UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CURSO DE ESPECIALIZAÇÃO EM TECNOLOGIA JAVA

ADAIR PERDOMO FALCÃO

sistema de apoio a cobrança de dívida ativa

MONOGRAFIA DE ESPECIALIZAÇÃO

PATO BRANCO
2017

ADAIR PERDOMO FALCÃO

SISTEMA DE APOIO À COBRANÇA DE DÍVIDA ATIVA

Monografia de especialização apresentada na disciplina de Metodologia da Pesquisa, do Curso de Especialização em Tecnologia Java, do Departamento Acadêmico de Informática, da Universidade Tecnológica Federal do Paraná, Campus Pato Branco, como requisito parcial para obtenção do título de Especialista.

Orientadora: Profa. Andreia Scariot Beulke

PATO BRANCO
2017

MINISTÉRIO DA EDUCAÇÃO
Universidade Tecnológica Federal do Paraná
Câmpus Pato Branco
Departamento Acadêmico de Informática
Curso de Especialização em Tecnologia Java

UTFPR

TERMO DE APROVAÇÃO

SISTEMA DE APOIO A COBRANÇA DE DÍVIDA ATIVA

por

ADAIR PERDOMO FALCÃO

Este trabalho de conclusão de curso foi apresentado em 17 de novembro de 2017, como requisito parcial para a obtenção do título de Especialista em Tecnologia Java. Após a apresentação o candidato foi arguido pela banca examinadora composta pelos professores Beatriz Terezinha Borsoi e João Guilherme Brasil Pichetti. Em seguida foi realizada a deliberação pela banca examinadora que considerou o trabalho aprovado.

_____
Andreia Scariot Beulke
Profa. Orientadora (UTFPR)

_____
Beatriz Terezinha Borsoi
Banca (UTFPR)

_____
João Guilherme Brasil Pichetti
Banca (UTFPR)

_____
Robison Cris Brito
Coordenador da IV Especialização em Tecnologia Java

A Folha de Aprovação assinada encontra-se na Coordenação do Curso.

RESUMO

FALCÃO, Adair Perdomo. Sistema de Apoio a Cobrança de Dívida Ativa. 2017. 49 f. Monografia (Trabalho de especialização) – Departamento Acadêmico de Informática, Universidade Tecnológica Federal do Paraná, Campus Pato Branco. Pato Branco, 2017.

A cobrança de impostos/tributos é a forma utilizada para arrecadação de verbas para o sustento e manutenção das entidades públicas. Em períodos de instabilidade econômica, o resgate da dívida ativa, aquelas advindas de crédito tributário já vencido e que ainda não foi pago, se mostra como uma grande fonte de arrecadação. As principais dificuldades na cobrança deste tipo de dívida está a prescrição, a decadência e a inexigibilidade. Esses complicadores tornam necessário um maior controle do cadastro de dívidas para prevenir perdas de arrecadação por prescrição e por cobrança indevida e que podem ser acusadas como renuncio de receita pelo Tribunal de Contas. Este trabalho apresenta o desenvolvimento de um sistema *web* que tem o objetivo de apoiar a cobrança da dívida ativa utilizando o gerenciamento de dívidas e notificações para facilitar a organização e controle da dívida de órgãos públicos, proporcionando a otimização da cobrança da dívida e a redução da perda de receitas por meio da prescrição. Para o desenvolvimento desta ferramenta foi utilizado o *framework* VRaptor IV juntamente com o Hibernate. Foram utilizadas, ainda, as bibliotecas Bootstrap 3 e JQuery que facilitaram o desenvolvimento de interfaces ricas, intuitivas e dinâmicas. Com o desenvolvimento deste trabalho, espera-se instigar mais pesquisas que procurem aliar tecnologia e gestão pública e que venham suprir a necessidade de modernização do serviço público e, assim, facilitar os desafios do cotidiano dos servidores a fim de proporcionar uma melhor prestação de serviço para a população.

Palavras-chave: Dívida Ativa. Execução Fiscal. Protesto. Gestão Pública.

ABSTRACT

FALCÃO, Adair Perdomo. Support system for tax enforcement. 2017. 49 f. Monografia (Trabalho de especialização) – Departamento Acadêmico de Informática, Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco. Pato Branco, 2017.

The collection of taxes is the form used to collect funds for the maintenance and maintenance of public entities. In periods of economic instability, the redemption of active debt, those arising from tax credit that has already matured and has not yet been paid, is shown as a major source of revenue. The main difficulties in collecting this type of debt are prescription, decay and unenforceability. These complications make it necessary to have a greater control of the debt register to prevent losses of collection due to prescription and improper collection and that can be accused as a waiver of revenue by the Court of Auditors. This work presents the development of a web system that aims to support the collection of active debt using debt management and notifications to facilitate the organization and control of debt of public agencies, providing optimization of debt collection and reduction of debt. loss of income through prescription. For the development of this tool the VRaptor IV framework was used together with Hibernate. We also used the Bootstrap 3 and JQuery libraries that facilitated the development of rich, intuitive and dynamic interfaces. With this work we hope to instigate more research that seeks to combine technology and public management that will meet the great need of modernization of the public service and, thus, facilitate the daily challenges of the servers in order to provide a better service for the population .

