city__code***: campo para código da cidade do doador,
- ***ibge__municipality__code***: campo para código do município (IBGE),
- ***origin***: campo para origem,
- ***box***: campo para o número da caixa que será armazenada a amostra,
- ***position***: campo para a posição da caixa,
- ***qty__exams***: campo para indicar a quantidade de exames que será realizada,
- ***remarks***: campo para observações,
- ***label__sequential***: campo para guardar o número sequencial para criação de etiquetas,
- ***label__identifier***: campo para guardar o identificador para criação de etiquetas,
- ***discarded__at***: campo que irá guardar a data/hora quando um registro for arquivado. Caso contrário será igual a *NULL*.

4.6.13 Tabela do Modelo *Donors_Management_List*

A Figura 41 mostra as principais entradas da tabela *Donors_Management_Lists*. Essa tabela tem como intuito salvar o nome da lista de doadores. Segue um descritivo dos campos da tabela.

- ***id***: campo de chave primária (*PK*) de auto incremento. Gerada automaticamente na criação do modelo,
- ***name***: campo para o nome do lista,

Figura 41 - Tabela do banco de dados - *Donors_Management_Lists*

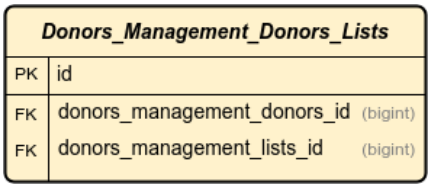


Fonte: O autor, 2018

4.6.14 Tabela do Modelo *Donors_Management_Donors_List*

A Figura 42 mostra as principais entradas da tabela *Donors_Management_Donors_List*. Essa tabela tem como intuito fazer uma associação entre os doadores no cadastro e os doadores separados pela lista. Segue um descritivo dos campos da tabela.

Figura 42 - Tabela do banco de dados - *Donors_Management_Donors_Lists*



Fonte: O autor, 2018

- ***id***: campo de chave primária (*PK*) de auto incremento. Gerada automaticamente na criação do modelo,
- ***donors_managment_donors_id***: campo de chave estrangeira (*FK*), referência para o modelo *Donors_Management_Donors*.
- ***donors_managment_lists_id***: campo de chave estrangeira (*FK*), referência para o modelo *Donors_Management_Lists*

4.6.15 Módulo *Microplate*

A criação das microplacas é um ponto bastante importante para a aplicação e parte crucial nas análises. Como mencionado anteriormente, o documento que gerencia a organização da microplaca (Figura 6) é o mais cansativo de ser preenchido, logo o mais sujeito a erros.

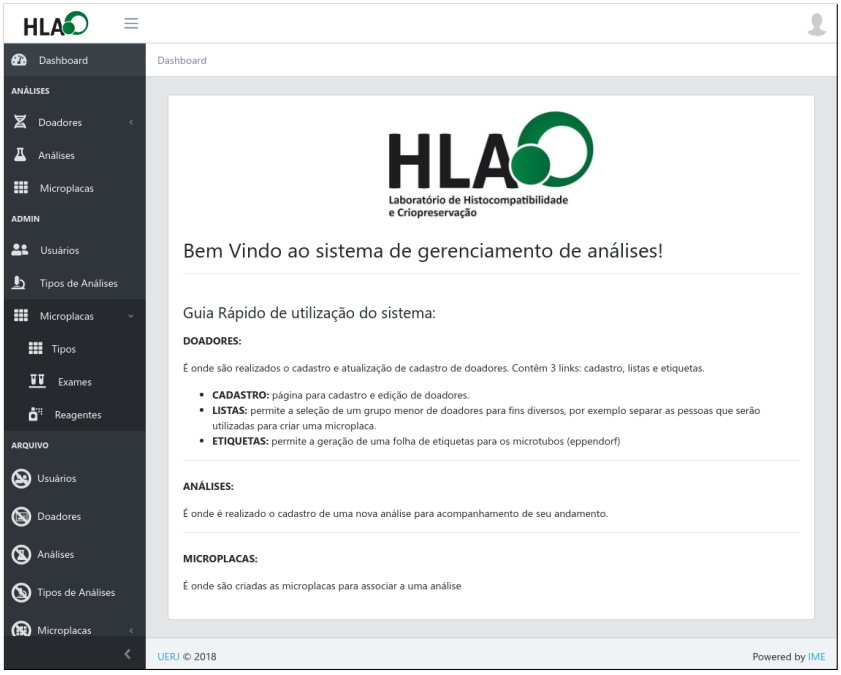
De modo a deixar a aplicação flexível para outros tipos de microplaca, a aplicação não se detêm somente no tipo usado pelo laboratório (96 poços). Foi construído um

ambiente onde é possível criar um “tipo” de microplaca com tamanhos especificados pelo usuário (linhas e colunas) e associar a ela as informações de reagentes e exames que podem ser feitos, conforme for necessário.

Como essa tarefa faz parte das configurações gerais do sistema, e tenderá a não sofrer mudanças ao longo do tempo, somente usuários com acesso de administrador podem fazê-la.

A Figura 43 apresenta a tela após login com o módulo de microplacas ativado. Além do *link* para as microplacas associadas as anaĺises, para os usuários administradores são criados dois campos extras: um na área de administração, que tem o intuito de permitir a criação de tipos de microplaca, e outro na área de arquivo para permitir o acesso aos tipos de microplacas e também as microplacas que estão associadas a análises que foram arquivadas.

Figura 43 - Tela após ao login - Visão do administrador (todos os módulos ativos)



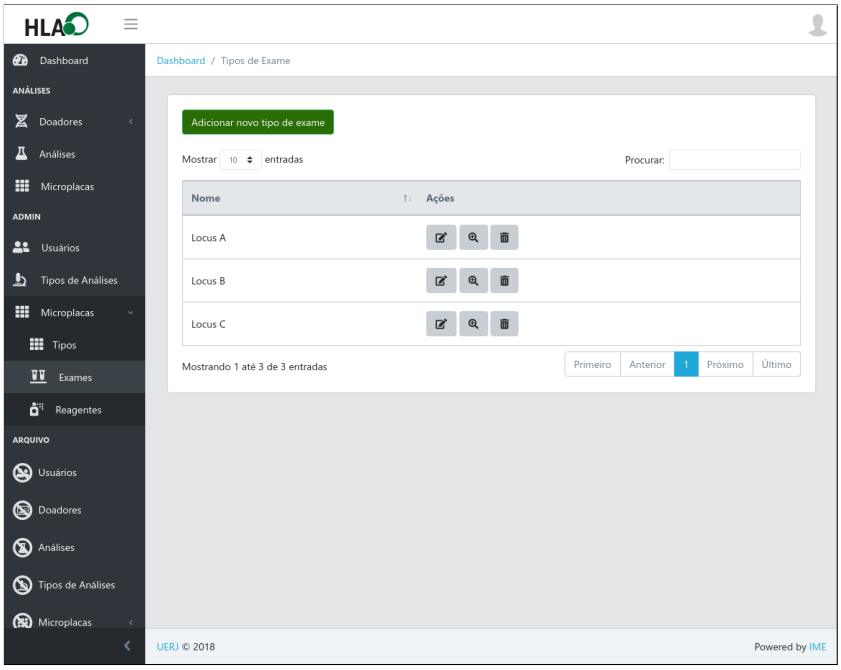
Fonte: O autor, 2018

Antes de criar o tipo de microplaca é preciso criar primeiro os exames e reagentes associados a ela.

Para criar os exames, basta clicar no botão “Criar novo tipo de exame” na tela que apresenta todos os exames. Esse campo requer somente um nome (Figura 44).

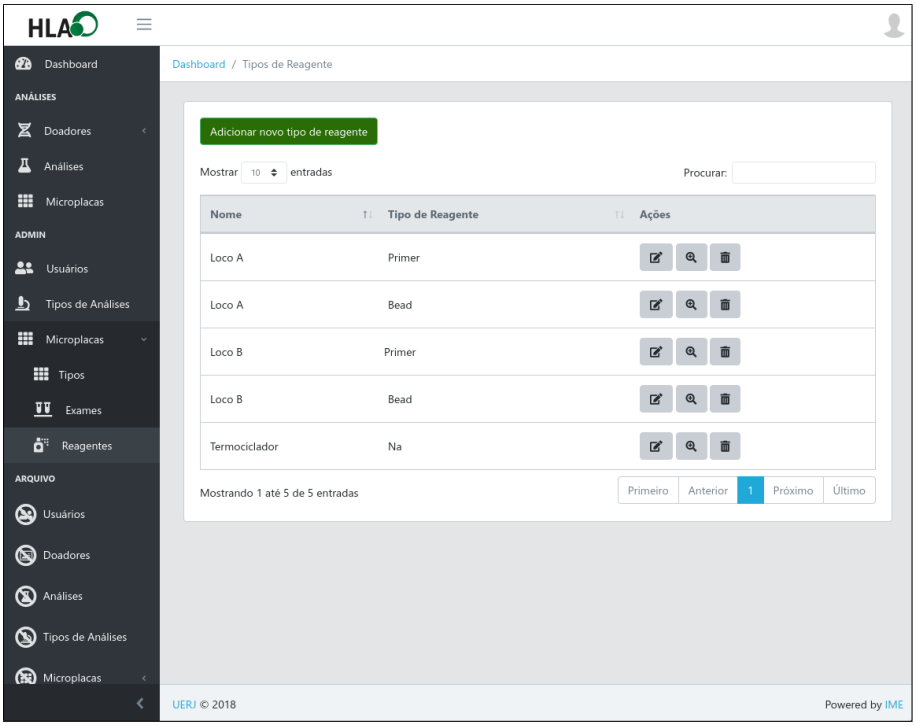
De forma análoga, pode-se criar os reagentes. A única diferença é que além do nome é preciso escolher um tipo: *primer*, *bead* ou *NA* (Não Aplicável). A Figura 45 apresenta a tela com os todos os reagente criados.