Keywords: Debt. Tax Execution. Collection of tax. Public administration.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTAGENS DE CÓDIGOS

LISTA DE SIGLAS

AJAX	*Asynchronous JavaScript and XML*
API	*Application Programming Interface*
CCB	Código Civil Brasileiro
CDA	Certidão de Dívida Ativa
CDI	*Contexts and Dependency Injenction*
CNPJ	Cadastro Nacional de Pessoa Jurídica
CPD	Central de Processamento de Dados
CPF	Cadastro de Pessoa Física
CSS	*Cascading Style Sheet*
CTN	Código Tributário Nacional
HQL	*Hibernate Query Language*
HTML	*HypertextMarkup Language*
HTTP	*HyperText Transfer Protocol*
IDE	Integrated Development Environment
JSON	*JavaScript Object Notation*
JSP	*Java Servlet Page*
JSTL	*JavaServer Pages Standard Tag Library*
LOA	Lei Orçamentária Anual
MVC	*Model, View, Controller*
PDF	*Portable Document Format*
POM	*Project Object Model*
SGBD	Sistema Gerenciados de Banco de Dados
SLF4J	*Simple Logging Facade for Java*
SLQ	*Structured Query Language*
TCE	Tribunal de Contas do Estado
TCM	Tribunal de Contas do Munícipio
TCU	Tribunal de Contas da União
URL	*Uniform Resource Locator*
XML	*Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

O principal meio de arrecadação e sustento da União, do Estado e dos Municípios é a cobrança de impostos/tributos. Eles são utilizados na manutenção das entidades, custeando folha de pagamento, gastos de expediente, na aplicação e fiscalização das leis e na manutenção das políticas públicas.

Em períodos de instabilidade econômica surge uma tendência de aumentar a inadimplência dos contribuintes, gerando dificuldades em manter as políticas públicas e a necessidade de melhorar a eficiência na gerência dos gastos e ganhos, controlando custos, investimentos e arrecadação. Nestas circunstâncias a arrecadação da dívida ativa se mostra como uma das principais fontes de sustendo das entidades públicas.

A dívida ativa é todo crédito tributário e não tributário que não é pago dentro do prazo previsto pelas normas legais, com isso, é inscrito em livro de dívida e tornado público. A inscrição em dívida ativa torna possível a emissão da Certidão de Dívida Ativa (CDA), que é um título formal que certifica, garante e representa a existência de uma dívida líquida e certa perante a Fazenda Pública (SANTANA, 2010). A instituição da dívida ativa é regulamentada pelo artigo 201º da lei nº 5.172 de 25 de outubro de 1966, Código Tributário Nacional (CTN):

> "Art. 201. Constitui dívida ativa tributária a proveniente de crédito dessa natureza, regularmente inscrita na repartição administrativa competente, depois de esgotado o prazo fixado, para pagamento, pela lei ou por decisão final proferida em processo regular." (BRASIL, 1966).

Entre as principais dificuldades na cobrança da dívida ativa está a prescrição, que é uma das formas de extinção da exigibilidade da dívida, ou seja, a extinção do direto de cobrança e ocorre a partir das possibilidades previstas pelo CTN e pelo Código Civil Brasileiro (CCB), especialmente o Artigo 206º, Parágrafo 5º, inciso I do CCB que define um prazo de cinco anos para a prescrição da "pretensão de cobrança de dívidas líquidas constantes de instrumento público ou particular".

Assim, faz-se necessário um maior controle da prescrição das dívidas para prevenir percas de arrecadação por prescrição e por cobrança indevida, que podem ser acusadas como renuncio de receita pelos Tribunais de Contas. Existem outros importantes controles de exigibilidade geridos pelo departamento Jurídico, como, depósitos judiciais, parcelamentos ou medidas cautelares que suspendem a exigibilidade da dívida.

A cobrança judicial é a medida mais comum utilizada pelo poder público para interromper o protesto visando também inibir a inadimplência e a sonegação, já que há a

possibilidade de o devedor ter os bens penhorados/leiloados para o pagamento da dívida, para tanto a dívida deve ser inscrita em dívida ativa e ter CDA emitida.

Nos últimos anos outra medida vem sendo tomada como forma de evitar a instância judicial, como a emissão de avisos de débitos e o protesto extrajudicial de CDA, mecanismos que visam inibir a inadimplência e a sonegação, juntamente com o protesto ocorre a inscrição do Cadastro de Pessoa Física (CPF) ou Cadastro Nacional de Pessoa Jurídica (CNPJ) do contribuinte em órgãos de proteção ao crédito.

O elevado valor arrecadado anualmente por meio de cobrança da dívida ativa – Segundo a Lei Orçamentária Anual (LOA) nº 18.660 de 22/12/2015 do estado do Paraná a previsão de receita da dívida ativa tributária para o exercício de 2016 é de R$ 46.476.000,00 – demonstra a importância da gestão adequada dos créditos inscritos em dívida, os Balanços Gerais apresentados pelo Estado do Paraná compravam como medidas podem aumentar consideravelmente a arrecadação da dívida ativa, o valor arrecadado em 2013 foi de R$ 33.724.414,34 e em 2014 R$ 121.313.179,91 (PÁRANA, 2017) tal diferença é justificada pelas medidas tomadas em 2014 pelo governo do estado para a arrecadação da dívida ativa, como refinanciamento de dívidas, execuções e protestos.

Entretanto, é um grande desafio, principalmente aos municípios, controlar de forma eficiente o estoque da dívida e agilizar a sua cobrança. Existem diversos aspectos que dificultam a gerência e controle da dívida ativa, como a troca de informações entre o Jurídico e a Fazenda que pode ser ineficiência prejudicando o controle da prescrição dos débitos, da exigibilidade e da cobrança judicial.

Nesse contexto, um sistema computacional com fim específico de gerenciamento da dívida, que forneça um canal acessível de troca de informações entre a Fazenda e o Jurídico pode aperfeiçoar a cobrança da dívida potencializando a arrecadação. Assim, o objetivo desse trabalho é desenvolver uma aplicação *web* para gerenciar a dívida ativa de entidades públicas especialmente, municípios.

O desenvolvimento desse sistema se justifica pela facilidade de acesso entre todos os envolvidos nas diversas formas de cobrança da dívida. Além disso, vale ressaltar que as novas tecnologias e técnicas de desenvolvimento para *web* permitem criar interfaces ricas e responsivas, facilitando a interação do usuário e o acesso do sistema por meio de dispositivos móveis, com isso dando um pequeno passo na direção da tão necessária modernização da gestão pública.

2 FERRAMENTAS, TECNOLOGIAS E PROCEDIMENTOS

Este capítulo apresenta as ferramentas e tecnologias utilizadas no desenvolvimento deste trabalho e os procedimentos técnicos necessários para utilizá-las.

2.1 FERRAMENTAS E TECNOLOGIAS

O sistema foi desenvolvido na forma de aplicação *web*. Para isso, foram utilizadas as ferramentas e tecnologias descritas no Quadro 1.

Ferramenta / Tecnologia	**Versão**	**Disponível em**	**Aplicação**
MySQL Server	5.7	https://dev.mysql.com/downloads/mysql/	Sistema de gerenciamento de banco de dados (SGBD), que utiliza a linguagem *Structured Query Language* (SQL).
MySQL Workbench	6.3	https://dev.mysql.com/downloads/mysql/	Modelagem do Banco de Dados do Sistema.
Java EE	8.0	http://www.oracle.com/technetwork/java/javaee/downloads/index.html	Linguagem para desenvolvimento da aplicações.
HTML	5.0	Linguagem de marcação interpretada pelos navegadores *web*.	Linguagem de marcação de textos utilizada para desenvolvimento de interfaces de aplicações.
CSS	3	Linguagem de folhas de estilos interpretadas pelos navegadores *web*.	Linguagem que serve para "descrever" a aparência/estilo de uma página *web* através de folhas de estilo em cascata.
Bootstrap	3.3.6	http://getbootstrap.com/	*Framework* de estilizações de páginas por meio de *Cascading Style Sheets* (CSS).
Hibernate	5.1.0	http://Hibernate .org/	Para mapeamento objeto relacional e persistência de dados.
JQuery	2.2.4	https://jquery.com/	Biblioteca JavaScript utilizada no desenvolvimento da interface.
NetBeans	8.1	https://netbeans.org/	*Integrated Development Environment* (IDE) para desenvolvimento da aplicação.
VRaptor IV	4.2.0	http://www.vraptor.org/	*Framework* para desenvolvimento ágil de sistemas *web* com a linguagem de programação Java.
Apache TomCat	8.5	http://tomcat.apache.org/	Container de *Servlets* que implementa as tecnologias Java, funciona como um servidor para aplicações em Java.

Maven	4.0	https://maven.apache.org/	Modelagem do projeto e gerenciamento de dependências.
FontAwesome	4.6.3	http://fontawesome.io/	Fonte de ícones vetorizados
DataTable *Plugin*	1.10.16	https://datatables.net	*Plugin* para JQuery de criação e manipulação de tabelas dinâmicas
Notify.js *Plugin*	1	https://notifyjs.com/	*Plugin* em JavaScript para customização de notificações.
JQuery File Upload *Plugin*	9.19.1	https://blueimp.github.io/jQuery-File-Upload/	*Plugin* para JQuery de *upload* assíncrono de arquivos
JQuery Mask *Plugin*	1.7.6	https://plugins.jquery.com/mask/	*Plugin* para JQuery para atribuição de de máscaras nos campos de um formulário.
Apache POI	3.14	https://poi.apache.org/	*Application Programming Interface* (API) em Java para manipular arquivos do pacote Microsoft Office.

Quadro 1 – Ferramentas e tecnologias utilizadas no desenvolvimento do sistema

Para facilitar a configuração geral do projeto, incluindo nome, proprietário, versão, método de empacotamento e dependências, foi utilizado o Maven 4.0, que é baseado no conceito de *Project Object Model* (POM). Em outras palavras, o POM é um arquivo *Extensible Markup Language* (XML) que descreve e gerencia a configuração e as dependências do projeto.

Foram utilizados, ainda, alguns *frameworks* sendo o principal deles o VRaptor IV na versão 4.2.0. É um *framework* brasileiro para desenvolvimento ágil com a linguagem de programação Java, que foi desenvolvido em 2004 na Universidade de São Paulo. Em 2006 foi lançada a primeira versão estável denominada de VRaptor 2. Em 2014 foi liberada a quarta versão do *framework*, totalmente baseado em *Contexts and Dependency Injenction* (CDI), possibilitando o uso dos recursos nativos do servidor de aplicação integrado com os componentes da aplicação.