Figura 44 - Tela para criar os exames que poderão ser realizados por uma microplaca



Fonte: O autor, 2018

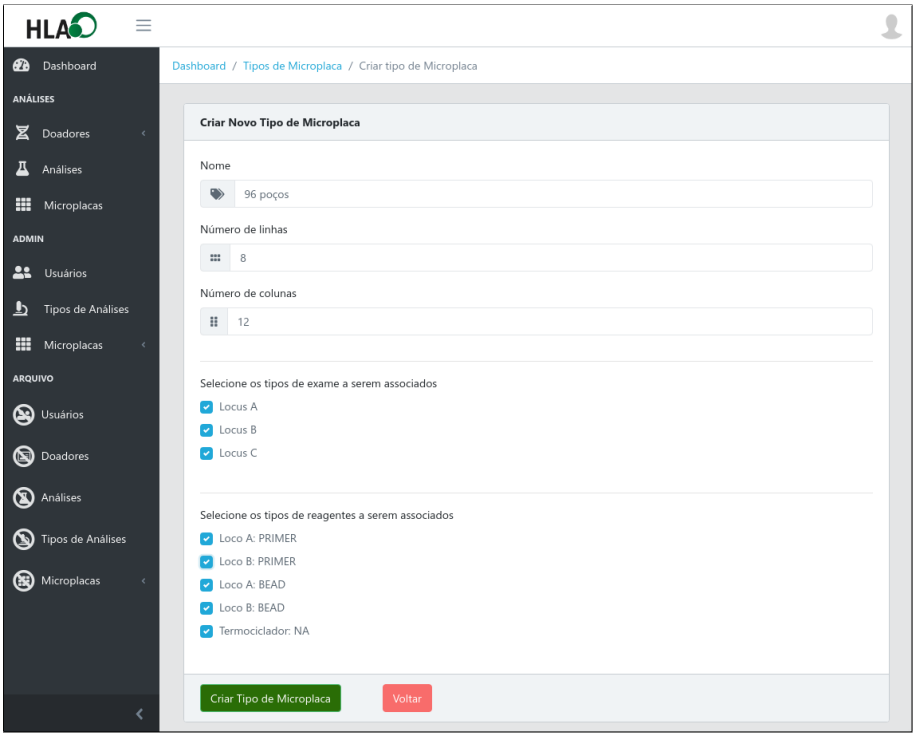
Figura 45 - Tela para o cadastro de reagentes a serem associados a uma microplaca



Fonte: O autor, 2018

Por fim, é possível então criar um tipo de microplaca. A Figura 46 apresenta a tela de criação de tipo de microplaca. Nela é preciso escolher a quantidade de linha e colunas, e selecionar os reagentes e exames associados a placa.

Figura 46 - Tela que a criação de tipos microplacas



Fonte: O autor, 2018

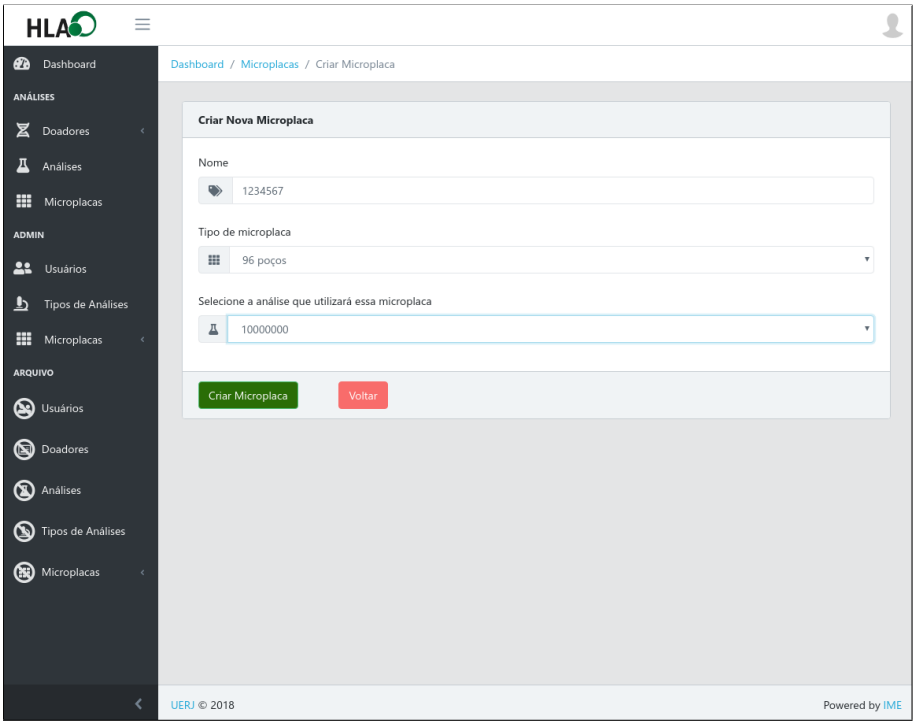
Para criar uma microplaca para uma análise específica. Como pode ser visto na Figura 47, é preciso dizer o número da microplaca, selecionar o tipo e a análises que será associada.

A microplaca aparece na lista geral. Para acessar o mapa de trabalho, basta clicar no ícone na coluna associada a microplaca criada, Figura 49.

Para preencher a microplaca, é preciso selecionar a lista criada previamente com os nomes dos doadores cujas amostras estão disponíveis. Feito isso, o conteúdo da lista aparecerá na menu “Pessoas cadastradas na lista”. Basta selecionar uma, selecionar o exame e apertar o botão “Criar amostra”. Será criado uma amostra com os dados informados na área delimitada denominada “amostra”. Basta então arrastar para o local que desejar.

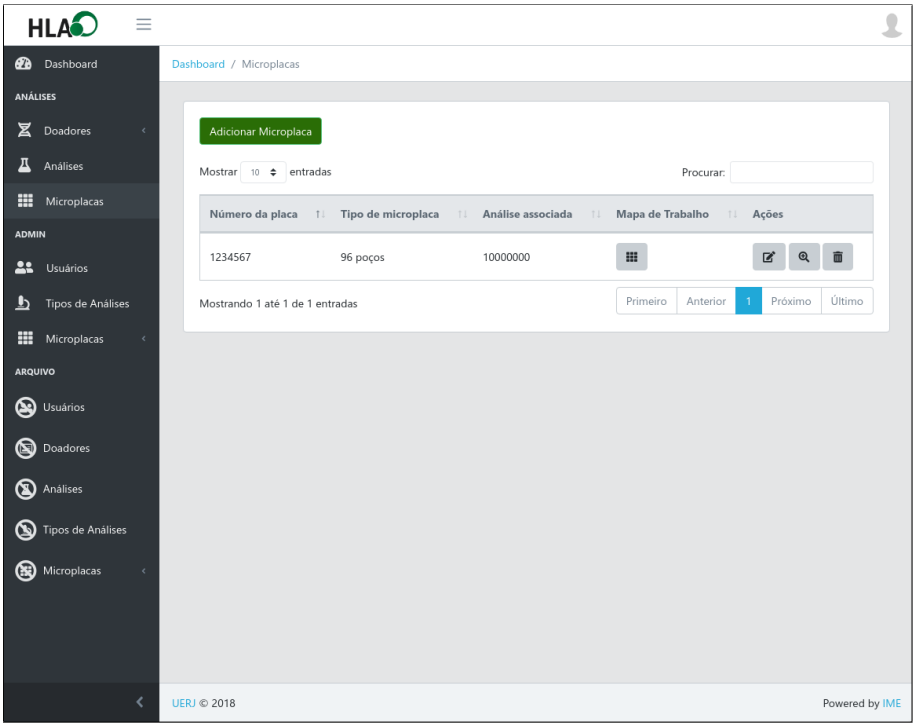
Para incluir os dados dos reagentes como o lote e data de fabricação, basta clicar no botão “Editar informações dos Reagentes” (Figura 50).

Figura 47 - Tela que mostra a criação de uma microplaca associada a uma análise



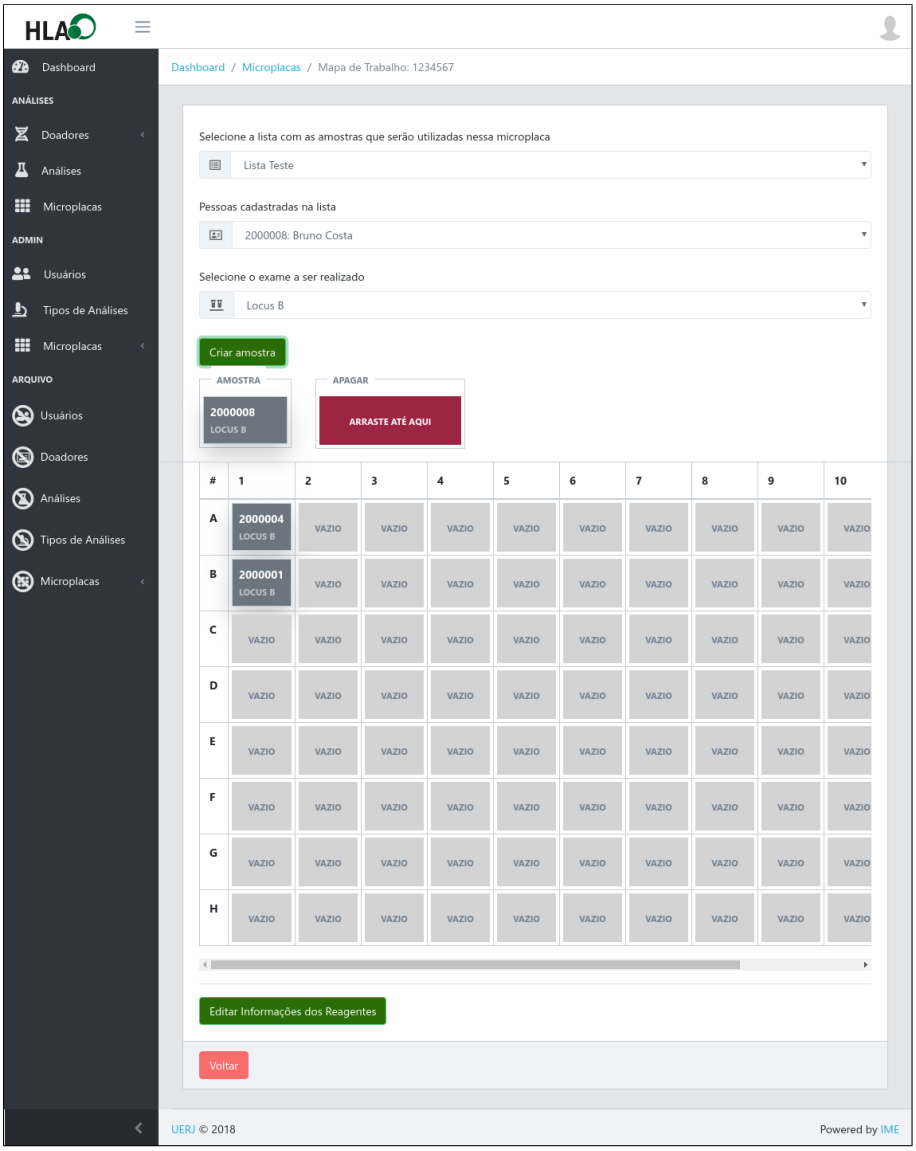
Fonte: O autor, 2018

Figura 48 - Tela que mostra todas as microplacas ativas



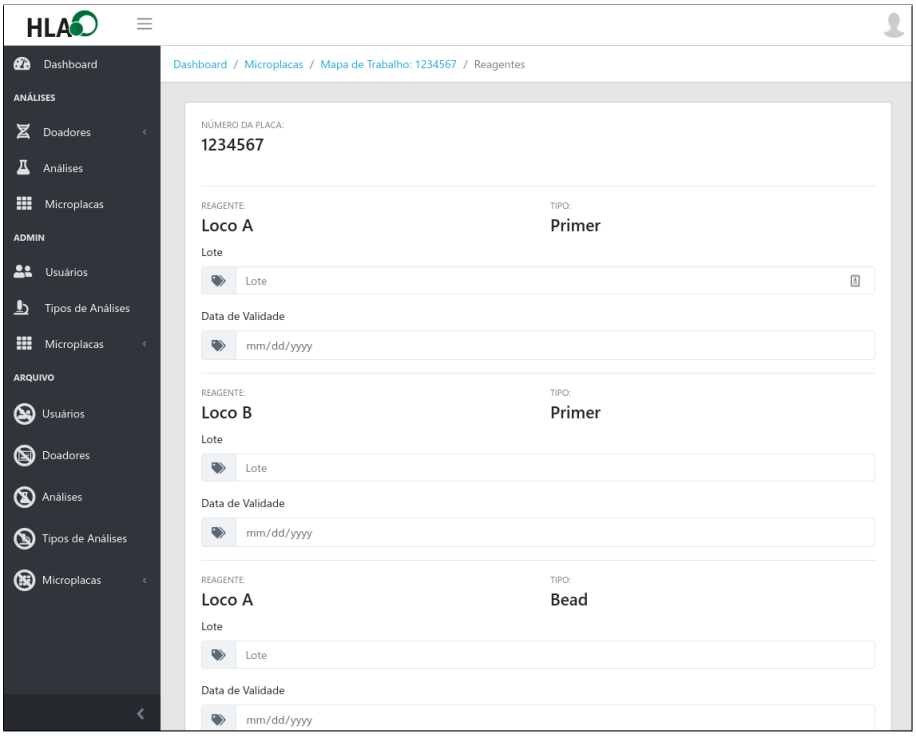
Fonte: O autor, 2018

Figura 49 - Tela que mostra o mapa de trabalho



Fonte: O autor, 2018

Figura 50 - Tela que mostra todos os reagentes associados a microplaca

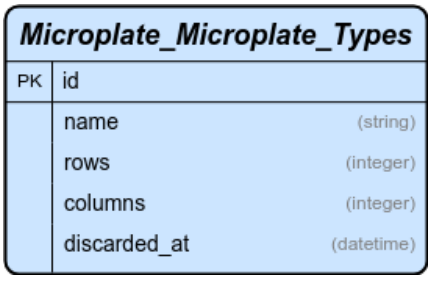


Fonte: O autor, 2018

4.6.16 Tabela do modelo *Microplate_Microplate_Type*

A Figura 51 mostra as entradas da tabela *Microplate_Microplate_Types*. Essa tabela tem como intuito salvar as informações dos tipos de microplacas criados. Segue um descritivo dos campos da tabela.

Figura 51 - Tabela do banco de dados - *Microplate_Microplate_Types*



Fonte: O autor, 2018

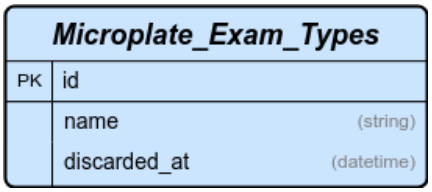
- ***id***: campo de chave primária (*PK*) de auto incremento. Gerada automaticamente na criação do modelo,

- ***name***: campo nome do tipo de microplaca,
- ***rows***: campo para a quantidade de linhas da microplaca,
- ***columns***: campo para a quantidade de colunas da microplaca,
- ***discarded_at***: campo que irá guardar a data/hora quando um registro for arquivado. Caso contrário será igual a *NULL*,

4.6.17 Tabela do modelo *Microplate_Exam_Type*

A Figura 52 mostra as entradas da tabela *Microplate_Exam_Types*. Essa tabela tem como intuito salvar as informações dos tipos exames a serem associados com um tipo de microplaca. Segue um descritivo dos campos da tabela.

Figura 52 - Tabela do banco de dados - *Microplate_Exam_Types*



Fonte: O autor, 2018

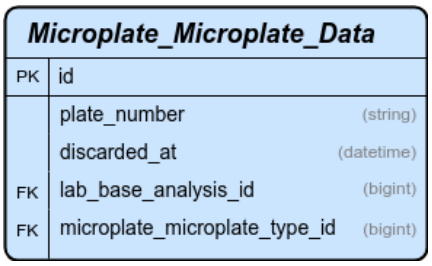
- ***id***: campo de chave primária (*PK*) de auto incremento. Gerada automaticamente na criação do modelo,
- ***name***: campo para o nome do tipo de microplaca,
- ***discarded_at***: campo que irá guardar a data/hora quando um registro for arquivado. Caso contrário será igual a *NULL*,

4.6.18 Tabela do modelo *Microplate_Microplate_Datum*

A Figura 53 mostra as entradas da tabela *Microplate_Microplate_Data*. Essa tabela tem como intuito salvar as informações das microplacas criadas para as análises. Segue um descritivo dos campos da tabela.

- ***id***: campo de chave primária (*PK*) de auto incremento. Gerada automaticamente na criação do modelo,

Figura 53 - Tabela do banco de dados - *Microplate_Microplate_Data*



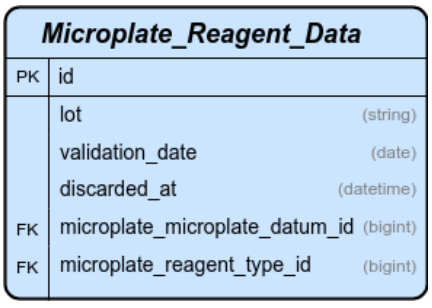
Fonte: O autor, 2018

- ***plate_number***: campo para o numero da microplaca,
- ***discarded_at***: campo que irá guardar a data/hora quando um registro for arquivado. Caso contrário será igual a *NULL*,
- ***lab_base_analysis_id***: campo de chave estrangeira (*FK*), referência para o modelo *Lab_Base_Analysis*,
- ***microplate_microplate_type_id***: campo de chave estrangeira (*FK*), referência para o modelo *Microplate_Microplate_Type*,

4.6.19 Tabela do modelo *Microplate_Reagent_Datum*

A Figura 54 mostra as entradas da tabela *Microplate_Reagent_Data*. Essa tabela tem como intuito salvar as informações dos reagentes associados a uma microplaca. Segue um descritivo dos campos da tabela.

Figura 54 - Tabela do banco de dados - *Microplate_Reagent_Data*



Fonte: O autor, 2018

- ***id***: campo de chave primária (*PK*) de auto incremento. Gerada automaticamente na criação do modelo,

- ***lot***: campo para o numero do lote do reagente,
- ***validation_date***: campo para a data de validade do reagente,
- ***discarded_at***: campo que irá guardar a data/hora quando um registro for arquivado. Caso contrário será igual a *NULL*,
- ***lab_base_analysis_id***: campo de chave estrangeira (*FK*), referência para o modelo *Lab_Base_Analysis*,
- ***microplate_microplate_datum_id***: campo de chave estrangeira (*FK*), referência para o modelo *Microplate_Microplate_Datum*,
- ***microplate_reagent_type_id***: campo de chave estrangeira (*FK*), referência para o modelo *Microplate_Reagent_Type*,

4.6.20 Discussão sobre a implementação com o supervisor do laboratório

Após a finalização do desenvolvimento, uma versão "*alpha*" foi apresentada para o supervisor do laboratório, com intuito de avaliar se os recursos planejados realmente atenderiam aos requisitos funcionais do laboratório e também coletar informações sobre melhorias ou propostas para desenvolvimentos futuros.