A quarta versão é voltada para desenvolvimento de aplicações *web* e APIs HTTP/REST para comunicação entre sistemas. VRpator é um *framework* que utiliza o padrão *Model, View, Controller* (MVC) e possibilita o uso de boas práticas da orientação a objetos e permite a escolha livre para a camada de visualização (CAVALCANTI, 2014).

Também foi utilizado o Hibernate 4.3.1 para o mapeamento objeto relacional em Java. Esse *framework* é utilizado para fazer o relacionamento com o banco de dados, facilitando o mapeamento das classes de modelo do projeto com as tabelas no banco de dados. Para isso, ele faz uso de arquivos XML e anotações Java, também facilita o desenvolvimento de

consultas e atualizações dos dados utilizando uma linguagem própria *Hibernate Query Language* (HQL).

Para facilitar o desenvolvimento do *front-end* foram utilizados os *frameworks open source* Bootstrap 3.3.7, para estilização de conteúdo e JQuery para dinamizar a interação com o usuário. Juntos, eles facilitam o desenvolvimento de interfaces responsivas e dinâmicas contando com uma grande biblioteca de classes e códigos em CSS e JavaScript.

Para o funcionamento da maioria dos recursos, principalmente dos *frameworks* foi necessário o uso de diversas bibliotecas secundárias que na maioria dos casos já foram instaladas automaticamente, o arquivo com as dependências necessárias para funcionamento do projeto pode ser visto no Apêndice A.

Todo o sistema foi desenvolvido utilizando um computador com especificações mínimas de 4GB de memória, processador Intel *core* i3 2.2Ghz, disco rígido com capacidade de 500GB e sistema operacional Windows 7.

2.2 PROCEDIMENTOS TÉCNICOS

Para desenvolver o trabalho com as ferramentas apresentadas na seção 2.1, deve-se instalar o NetBeans 8.1, distribuição Java EE incluindo o Apache TomCat 8.0.27. Para criar um projeto novo que faça uso dos recursos do VRaptor IV são necessárias algumas configurações adicionais:

1) Criar um novo projeto Maven de aplicação *web* e estruturar os diretórios conforme apresentado na Figura 1.

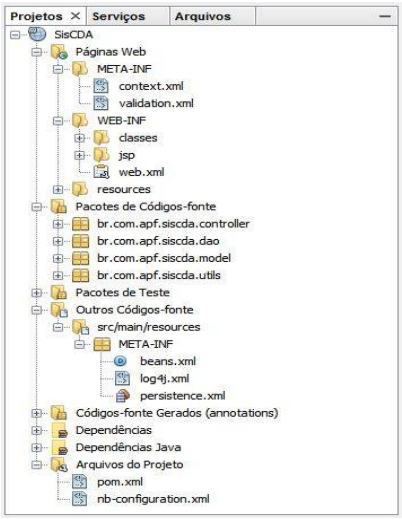

Figura 1 - Estruturação de diretórios do projeto

2) Adicionar ao arquivo pom.xml as dependências necessárias para o funcionamento correto dos recursos utilizados, tais dependências podem ser visualizadas no Apêndice A.

3) Ao usar um *Servlet Container,* como, por exemplo o TomCat, é necessário ativar os recursos do CDI. Para isso, é necessário adicionar um *listener* (Listagem 1) no *web*.xml. Caso o projeto não possua o *web*.xml basta adicionar um novo arquivo do tipo "Descritor de Implantação Padrão".

```
<listener>
        <listener-class> org.jboss.weld.environment.servlet.Listener </listener-class>
</listener>
```

Listagem 1 - Listener

4) Também é preciso indicar ao CDI para não validar os métodos automaticamente. Para isso, basta adicionar no diretório META-INF o arquivo *validation.xml* com o conteúdo da Listagem 2.

```
<?xml version="1.0" encoding="UTF-8"?>
<validation-config xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/configurationvalidation-configuration-1.1.xsd"
version="1.1">
      <executable-validation enabled="false"/>
</validation-config>
```

Listagem 2 - Desabilitando a validação automática de métodos

5) Para o devido funcionamento do CDI deve ser configurado um *beans* para o projeto. O arquivo "beans.xml" (Listagem 3) deve estar no diretório *WEB*-INF da aplicação.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
    version="1.1" bean-discovery-mode="all">
</beans>
```

Listagem 3 - Beans

6) Para imprimir os logs dos eventos internos o VRaptor utiliza o *Simple Logging Facade for Java* (SLF4J). Para isso, deve-se adicionar nas dependências do projeto o artefato do SLF4J e incluir o arquivo de configuração log4j.xml no diretório *src/main/resources* do projeto com o conteúdo, conforme Listagem 4.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
    <appender name="stdout" class="org.apache.log4j.ConsoleAppender">
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%d{HH:mm:ss,SSS} %5p [%-20c{1}] %m%n" />
        </layout>
    </appender>
    <category name="br.com.caelum.vraptor">
        <priority value="DEBUG" />
        <appender-ref ref="stdout" />
    </category>
</log4j:configuration>
```

Listagem 4 - Configuração do arquivo de logs

Após realizada essas configurações, o projeto está pronto para codificação do sistema utilizando como base o Vraptor4.

3 RESULTADOS

Este capítulo apresenta o resultado da realização deste trabalho, do seu contexto e objetivos, planejamento e desenvolvimento e resultado final.

3.1 ESCOPO DO SISTEMA

O escopo e os requisitos do sistema foram elaborados juntamente com os envolvidos (funcionários responsáveis pela tributação e pelo jurídico) no processo de cobrança de dívida ativa da prefeitura municipal da cidade de Francisco Beltrão no Paraná.

O sistema de apoio à cobrança de dívida ativa, que é o resultado desse trabalho, visa auxiliar nas atividades rotineiras de departamentos tributários e jurídicos, tornando-se um facilitador no momento de organizar, controlar e cobrar uma dívida ativa. Para tanto, é importante que o sistema possua formas acessíveis de gerenciar o cadastro de dívidas e a localização rápida de informações pertinentes ao usuário.

Para o gerenciamento e controle das dívidas, o sistema deve ser alimentado com um mínimo de informações (valor, data de vencimento, nome, estado e documento do contribuinte). Dentre as características de uma dívida, o estado da dívida é a mais importante já que que define a sua situação atual, ou ação ocorrida com a mesma. Este estado pode ter oito variações:

1. Administrativa: quando a dívida está vencida a menos de 05 anos a contar da data de vencimento e não foi executada nem protestada.
2. Protestada: quando a dívida está vencida a menos de 05 anos a contar da data de vencimento e possuir CDA emitida e protestada em cartório de protesto de títulos.
3. Aguardando Execução: quando a dívida está vencida a menos de 05 anos a contar da data de vencimento e possuir CDA emitida e enviada ao departamento Jurídico para iniciar cobrança judicial junto ao fórum.
4. Executada: quando a dívida está vencida a menos de 05 anos a contar da data de vencimento e possuir CDA emitida e protocolada em ação de cobrança judicial junto ao fórum.
5. Protestada e Executada: quando a dívida está vencida a menos de 05 anos a contar da data de vencimento e possuir CDA emitida e protestada em cartório de protesto de títulos e ainda protocolada em ação de cobrança judicial junto ao fórum.

6. Prescrita: quando a dívida não foi executada dentro de um prazo de 5 anos a contar da data de vencimento.

7. Quitada: quando a dívida foi quitada.

8. Inexigível: quando uma dívida se encontra em estado de inexigibilidade.

Devido ao prazo de prescrição das dívidas é necessário que o sistema notifique automaticamente o departamento jurídico e tributário de dívidas que necessitam ser executadas em diferentes períodos:

1. Contados 06 meses antes da data final de prescrição (5 anos a partir da data de lançamento) o sistema deve gerar ao menos uma vez por mês uma notificação informando sobre a data de prescrição da dívida.

2. Nos 15 dias anteriores à data de prescrição o sistema deve gerar uma notificação por dia informando sobre a data de prescrição da dívida.

A notificação de prescrição deve conter no mínimo a data de prescrição da dívida o nome e documento do contribuinte. Outras notificações geradas automaticamente pelo sistema devem ocorrer após algumas ações realizadas pelos operadores do sistema:

1. Notificar o departamento Jurídico quando uma dívida for encaminhada para execução fiscal, contendo: Nome e documento do contribuinte e data de vencimento.

2. Notificar o departamento Jurídico quando uma dívida foi quitada, contendo: Data de quitação da dívida, nome e documento do contribuinte.

3. Notificar ao departamento de tributação quando uma dívida foi executada no fórum, contendo: Nome e documento do contribuinte e data de vencimento.

4. Notificar ao departamento de tributação e jurídico quando uma dívida prescrever, contendo: Data de vencimento, data de prescrição, nome e documento do contribuinte.

O sistema deve manter um histórico das notificações recebidas e uma forma de consultá-lo. Além disso deve manter um histórico de cada dívida que conste a data de alteração dos seus dados e estados, identificação do operador que realizou a alteração e registro dos dados alterados.

Cada dívida deve possuir um espaço para anotações diversas. Cada anotação incluída, neste espaço, deve registrar a data da anotação e identificação do operador gerador da anotação. Para segurança dos dados o sistema deve gerar *backups* periódicos da base de dados e possibilitar que operadores administradores realizem alterações cadastrais diversas, gerem *backups* e recuperem dados a partir desses *backups.*

Algumas ações devem ser restritas ao papel de cada usuário, o operador de tributação tem o papel de incluir dívidas no sistema, alterar os dados primários da dívida, como, data de

vencimento, valor, documento do contribuinte e etc. cabe ainda ao operador de tributação a emissão e registro de CDA, o protesto da dívida e a sua baixa, além da quitação.

Aos operadores do departamento jurídico cabe a execução da dívida e baixa da execução fiscal. Uma dívida não pode ser quitada sem ter dado baixa na execução fiscal. Ambos os operadores podem consultar todas as informações presentes no sistema, assim como incluir anotações nas dívidas. Vários filtros de busca foram desenvolvidos para facilitar a consulta das dívidas principalmente por meio dos campos de datas e estado da dívida.