Foi realizada uma apresentação geral do funcionamento da aplicação e, em seguida, uma discussão sobre como seria o fluxo de informações em um caso real do laboratório.

O produto mostrou-se apto para o funcionamento atendendo confortavelmente todas as necessidades atuais.

Pequenos ajustes foram sugeridos, bem como sugestões para expansões futuras.

Para o módulo *Core*, solicitou-se a inclusão de um campo extra no cadastro de pessoas, um campo para um número interno de registro, que seja sequencial. Para facilitar o cadastro, ainda foi solicitado que este número seja automaticamente definido.

Outra sugestão foi criar um numeração automática para o campo "posição". Esse campo corresponde a aproximadamente o local onde será guardada a amostra na caixa. Como a caixa comporta 100 amostras, essa numeração seguiria de 1 a 100, voltando ao início no fim (nova caixa).

Para o módulo *Lab_Base*, o supervisor levantou uma questão importante sobre a relação entre as análises. Em muitos casos, uma análise é criada com o objetivo de corrigir problemas em análises anteriores. Sugeriu-se então a possibilidade de criar relações entre as análises.

Para o módulo *Microplate*, a única sugestão foi permitir a edição do tipo de reagente, atualmente está opção não está disponível.

Como módulo extra, sugeriu-se um módulo de pendências, que marcaria automaticamente as pessoas não incluídas em análises e também os casos onde ocorreram problemas, para alertar aos usuários.

Todas as solicitações podem facilmente ser atendidas com poucos ajustes. Como o presente trabalho é fruto de um projeto ainda em andamento, a aplicação será atualizada com as sugestões antes do lançamento final.

CONCLUSÃO

O presente trabalho teve como objetivo refazer todo o mapeando das etapas associadas ao processo de tipificação HLA desenvolvido no laboratório HLA-UERJ, e propor e desenvolver uma aplicação customizável que auxiliasse no controle das atividades envolvidas nessa análise. O tamanho da tarefa exigiu com que fossem utilizadas algumas técnicas computacionais para garantir que o sistema não só atendesse aos requisitos imediatos do laboratório, mas pudesse ser de fácil manutenção, atualização e expansão.

Para essa tarefa foi criada uma aplicação completa, que mapeia toda a vida de uma amostra ao longo das diversas etapas de uma análise dentro do laboratório. Foi utilizado para isso o *framework Ruby on Rails*, por ser gratuito e ter diversas características que se enquadram perfeitamente no objetivo, como ser de fácil aprendizado, implementação e modularização.

O sistema desenvolvido teve como base o fluxo de informações apresentado no trabalho, obtido com a colaboração dos funcionários do laboratório. Foram apresentados todas as parte da aplicação e também estão disponíveis instruções para atualização e expansão futuras.

A disponibilidade da aplicação em módulos e a possibilidade de criação de tipos de análise e microplacas, por exemplo, fazem com que seja possível utilizar a aplicação em outros tipos de laboratórios, pois permite fácil adaptação para outros processos.

Como trabalhos futuros, são muitas as funcionalidades que podem ser incluídas no sistema para deixá-lo mais completo e robusto. Dentre elas sugere-se a criação de um sistema de *chat* interno para trocas de mensagens rápidas entre os funcionários. Dessa forma evitaria-se o deslocamento para pequenos esclarecimentos. Outro módulo importante seria um que pudesse obter dados estatísticos das análises realizadas, como por exemplo o tempo total, onde estão os maiores retrabalhos, entre outros. Poderiam também ser incluídos módulos de controle de estoque, que monitorasse inclusive a validade dos reagentes e emitisse alertas quando os produtos estivessem acabando ou perto do prazo de validade. Com o auxílio de microcontroladores e um pequeno módulo extra na aplicação, seria possível automatizar uma tarefa manual do laboratório que é bastante laboriosa, que é a verificação das temperaturas de todos os freezers onde as amostras estão guardadas. Seriam colocados sensores em todos os freezers e por meio do controlador seriam enviados dados para o módulo que faria o controle, emitindo alertas quando necessário.

REFERÊNCIAS

1 ACKERKNECHT, E.; ROSENBERG, C. *A Short History of Medicine*. [S.l.]: Johns Hopkins University Press, 2016. ISBN 9781421419558.

2 HAIDEGGER, T.; SÁNDOR, J.; BENYÓ, Z. Surgery in space: the future of robotic telesurgery. *Surgical Endoscopy*, v. 25, n. 3, p. 681–690, Mar 2011.

3 RIFKIN, J. *O Século da biotecnologia: a valorização dos genes e a reconstrução do mundo*. [S.l.]: Makron Books, 1999. 290 p.

4 GARCIA, C. D.; PEREIRA, J. D.; GARCIA, V. D. *Doação e transplante de órgãos e tecidos*. São Paulo: Segmento Farma Editores Ltda, 2015. ISBN 978-85-7900-090-4.

5 ABBAS, A. K.; LICHTMAN, A. H. *Basic Immunology. Functions and Disorders of the Immune System*. 2. ed. [S.l.]: Elsevier, 2004.

6 OPELZ, G.; DöHLER, B. Effect of human leukocyte antigen compatibility on kidney graft survival: Comparative analysis of two decades. *Transplantation*, Ovid Technologies (Wolters Kluwer Health), v. 84, n. 2, p. 137–143, jul 2007. Disponível em: <https://doi.org/10.1097/01.tp.0000269725.74189.b9>.

7 HLA-UERJ. *HLA - Laboratório de Histocompatibilidade e Criopreservação - UERJ*. 2018. Disponível em: <https://www.hla.uerj.br/site>. Acesso em: 01 jan. 2018.

8 CAMPANA, G. A.; OPLUSTIL, C. P.; FARO, L. B. de. Tendências em medicina laboratorial. *Jornal Brasileiro de Patologia e Medicina Laboratorial*, FapUNIFESP (SciELO), v. 47, n. 4, p. 399–408, aug 2011. Disponível em: <https://doi.org/10.1590/s1676-24442011000400003>.

9 FELDER, R. A. Modular workcells: modern methods for laboratory automation. *Clinica Chimica Acta*, Elsevier BV, v. 278, n. 2, p. 257–267, dec 1998. Disponível em: <https://doi.org/10.1016/s0009-8981(98)00151-x>.

10 PASZKO, C.; TURNER, E. *Laboratory Information Management Systems, Second Edition,*. [S.l.]: CRC Press, 2001. ISBN 0824705211.

11 RIBEIRO, B. G. C. *Estudo e proposta de sistema para o processo de tipificação HLA praticado pelo Laboratório de HLA da UERJ*. 2017. Dissertação (Mestrado em Ciências Computacionais) — Universidade do Estado do Rio de Janeiro, Instituto de Matemática e Estatística, Rio de Janeiro, 2017.

12 RICHARDSON, R. *Pesquisa social: métodos e técnicas*. [S.l.]: Atlas, 1999. ISBN 9788522421114.

13 DENIZET, T. *The Complete Guide to Modular Rails Application*. 9. ed. [S.l.]: Thibault Denizet, 2015.

14 MIRANDA-VILELA, A. L. *A diversidade genética do complexo principal de histocompatibilidade (MHC) e sua relação com a susceptibilidade para doenças auto-imunes e câncer*. Monografia (Especialização em Genética Humana) — Universidade de Brasília, Ribeirão Preto, SP, 2007.

15 MBBS, A. K. A.; PHD, A. H. H. L. M.; PHD, S. P. M. *Cellular and Molecular Immunology*. [S.l.]: Elsevier, 2017. ISBN 9780323479783.

16 MONTGOMERY, R. A. et al. Hla in transplantation. *Nature Reviews Nephrology*, v. 14, n. 9, p. 558–570, 2018. ISSN 1759-507X. Disponível em: <https://doi.org/10.1038/s41581-018-0039-x>.

17 PETERSDORF, E. W. Hla mismatching in transplantation. *Blood*, American Society of Hematology, v. 125, n. 7, p. 1058–1059, 2015. ISSN 0006-4971.

18 BARBOSA, P.; MÜLLE, R. Diretrizes Brasileiras para diagnóstico, tratamento e prevenção da febre reumática. *Arq Bras Cardiol*, scielo, n. Suppl 4, p. 127–47.

19 Eppendorf International. *Laboratory Equipment, supplies & Services*. s.d. Disponível em: <https://www.eppendorf.com/>. Acesso em: 05 de janeiro 2019.

20 WORDSWORTH, P. Pcr-sso typing in hla-disease association studies. *International Journal of Immunogenetics*, Wiley Online Library, v. 18, n. 1-2, p. 139–146, 1991.

21 Sarstedt. *Sarstedt*. s.d. Disponível em: <https://www.sarstedt.com/>. Acesso em: 05 de janeiro 2019.

22 CORPORATION, L. *Luminex Corporation | Complexity Simplified*. 2018. Disponível em: <https://www.luminexcorp.com/>. Acesso em: 08 mar. 2018.

23 BOYD, J. Robotic laboratory automation. *Science*, American Association for the Advancement of Science, v. 295, n. 5554, p. 517–518, 2002.