Os Tribunais de Contas do Estado, da União e do Munício (TCU, TCE e TCM) são tribunais administrativo que têm a competência de julgar as contas de administradores públicos e dos demais responsáveis por dinheiros, bens e valores públicos nos âmbitos federais, estaduais e municipais, assim como as pessoas que derem causa a perda, extravio ou outra irregularidade de que resulte prejuízo ao erário público, portanto devido à aspectos legais e fiscalizatórios o sistema deve manter a informações sobre as dívidas e suas alterações assim como todas as informações necessárias para possíveis auditorias, para isso, não é possível excluir usuário, dívida e anotações gravadas no sistema.

3.2 MODELAGEM DO SISTEMA

O processo de desenvolvimento do sistema foi realizado com base no modelo clássico da Engenharia de Software iterativo e incremental, no qual cada funcionalidade é dividida em ciclos chamados de incrementos e cada ciclo possui seis etapas que são: levantamento de requisitos, análise dos requisitos, projeto/modelagem, codificação, testes e implantação (SOMMERVILLE, 2003).

A primeira etapa consistiu em realizar o levantamento de requisitos, uma vez que o desenvolvedor ocupa também a posição de usuário do sistema e conhecedor do problema. O principal meio de obter os requisitos foi por compreensão de domínio, técnicas de entrevista e *brainstorming* com o intuito de levantar os requisitos juntamente com todos os *stackeholders* envolvidos no projeto, funcionários do setor de tributação e do departamento jurídico da prefeitura Municipal de Francisco Beltrão. Os requisitos levantados estão definidos em requisitos funcionais (Quadro 2), ou seja, todas as funções esperadas que o sistema deve fornecer e requisitos não funcionais (Quadro 3) que são as restrições sobre as funções e serviços prestados pelo sistema, geralmente ligadas a características como confiabilidade, tempo de resposta e alocação e espaço (SOMMERVILLE, 2003).

Identificação	Nome	Descrição
RF-01	Manter usuários	Permite o cadastro dos usuários do sistema, contendo os dados necessários para identificação, *login* e senha e a definição das restrições de acesso. Basicamente, os usuários do sistema são: administrador, tributação e jurídico.
RF-02	Gerar *backup*	O administrador possui uma interface no sistema para gerar *backups* de todos os dados a qualquer momento.
RF-03	Importação de dívidas em lote	O operador da tributação pode importar dívidas em lote por meio de uma planilha no formato XLS.
RF-04	Manter dívidas	O operador da tributação gerencia os dados dos cadastros das dívidas por meio das operações de salvar, editar, alterar, consultar e inativar uma dívida. Não será possível excluir a dívida por questões legais.
RF-05	Gerar relatórios	O sistema deve permitir que seja gerado relatórios de dívidas executadas, não executadas e protestadas.
RF-06	Incluir anotações sobre a dívida	O sistema permite que sejam inseridas anotações referentes a uma dívida. Não é permitido alterar e excluir as informações por questões legais.
RF-07	Notificar a prescrição de dívida aos usuários	O sistema deve notificar ao departamento de tributação e jurídico sobre os prazos de prescrição das dívidas.
RF-08	Notificar o envio de dívidas para execução	O sistema deve notificar os usuários do Jurídico sobre o envio de novas dívidas para execução.
RF-09	Notificar a quitação de dívidas	O sistema deve notificar os usuários do jurídico quando uma dívida foi marcada como quitada.
RF-10	Notificar a execução de dívidas	O sistema deve notificar ao departamento de tributação quando uma dívida foi marcada como executada.

Quadro 2 - Requisitos funcionais

No Quadro 2 estão elencados os requisitos não funcionais do sistema.

Identificação	Nome	Descrição
RNF – 01	Recuperar dados	Fornecer ao administrador uma interface para recuperação do banco de dados a partir dos *backups* realizados
RNF – 02	Quitar a dívida	O funcionário da tributação é o único que pode alterar o estado da dívida para quitado.
RNF – 03	Executar dívida	O funcionário do jurídico é o único que pode alterar o estado da dívida para executado ou protestado e executado.
RNF – 04	Enviar dívida para execução	O funcionário da tributação é o único que pode alterar o estado da dívida para enviado para

		execução.
RNF – 05	Gerar *backup* de dívidas automatizadas pelo sistema	O sistema deve gerar *backups* automáticos das dívidas antes de realizar uma importação de dados por lote.
RNF – 06	Cadastrar anotação sobre dívida	O sistema registra as alterações realizadas nas dívidas na forma de uma anotação, contendo o CPF do operador e o campo que foi alterado, mantendo assim um histórico de alterações nas anotações.
RNF – 07	Verifica integridade das dívidas	O sistema deve verificar a integridade dos dados das dívidas antes de inseridos ou editados, garantindo que possuam dados suficientes para o devido funcionamento do sistema, como, por exemplo o atributo vencimento que é necessário para o sistema gerar notificações automaticamente.

Quadro 3 - Requisitos não funcionais

Com base nos requisitos levantados foram elaborados dois diagramas de casos de uso. A Figura 2 mostra o diagrama de casos de uso do administrador, responsável por manter cadastro de usuários do sistema, gerar *backup* e recuperar os dados.