24 GIBBON, G. A. A brief history of LIMS. *Laboratory Automation and Information Management*, v. 32, n. 1, p. 1–5, 1996. ISSN 1381141X.

25 SOFT, H. *Hyla - Soft*. 2018. Disponível em: <https://www.hylasoft.com/>. Acesso em: 08 mar. 2018.

26 HOLECZEKC, L.; Core UI Team. *Free Bootstrap Admin Template - CoreUI*. 2018. Disponível em: <https://coreui.io/>. Acesso em: 30 dez. 2018.

27 Ruby Community. *Ruby, A Programmer's Best Friend*. s.d. Disponível em: <https://www.ruby-lang.org/>. Acesso em: 01 de novembro 2018.

28 Ruby Community. *RubyGems.org, your community gem host*. s.d. Disponível em: <https://rubygems.org/>. Acesso em: 01 de novembro 2018.

29 Ruby on Rails Community. *Rails*. s.d. Disponível em: <http://rubyonrails.org/>. Acesso em: 09 de abril 2018.

30 AkitaOnRails. *Why it is just lazy to bad-mouth Ruby on Rails*. s.d. Disponível em: <http://www.akitaonrails.com/2017/08/03/why-is-it-just-lazy-to-bad-mouth-ruby-on-rails>. Acesso em: 05 de janeiro 2019.

31 LINDSAAR, M. *Why Ruby on Rails is still the best choice?* s.d. Disponível em: <https://reinteractive.com/posts/320-why-ruby-on-rails-is-still-the-best-choice>. Acesso em: 05 de janeiro 2019.

32 EquiValent. *Is Rails still relevant in 2018?* s.d. Disponível em: <https://blog.eq8.eu/article/is-rails-still-relevant-in-2018.html>. Acesso em: 05 de janeiro 2019.

33 GRIFFEL, M. *10 Reasons Beginners Should Learn Ruby on Rails*. s.d. Disponível em: <https://learn.onemonth.com/10-reasons-beginners-learn-ruby-rails/>. Acesso em: 05 de janeiro 2019.

34 CHEKALIN, D.; GRITSAY, K. *7 Reasons to Use Ruby on Rails for Your Startup*. s.d. Disponível em: <https://www.codica.com/blog/why-build-startup-with-ruby-on-rails/>. Acesso em: 05 de janeiro 2019.

35 PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, ACM, New York, NY, USA, v. 15, n. 12, p. 1053–1058, dez. 1972. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/361598.361623>.

36 ASTELS, D. *Test-Driven Development: A Practical Guide: A Practical Guide*. [S.l.]: Prentice Hall, 2003. ISBN 9780131016491.

37 JANZEN, D.; SAIEDIAN, H. Test-driven development concepts, taxonomy, and future direction. *Computer*, v. 38, n. 9, p. 43–50, Sep. 2005. ISSN 0018-9162.

38 KARAC, I.; TURHAN, B. What do we (really) know about test-driven development? *IEEE Software*, v. 35, n. 4, p. 81–85, July 2018. ISSN 0740-7459.

39 MARTIN, R. C. *The Clean Coder: A Code of Conduct for Professional Programmers*. [S.l.]: Prentice Hall, 2011. ISBN 0137081073.

40 HARTL, M. *Ruby on Rails Tutorial: Learn Web Development with Rails*. Michael Hartl, 2016. Disponível em: <https://www.railstutorial.org/book>.

41 DATE, C. J. *An Introduction to Database Systems*. 8 edition. ed. Boston: Pearson, 2003. ISBN 978-0-321-19784-9.

42 PostgreSQL. *PostgreSQL: The World's most advanced Open Source relational database*. s.d. Disponível em: <https://www.postgresql.org/>. Acesso em: 01 de novembro 2018.

43 Ruby on Rails Community. *Ruby on Rails Guides*. s.d. Disponível em: <https://edgeguides.rubyonrails.org/>. Acesso em: 01 de novembro 2018.

44 PLATAFORMATEC. *Flexible authentication solution for Rails with Warden - plataformatec/devise*. s.d. Disponível em: <https://github.com/plataformatec/devise>. Acesso em: 30 dez. 2018.

45 HALE, C. *bcrypt-ruby is a Ruby binding for the OpenBSD bcrypt() password hashing algorithm, allowing you to easily store a secure hash of your users' passwords. - codahale/bcrypt-ruby*. s.d. Disponível em: <https://github.com/codahale/bcrypt-ruby>. Acesso em: 30 dez. 2018.

46 HAWTHORN, J. *Soft delete for ActiveRecords done write*. s.d. Disponível em: <https://github.com/jhawthorn/discard>. Acesso em: 30 dez. 2018.

47 CANCANCOMMUNITY. *CanCanCommunity/cancancan - The authorization Gem for Ruby on Rails*. s.d. Disponível em: <https://coreui.io/>. Acesso em: 30 dez. 2018.

48 MAROHNIC, J. *Shrine*. s.d. Disponível em: <https://github.com/shrinerb/shrine/>. Acesso em: 09 de abril 2018.

APÊNDICE A – Construção da aplicação base

Esse apêndice fornece instruções técnicas detalhadas de como construir a aplicação rails que servirá de base para os módulos. A aplicação foi nomeada de *base_app* e consiste de uma aplicação padrão rails com alguns ajustes.

Todas as instruções são para o ambiente Linux (qualquer distribuição) e supõem que já foi realizada a instalação do *Ruby* e da *gem Rails*.

A.1 Criação da aplicação base

No terminal, no diretório da pasta escolhida, use os seguintes comandos para criar a aplicação base e entrar na pasta criada:

```
1 rails new base_app --database=postgresql -T
2 cd base_app
```

Dessa forma será automaticamente criado toda a estrutura de pastas e arquivos básica, típica de uma aplicação *Rails*. Como gerenciador de bancos de dados é utilizado o *PostgreSQL* (diferente do padrão, que é o *SQLite*), mais indicado para trabalhar com aplicações extensas e complexas. A diretriz "-T" é para que não seja instalado o *framework* de testes padrão do *Rails*.

A.2 Criação da pasta para salvar os módulos

Os módulo serão salvos em uma pasta denominada "*engines*" dentro do diretório da aplicação. O comando apresentado abaixo cria a pasta "*engines*", com um arquivo em branco. A finalidade desse arquivo é garantir que a pasta será salva no repositório remoto (por padrão pastas vazias não são salvas).

```
1 mkdir engines
2 touch engines/.keep
```

A.3 Configuração do git

Como essa aplicação é só uma base para os módulos, não é necessário a criação de um repositório para *backup* desse código. Porém, caso julgue necessário, por padrão o *rails* já iniciou o versionamento *git* na pasta da aplicação. Só é preciso uma pequena configuração para que todos os módulos não sejam incluídos (recomenda-se que cada um

esteja em seu próprio repositório).

Para isso edite o arquivo *.gitignore* que se encontra na pasta da aplicação, adicionando as entradas apresentadas a seguir no final.

```
# .gitignore
# ...
engines/*
!engines/.keep
```

A.4 **Configurações finais**

A.4.1 Atualização do arquivo README.md

É recomendado, atualização do arquivo *README.md* com detalhes sobre a aplicação que se está criando.

A.4.2 Adição de gems para depuração de código

Como mencionado anteriormente, essa aplicação será somente uma base e só sofrerá pequenas modificações realizadas automaticamente pelo *Rails* durante a interação com os módulos. Por esse motivo a inclusão de ferramentas de depuração é opcional. Caso necessário, recomenda-se que sejam utilizadas as seguintes *gems*:

- *pry*:
  - Versão: <https://rubygems.org/gems/pry>
  - Instruções: <https://github.com/pry/pry>
- *pry-byebug*:
  - Versão: <https://rubygems.org/gems/pry-byebug>
  - Instruções: <https://github.com/deivid-rodriguez/pry-byebug>
- *pry-rails*:
  - Versão: <https://rubygems.org/gems/pry-rails>
  - Instruções: <https://github.com/rweng/pry-rails>

Verifique sempre a última versão disponível no site da *gem*. Nesse endereço também podem ser vistas instruções de como depurar o código utilizando as bibliotecas instaladas.

Para finalizar é preciso adicionar na *Gemfile* as seguintes entradas:

```
# Gemfile
# ...
group :development, :test do
#...
  gem 'pry-byebug', '~> 3.6'
end

group :development do
#...
  gem 'pry-rails', '~> 0.3.6'
end

#...
gem 'pry', '~> 0.11.3'
```

Instale as *gems* executando o comando a seguir no terminal.

```
bundle
```

A.4.3 Configurações finais do *git*

Para finalizar, vamos criar um ponto de *commit* para salvar todas as alterações feitas até agora. No terminal execute o comando apresentado a seguir.

```
git add .
git commit -m 'Primeiro commit'
```

É recomendado a criação de um repositório remoto. Para isso existem várias opções possíveis (*Github*, *Gitlab*, *Bitbucket*, etc).

Crie um e siga as instruções fornecidas para as configurações iniciais. Em seguida envie o código para lá com o comando apresentado a seguir no terminal.

```
git push
```

APÊNDICE B – Construção inicial de um módulo

Esse apêndice fornece instruções técnicas detalhadas de como criar a estrutura básica de um módulo. Todos os módulos devem seguir as instruções apresentadas. Findada essa etapa o desenvolvedor pode continuar incorporando as funcionalidades específicas que deseja.

Todas as instruções são para o ambiente Linux e supõem que já foi realizada a instalação do *Ruby*, da *gem Rails* e que já foi realizado todo o procedimento descrito no Apêndice A.