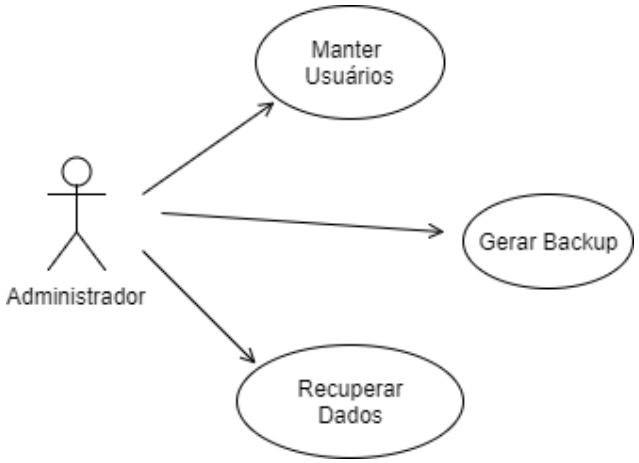

Figura 2 - Diagrama de Caso de Uso Administrador

A Figura 3 apresenta o diagrama de casos de uso para as funcionalidades dos funcionários da tributação e jurídico. Apresenta, ainda, o sistema como um dos atores envolvidos devido a importância das funcionalidades automatizadas por ele, como, por exemplo, a notificação de usuários.

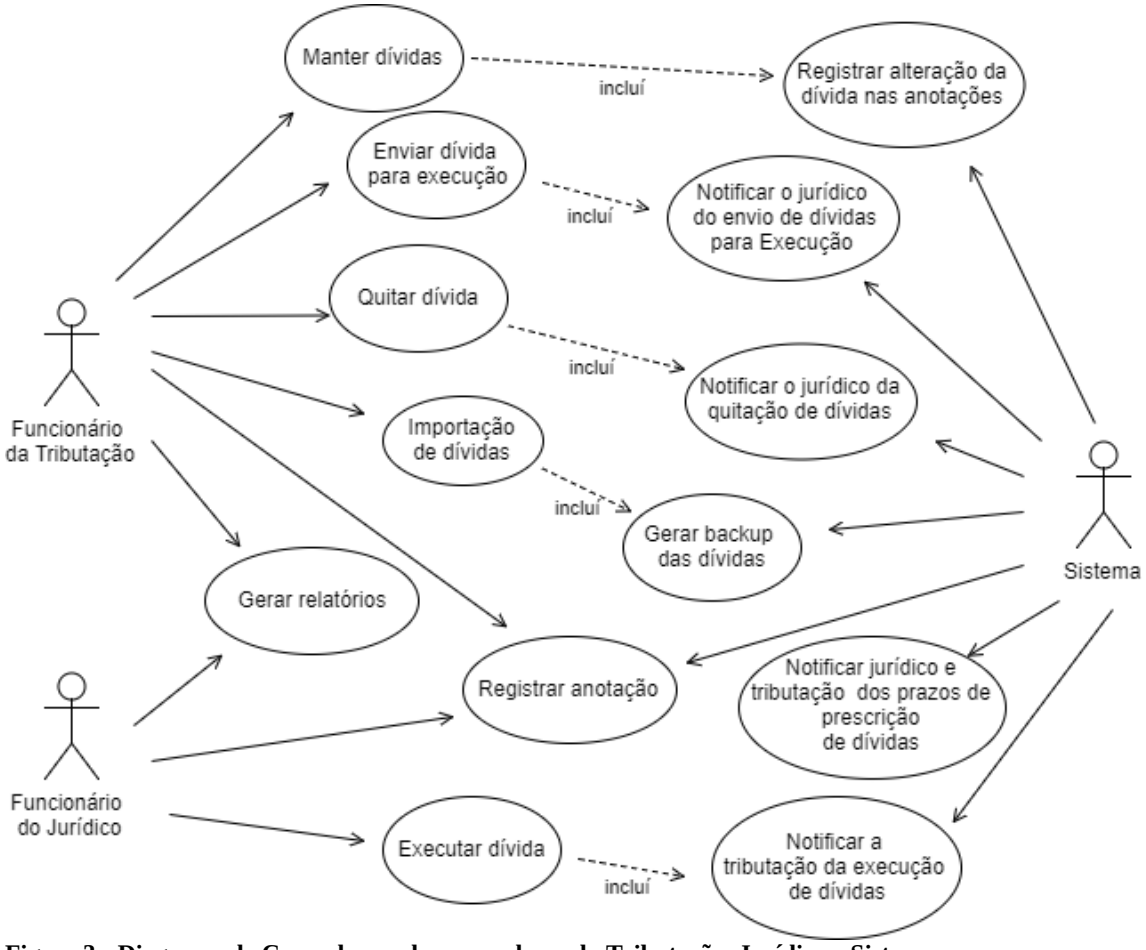

Figura 3 - Diagrama de Casos de uso dos operadores da Tributação, Jurídico e Sistema

Os dados obtidos com o levantamento de requisitos, acrescidos dos diagramas de casos de uso, foram necessários para elaborar o modelo conceitual e diagrama de entidade e relacionamento. A Figura 4 exibe o modelo conceitual de entidade e relacionamento do banco de dados que foi importante para o processo de criação do diagrama de entidade e relacionamentos, utilizado para a criação do banco de dados do sistema.