B.1 Construção inicial de um módulo

O *rails* tem um comando próprio para criação da estrutura mínima de diretórios e arquivos necessário para a criação de módulos. Dentro da raiz do diretório da aplicação principal, digite o seguinte comando no terminal:

```
1 rails plugin new engines/dat-base_engine --mountable --dummy-path=spec/
    dummy --database=postgresql --skip-gemfile-entry -T
```

É recomendado que seja utilizado um *namespace* para organizar o código em grupos lógicos e prevenir possíveis conflitos de nomes (classes, métodos, etc), visto que a aplicação pode possuir muitos módulos. No exemplo foi utilizado "*dat*".

O nome da aplicação utilizado é *base_engine*. O desenvolvedor pode escolher o nome que julgar necessário.

As *engines* são como mini-aplicações, e por esse motivo, serão criados arquivos e diretórios para que seja possível executar testes independente do sistema como um todo. Por isso é necessário especificar onde estarão esses arquivos pois caso contrário será preciso executar etapas adicionais de configuração. Isso é feito com a diretriz "*- -dummy-path*". Nela informamos onde será realizada a instalação dessa estrutura tendo como base o diretório do módulo, no caso será "*spec/dummy*".

B.2 O arquivo *Gemspec*

Nesse arquivo são colocadas informações importantes sobre o módulo e deve ser todo preenchido, pois a aplicação não irá executar se essas informações estiverem em branco ou com as informações geradas automaticamente quando da criação dos arquivos.

Veja o exemplo de um arquivo a seguir.

```
1 $:.push File.expand_path("../lib", __FILE__)
```

```
2
3  # Maintain your gem's version:
4  require "dat/base_engine/version"
5
6  # Describe your gem and declare its dependencies:
7  Gem::Specification.new do |s|
8    s.name        = "dat-base_engine"
9    s.version     = Dat::BaseEngine::VERSION
10   s.authors     = ["Jonh Doe"]
11   s.email       = ["jonh.doe@gmail.com"]
12   s.homepage    = "http://www.somepage.com"
13   s.summary     = "Summary of Dat::BaseEngine."
14   s.description = "Description of Dat::BaseEngine."
15   s.license     = "MIT"
16
17   s.files = Dir["{app,config,db,lib}/**/*", "MIT-LICENSE", "Rakefile", "
       README.md"]
18
19   s.add_dependency "rails", "~> 5.2.1"
20
21   s.add_development_dependency "pg"
22 end
```

B.3 Adição de *gems* para depuração de código

Para depurar o código, recomenda-se que sejam utilizadas as *gems* mencionadas a seguir. Maiores informações sobre a utilização podem ser encontradas nos *links* de instruções.

- *pry*:
  - Versão: <https://rubygems.org/gems/pry>
  - Instruções: <https://github.com/pry/pry>
- *byebug*:
  - Versão: <https://rubygems.org/gems/byebug>
  - Instruções: <https://github.com/deivid-rodriguez/byebug>
- *pry-byebug*:
  - Versão: <https://rubygems.org/gems/pry-byebug>
  - Instruções: <https://github.com/deivid-rodriguez/pry-byebug>

- *pry-rails*:
  - Versão: <https://rubygems.org/gems/pry-rails>
  - Instruções: <https://github.com/rweng/pry-rails>

B.4 Operações nos bancos de dados

Para facilitar o processo de implementação de mudanças nas tabelas do banco de dados, o *rails* oferece um recurso denominado *Migrations*. O objetivo é permitir que as mudanças sejam escritas usando a linguagem *Ruby* ao invés de SQL, assim o desenvolvedor não precisa se envolver com detalhes das diferentes implementações de cada distribuição de banco de dados (SQLite, PostgreSQL, MySQL). Dessa forma o desenvolvedor só precisa especificar em um arquivo o código *rails* com as mudanças que quer implementar, e executar um comando no terminal ("*rails db:migrate*") para que as mesmas sejam efetivamente aplicadas. Maiores detalhes podem ser visto no guia do RoR [43].

Como as *engines* são vistas como "mini-aplicações", as evoluções realizadas dentro desse contexto são consideradas como locais, ou seja, ao executar o *script* de aplicação das migrações, será criado um novo banco de dados local que não terá relação com o da aplicação.

Para realizar mudanças no contexto da aplicação, é necessário incluir o código a seguir no arquivo localizado dentro do diretório "*dat-base_engine/lib/dat/base_engine/engine.rb*". Esse arquivo é sempre carregado na inicialização da *engine* e o código copia automaticamente os arquivos de migrações para a aplicação principal. Assim quando for executado o comando para efetivar as mudanças na aplicação principal, as mudanças requeridas pela *engine* serão também efetivadas.

```
1  module Dat
2    module BaseEngine
3      class Engine < ::Rails::Engine
4        isolate_namespace Dat::BaseEngine
5
6        paths["app/views"] << "app/views/dat"
7
8        initializer :append_migrations do |app|
9          unless app.root.to_s.match(root.to_s)
10           config.paths["db/migrate"].expanded.each do |p|
11             app.config.paths["db/migrate"] << p
12           end
13         end
14       end
15
16     end
```

```
17   end
18 end
```

B.5 Configuração do ambiente de teste

De modo a facilitar a escrita e execução de testes, são utilizadas as *gems* apresentadas a seguir. Esse conjunto de bibliotecas permite com que sejam facilmente escritos testes para componentes internos, como modelos e controles, e também testes para componentes de interface gráfica (através de um simulador de *browser*). Assim é possível testar inclusive ações como cliques em botões e preenchimento de formulários de forma simples.

Exemplos de testes diversos são apresentados no apêndice D.

- *rspec-rails*:
  - Versão: <https://rubygems.org/gems/rspec-rails>
  - Instruções: <https://github.com/rspec/rspec-rails>
- *capybara*:
  - Versão: <https://rubygems.org/gems/capybara>
  - Instruções: <https://github.com/teamcapybara/capybara>
- *selenium-webdriver*:
  - Versão: <https://rubygems.org/gems/selenium-webdriver>
  - Instruções: <https://github.com/SeleniumHQ/selenium/tree/master/rb>
- *puma*:
  - Versão: <https://rubygems.org/gems/puma>
  - Instruções: <https://github.com/puma/puma>
- *factory_bot_rails*:
  - Versão: <https://rubygems.org/gems/factory_bot_rails>
  - Instruções: <https://github.com/thoughtbot/factory_bot_rails>
- *database_cleaner*:
  - Versão: <https://rubygems.org/gems/database_cleaner>
  - Instruções: <https://github.com/DatabaseCleaner/database_cleaner>

- *faker*:
  - Versão: <https://rubygems.org/gems/faker>
  - Instruções: <https://github.com/stympy/faker>

Obs: A *gem faker* é opcional. Ela facilita a criação de dados aleatórios para os testes (nomes, endereços, emails, etc) mas é possível realizar os testes utilizando combinações de *strings* utilizando somente o *Ruby*.

É preciso adicionar as *gems* ao arquivo de configuração da *engine* (*Gemspec*), conforme o exemplo a seguir.

```
# base_app/engines/dat-base_engine/dat-base_engine.gemspec
# ...
s.add_development_dependency "rspec-rails", "~> 3.8"
s.add_development_dependency "capybara", "~> 3.7", ">= 3.7.2"
s.add_development_dependency "selenium-webdriver", "~> 3.14"
s.add_development_dependency "puma", "~> 3.12"
s.add_development_dependency "factory_bot_rails", "~> 4.11"
s.add_development_dependency "database_cleaner", "~> 1.7"
s.add_development_dependency "faker",  "~> 1.9", ">= 1.9.1"
```

Para finalizar é preciso instalar as *gems* de teste executando no terminal o comando a seguir no diretório raiz da *engine*.

```
# engines/dat-base_engine
bundle
```

B.6 Adicionar a referência a *engine* na aplicação principal

A aplicação principal não detecta automaticamente a *engine*. Para integrá-la a aplicação principal é preciso indicar o diretório onde estão seus arquivos. Para isso adicione a entrada a seguir ao arquivo *Gemfile* da aplicação principal.

```
#...
gem 'dat-base_engine', path: 'engines/dat-base_engine'
```

Em seguida execute no terminal comando para instalar a integração na aplicação principal.

```
# diretorio da aplicacao principal
bundle
```

B.7 Configurações do ambiente de teste

Após instalação, é preciso inicializar o ambiente de teste para que sejam criados os arquivo necessários de configuração e também sejam instalados os comandos que irão executar os testes em si. Para isso execute o comando a seguir no terminal (no diretório da *engine*).

```
rails g rspec:install
```

Dentro do diretório *spec* serão criados os arquivos de configuração. Recomenda-se que sejam implementas as configurações apresentadas a seguir. Elas servem para configurar parâmetros básicos de integração entre as *gems* e também o ambiente de simulação de *browser*.

```
# ARQUIVO spec/rails_helper.rb
#
# Nao esqueca de atualizar as entradas com o nome correto da sua engine onde necessario!
#
require 'spec_helper'
ENV['RAILS_ENV'] ||= 'test'
require File.expand_path('../../../../config/environment', __FILE__)
abort("The Rails environment is running in production mode!") if Rails.env.production?
require 'rspec/rails'
require 'capybara/rspec'
require 'selenium-webdriver'
require 'factory_bot_rails'
require 'database_cleaner'

include Warden::Test::Helpers

Capybara.server = :puma, { Silent: true }

Capybara.register_driver :headless_chrome do |app|
  options = ::Selenium::WebDriver::Chrome::Options.new
  options.add_argument('--headless')
  options.add_argument('--no-sandbox')
  options.add_argument('--disable-dev-shm-usage')
  options.add_argument('--window-size=1400,1400')

  Capybara::Selenium::Driver.new(
    app,
    browser: :chrome,
    options: options
  )
end
```

```
32
33 Capybara.javascript_driver = :headless_chrome
34
35 Capybara.default_max_wait_time = 10
36
37 # Caso sua engine dependa da engine de autenticacao (Core), adicione a
       linha de codigo a seguir
38 # Isso e necessario para que testes sejam realizados com usuarios logados (
       admin ou nao)
39 Dir[Rails.root.join('engines/dat-core/spec/factories/**/*.rb')].each { |f|
       require f }
40
41 begin
42   ActiveRecord::Migration.maintain_test_schema!
43 rescue ActiveRecord::PendingMigrationError => e
44   puts e.to_s.strip
45   exit 1
46 end
47
48 RSpec.configure do |config|
49   # Nao esqueca de trocar o 'dat-base_engine' pelo nome da sua engine!
50   config.fixture_path = "#{::Rails.root}/engines/dat-base_engine/spec/
       fixtures"
51   config.use_transactional_fixtures = false
52   config.infer_spec_type_from_file_location!
53   config.filter_rails_from_backtrace!
54
55   config.before(:suite) do
56     DatabaseCleaner.strategy = :transaction
57     DatabaseCleaner.clean_with(:truncation)
58   end
59
60   config.before(:each, :js => true) do
61     DatabaseCleaner.strategy = :truncation
62   end
63
64
65   config.before(:each) do
66     DatabaseCleaner.start
67   end
68
69   config.after(:each) do
70     DatabaseCleaner.clean
71   end
72
73   config.include FactoryBot::Syntax::Methods
74
```

```
  # Nao esqueca de trocar o 'BaseEngine' pelo nome da sua engine!
  config.include Dat::BaseEngine::Engine.routes.url_helpers

  # So adicione essas 3 linhas de codigo se sua engine depender da engine Core
  config.include Devise::Test::ControllerHelpers, type: :controller
  config.include Devise::Test::ControllerHelpers, type: :view
  config.include Devise::Test::IntegrationHelpers, type: :feature
end
```

```
# ARQUIVO spec/spec_helper.rb
#
RSpec.configure do |config|
  config.expect_with :rspec do |expectations|
    expectations.include_chain_clauses_in_custom_matcher_descriptions = true
  end

  config.mock_with :rspec do |mocks|
    mocks.verify_partial_doubles = true
  end

  config.shared_context_metadata_behavior = :apply_to_host_groups
  config.filter_run_when_matching :focus
  #  config.example_status_persistence_file_path = "spec/examples.txt"
  config.disable_monkey_patching!
  if config.files_to_run.one?
    config.default_formatter = "doc"
  end
  # config.profile_examples = 10
  config.order = :random
  Kernel.srand config.seed
end
```

```
# ARQUIVO .rspec.rb
# (localizado no diretorio raiz da engine)
--color
--format documentation
```

B.8 **Configurações finais**

B.8.1 Atualização do arquivo *README.md*

É recomendado, atualização do o arquivo *README.md* com detalhes sobre a aplicação que se está criando.

B.8.2 Configurações finais do *git*

Para finalizar, vamos criar um ponto de *commit* para salvar todas as alterações feitas até agora. No terminal execute o comando apresentado a seguir.

```
1 git add .
2 git commit -m 'Primeiro commit'
```

É recomendado a criação de um repositório remoto. Para isso existem várias opções possíveis (*Github*, *Gitlab*, *Bitbucket*, etc).

Crie um e siga as instruções fornecidas para as configurações iniciais. Em seguida envie o código para lá com o comando apresentado a seguir no terminal.

```
1 git push
```

APÊNDICE C – Construção de um módulos com dependências entre si

Esse apêndice fornece instruções técnicas detalhadas de como criar um módulo que precise ter usuários autenticados pelo módulo *Core*. Também serve para criação de módulos que precisem criar relações entre tabelas de outros módulos, ou seja, dependa de outros módulos para funcionar. Um exemplo desse comportamento na atual aplicação é o módulo de microplacas - precisa de usuários autenticados e tem relações com tabelas dos módulos de doadores e do módulo de análises.

Todas as instruções são para o ambiente Linux e supõem que já foi realizada a instalação do *Ruby*, da *gem Rails* e que já foi realizado todo o procedimento descrito nos Apêndices A e B.

C.1 Adição das dependências e da *gem Deface*

Para especificar as depedências do módulo, é preciso declará-las no arquivo *gemspec*. Por exemplo, a seguir é mostrado como declarar a dependência com o módulo *Core*.

É importante também adicionar um *gem* denominada *Deface*. Essa *gem* tem como intuito permitir a inserção de trechos de código para customizar *templates* do *Rails* especificados nos arquivos *.erb* em pontos pré-determinados.

- *deface*:
  - Versão: <https://rubygems.org/gems/deface>
  - Instruções: <https://github.com/spree/deface>

```
# app_name/engines/dat-dep_engine/dat-dep_engine.gemspec
# ...


Gem::Specification.new do |s|
# ...

  s.add_dependency "dat-core"
  s.add_dependency "deface", "~> 1.3"
end
```

Não esquecer que *gems* que não são de teste precisam também ser “requeridas” no arquivo *dep_engine.rb*, conforme exemplo a seguir.

```
# app_name/engines/dat-dep_engine/lib/dat/dep_engine.rb
#...
```

```
require 'deface'
#...

module Dat
  module BaseEngine
  end
end
```

Como a *engine Core* está instalada localmente, é preciso indicar o caminho dela, de modo análogo como foi feito no Apêndice B. Esse processo deve ser repetido para todas as *engines* locais.

```
# app_name/engines/dat-dep_engine/Gemfile
# ... other stuff...

gem 'dat-core', path: '../dat-core'
```

C.2 Mudar o arquivo de rotas

Como estamos construindo esse módulo em cima do módulo *Core*, para facilitar é recomendado estender as rotas do *Core* em vez de implementar as próprias. Para isso basta alterar o arquivo de rotas para coincidir com o da *engine* Core.

```
# app_name/engines/dat-dep_engine/config/routes.rb

Dat::Core::Engine.routes.draw do
end
```

Para finalizar é preciso instalar as *gems* executando no terminal o comando a seguir no diretório raiz da aplicação.

```
#app_name/
bundle
```

C.3 Criação das pastas para os "*decorators*" e os "*overrides*"

Os chamados "*decorators*" são os arquivos que iremos usar para inserir códigos extras em classes adicionando funcionalidades ou relações entre tabelas. Os "*overrides*" são os arquivos que iremos usar para inserir elementos na interface gráfica em pontos pré-determinados.

No diretório raíz da *engine*:

```
mkdir -p app/decorators/controllers
touch app/decorators/controllers/.keep
```

```
mkdir app/decorators/models
touch app/decorators/models/.keep
mkdir app/overrides
touch app/overrides/.keep
```

Ou em uma linha:

```
mkdir -p app/decorators/controllers && touch app/decorators/controllers/.keep && mkdir app/decorators/models && touch app/decorators/models/.keep && mkdir app/overrides && touch app/overrides/.keep
```

C.4 Código para carregar automaticamente os "*decorators*"

Para que os "*decorators*" criados sejam inicializados junto com a *engine*, é preciso incluir o código abaixo no arquivo *engine.rb*. Conforme mencionando anteriormente, esse arquivo é carregado na inicialização do sistema.

```
# base_app/engines/dat-dep_engine/lib/dat/dep_engine/engine.rb
module Dat
  module DepEngine
    class Engine < ::Rails::Engine
      # ...

      config.to_prepare do
        Dir.glob(Engine.root.join("app", "decorators", "**", "*_decorator*.rb")) do |c|
          Rails.configuration.cache_classes ? require(c) : load(c)
        end
      end

    end
  end
end
```

Os arquivos devem ser nomeados de acordo com o seguinte o padrão: *nome_decorator.rb*.

C.5 Arquivo de permissões

Para criar o arquivo de permissões específicas para o módulo, é preciso adicionar ao arquivo principal que se encontra no módulo *Core*, conforme mencionado no capítulo 4.6.3. Basta para isso criar o arquivo *decorator* necessário:

```
mkdir app/decorators/models/dat/
touch app/decorators/models/dat/abilities_decorator.rb
```

Com o seguinte código:

```
# dep-dep_engine/app/decorators/models/dat/abilities_decorator.rb
require 'cancancan'

module Dat
  module DepEngine
    class AbilityDecorator
      include CanCan::Ability

        def initialize(user)
          # inserir permissoes aqui!
        end
      end
    end
  end

# Registra as permissoes definidas.
# Nao esquecer de mudar o DepEngine para o nome da engine
Dat::Ability.register_ability(Dat::DepEngine::AbilityDecorator)
```

C.6 Exemplos de "*decorators*" e os "*overrides*"

Os exemplos a seguir são da *engine Microplate*, e servem para ilustrar os conceito apresentado anteriormente.

Acrescentando uma relação a tabela *Analysis*:

```
# base_app/engines/dat-microplate/app/decorators/analysis_decorator.rb
Dat::LabBase::Analysis.class_eval do
  has_many :microplate_data,
    class_name: "Dat::Microplate::MicroplateDatum",
    foreign_key: "dat_lab_base_analysis_id",
    inverse_of: :analysis,
    dependent: :destroy
end
```

Para definição de um "*override*" é preciso utilizar um ponto de entrada, que é uma "marca" no arquivo *.erb* que será usada como referência.

Na arquivo que define o menu lateral na *engine Core*, temos:

```
<div class="sidebar">
  <nav class="sidebar-nav">
    <ul class="nav">
      <li class="nav-item">
        <%= link_to dat.root_path, class: "nav-link" do %>
          <i class="fas fa-tachometer-alt fa-lg"></i> Dashboard
```

```
        <% end %>
      </li>
      <li data-dat-hook='users_nav' class="nav-title">Análises</li>
      <% if current_user.admin? %>
        <li class="nav-title">Admin</li>
        <li  data-dat-hook='admin_nav' class="nav-item">
          <%= link_to dat.admin_users_path, class: "nav-link" do %>
            <i class="fas fa-user-friends fa-lg"></i> Usuários
          <% end %>
        </li>
        <li class="nav-title">Arquivo</li>
        <li  data-dat-hook='admin_archive_nav' class="nav-item">
          <%= link_to dat.admin_archived_users_path, class: "nav-link" do %>
            <span class="fa-stack mr-1">
              <i class="fas fa-ban fa-stack-2x"></i>
              <i class="fas fa-user-friends fa-stack-1x"></i>
            </span>
            Usuários
          <% end %>
        </li>
      <% end %>
    </ul>
  </nav>
  <button class="sidebar-minimizer brand-minimizer" type="button"></button>
</div>
```

Os chamados pontos de entrada estão definidos nos "*li*" como *data-dat-hook*. De modo a inserir alguma nova entrada no menu deve-se usar esse pontos como referência.

Exemplo para adicionar um link para as microplacas no menu de usuários:

```
# base_app/engines/dat-microplate/app/overrides/add_microplate_link_to_nav.rb
Deface::Override.new(:virtual_path => "dat/shared/_right_sidebar",
  :original => '9a7c417ba0082a0da5ba411a4b13a826fc74b60a',
  :name => "add_microplate_link_to_nav",
  :insert_after => "[data-dat-hook='users_nav']",
  :partial => "overrides/microplate_link",
  :sequence => 1,
  :namespaced => true)
```

Em seguida, deve-se adicionar um arquivo de *view* com o trecho do código que deseja-se inserir:

```
<! -- base_app/engines/dat-microplate/app/views/dat/overrides/_microplate_link.html.erb
<li class="nav-item">
  <%= link_to dat.microplate_data_path, class: "nav-link" do %>
    <i class="fas fa-th fa-lg pl-1"></i> Microplacas
```

```
5   <% end %>
6 </li>
```

Basta adaptar os exemplos a realidade da *engine* que está se criando.

APÊNDICE D – Exemplos de Testes

Esse apêndice fornece alguns exemplos dos testes escritos utilizando o *framework* de teste, composto pelas *gems* apresentadas no Apêndice B.5.

O exemplo a seguir testa as validações feitas para tabela *Users*. O modelo deve impedir a criação de registros com dados faltantes, como nome de usuário ou senha. Também estão sendo testadas a unicidade de certos campos na tabela, acessos com senha incorreta e também o tipo de usuário (comum ou administrador).

```
require 'rails_helper'

module Dat
  RSpec.describe User, type: :model do
    let(:user) { FactoryBot.build(:user) }

    describe "Basic User" do
      it "has a valid factory" do
        expect(FactoryBot.build(:user)).to be_valid
      end

      it "is invalid without an email" do
        user.email = nil
        expect(user).to_not be_valid
      end

      it "is invalid without a username" do
        user.username = nil
        expect(user).to_not be_valid
      end

      it "is invalid with a username that contains @" do
        user.username = "test@example.com"
        expect(user).to_not be_valid
      end

      it "is invalid with a username that is already on database" do
        FactoryBot.create(:user, username: "jonhdoe")
        user.username = "jonhdoe"
        expect(user).to_not be_valid
      end

      it "is invalid without a password" do
        user.password = nil
        expect(user).to_not be_valid
      end
```

```
37
38      it "is invalid with different password and password confirmation" do
39        user.password = "password"
40        user.password_confirmation = "drowssap"
41        expect(user).to_not be_valid
42      end
43
44      it "default role is user" do
45        expect(user.role).to eq("user")
46      end
47    end
48
49    describe "Admin User" do
50      let(:admin_user) { FactoryBot.build(:admin) }
51
52      it "has a valid factory" do
53        expect(FactoryBot.build(:admin)).to be_valid
54      end
55
56      it "the role is admin" do
57        expect(admin_user.role).to eq("admin")
58      end
59    end
60  end
61 end
```

O exemplo a seguir testa aspectos da interface gráfica. No caso a tela responsável pela criação de novas análises. Estão sendo testados o acesso a página, a existência de campos no formulário, o preenchimento e a submissão do mesmo.

```
1  require 'rails_helper'
2
3  module Dat
4    RSpec.describe "New Analysis", type: :feature do
5      before(:each) do
6        @routes = Dat::Core::Engine.routes
7      end
8
9      context "admin sign in" do
10       before(:each) do
11         @admin = FactoryBot.create(:admin)
12         sign_in @admin
13
14         visit dat.new_analysis_path
15       end
16
17       it "has the code input" do
18         xpath = "//*[input[contains(@class,'form-control') and contains(
```

```
@name, '[code]')]]"
19        expect(page).to have_xpath(xpath)
20      end
21
22      it "has the description input" do
23        xpath = "//*[textarea[contains(@class,'form-control') and contains(
@name, '[description]')]]"
24        expect(page).to have_xpath(xpath)
25      end
26
27      it "has the remarks input" do
28        xpath = "//*[textarea[contains(@class,'form-control') and contains(
@name, '[remarks]')]]"
29        expect(page).to have_xpath(xpath)
30      end
31
32      it "has the select analysis_type input" do
33        xpath = "//*[select[contains(@class,'form-control') and contains(
@name, '[dat_lab_base_analysis_type_id]')]]"
34        expect(page).to have_xpath(xpath)
35      end
36
37      it "create a analysis successfully" do
38        analysis_type = FactoryBot.create(:analysis_type)
39        analysis = FactoryBot.build(:analysis)
40
41        visit dat.new_analysis_path
42
43        fill_in 'analysis_code', with: analysis.code
44        fill_in 'analysis_description', with: analysis.description
45        fill_in 'analysis_remarks', with: analysis.remarks
46        select analysis_type.name, from: '
analysis_dat_lab_base_analysis_type_id'
47
48        click_button 'Criar Analise'
49
50        sleep(1)
51
52        analysis_created = Dat::LabBase::Analysis.last
53        expect(page).to have_current_path(dat.analysis_path(
analysis_created))
54
55        expect(analysis_created.code).to eq(analysis.code)
56        expect(analysis_created.description).to eq(analysis.description)
57        expect(analysis_created.remarks).to eq(analysis.remarks)
58        expect(analysis_created.analysis_type.name).to eq(analysis_type.
name)
```

```
      end
    end
  end
end
```

O exemplo a seguir mostra parte do arquivo de teste do modelo de análises, em particular o teste relativo as relações entre tabelas do banco de dados.

```

require 'rails_helper'

module Dat
  RSpec.describe LabBase::Analysis, type: :model do
    let(:analysis) { FactoryBot.build(:analysis) }

    describe "Basic Analysis" do

      #...
      # other tests...

      it "belongs_to analysis_type" do
        assc = described_class.reflect_on_association(:analysis_type)
        expect(assc.macro).to eq(:belongs_to)
      end

      it "belongs_to user" do
        assc = described_class.reflect_on_association(:user)
        expect(assc.macro).to eq(:belongs_to)
      end
    end
  end
end
```

APÊNDICE E – Gems utilizadas

Esse apêndice apresenta as gems que foram utilizadas e suas respectivas versões.

Tabela 1 - Gems utilizadas na aplicação

Nome da Gem	Versão	Repositório
ajax-datatables-rails	~> 1.0	<https://github.com/jbox-web/ajax-datatables-rails>
bootsnap	~> 1.3	<https://github.com/Shopify/bootsnap>
bootstrap	~> 4.1	<https://github.com/twbs/bootstrap-rubygem>
byebug	~> 10.0	<https://github.com/deivid-rodriguez/byebug>
cancancan	~> 2.2	<https://github.com/CanCanCommunity/cancancan>
capybara	~> 3.7	<https://github.com/teamcapybara/capybara>
cocoon	~> 1.2	<https://github.com/nathanvda/cocoon>
dat_gretel	~> 4.0	<https://github.com/marciodat/dat_gretel>
database_cleaner	~> 1.7	<https://github.com/DatabaseCleaner/database_cleaner>
deface	~> 1.3	<https://github.com/spree/deface>
discard	~> 1.0	<https://github.com/jhawthorn/discard>
factory_bot_rails	~> 4.11	<https://github.com/thoughtbot/factory_bot_rails>
faker	~> 1.9	<https://github.com/stympy/faker>
devise	~> 4.4	<https://github.com/plataformatec/devise>
jquery-rails	~> 4.3	<https://github.com/rails/jquery-rails>
prawn-labels	~> 1.2	<https://github.com/jordanbyron/prawn-labels>
pry	~> 0.11.3	<https://github.com/pry/pry>
pry-byebug	~> 0.3.6	<https://github.com/deivid-rodriguez/pry-byebug>
puma	~> 3.12	<https://github.com/puma/puma>
rspec-rails	~> 3.8	<https://github.com/rspec/rspec-rails>
rspec-retry	~> 0.4.5	<https://github.com/NoRedInk/rspec-retry>
sass-rails	~> 5.0	<https://github.com/rails/sass-rails>
selenium-webdriver	~> 3.14	<https://github.com/SeleniumHQ/selenium/tree/master/rb>
shrine	~> 2.12	<https://github.com/shrinerb/shrine>
turbolinks	~> 5.0	<https://github.com/turbolinks/turbolinks>

Legenda: ~> maior que X.X porém menor que (X+1).0

Fonte: O autor, 2018