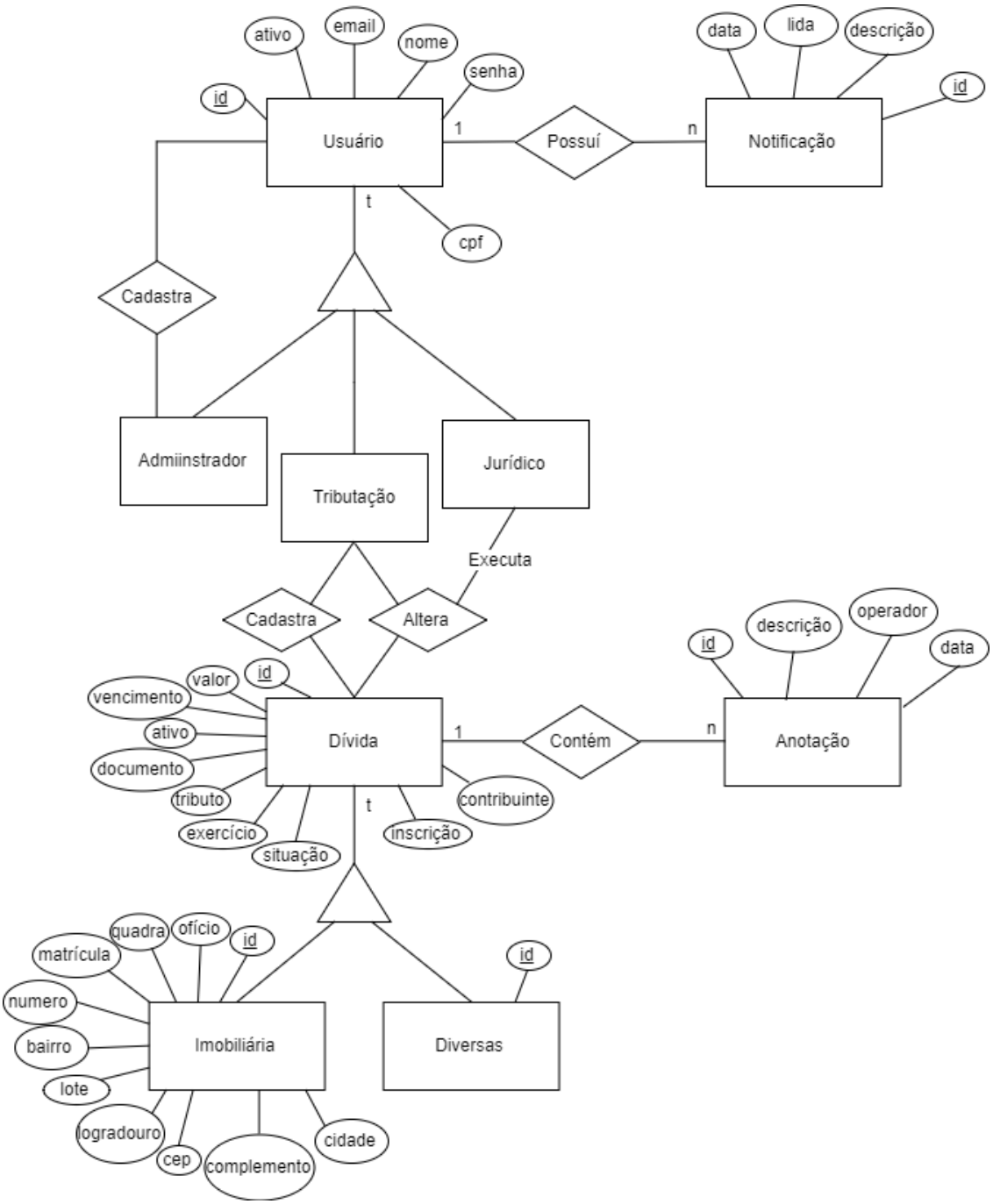

Figura 4 - Modelo Conceitual de Entidade Relacionamento

A Figura 5 apresenta o diagrama de entidade e relacionamento do sistema. A entidade denominada dívida é uma representação dos dados comuns de todas as dívidas. A especialização denominada dividaimoveis representa uma dívida que está diretamente associada a um imóvel e, por isso, possui alguns atributos exclusivos desse tipo de dívida, como, por exemplo, os atributos obrigatórios quadra e lote, além dos atributos não

obrigatórios, como aqueles relacionados ao endereço do imóvel. A especialização denominada dividadiveras é utilizada para diferenciar todas as outras dívidas, e são derivadas de outras fontes, como, empresas, podem ser taxas diversas de prestação de serviço ou até mesmo multas. Optou-se por criar essa especialização como facilitador de possíveis mudanças futuras que tragam a necessidade de adição de atributos específicos para essas dívidas.

Cada dívida pode conter várias anotações próprias e, com isso, a entidade anotação foi criada contendo atributos que armazenam a data de criação da anotação e a descrição da mesma. O campo divida_id é utilizado para associar a anotação com a dívida e o atributo chamado operador que associa a anotação com o CPF do operador que a gerou.

Os usuários que podem acessar o sistema estão representados pela entidade usuário. Estes usuários podem ser notificados de diversas ações e situação durante o funcionamento do sistema e, por isso, foi criado uma entidade denominada notificação para armazenar as informações de cada notificação que o usuário receber, a data que foi gerada, se foi lida e a sua descrição. Além disso, existe o atributo denominado tipo que é importante para o funcionamento do sistema, pois é por meio dele que o sistema identifica a função do usuário e restringe ou libera funcionalidades do sistema.

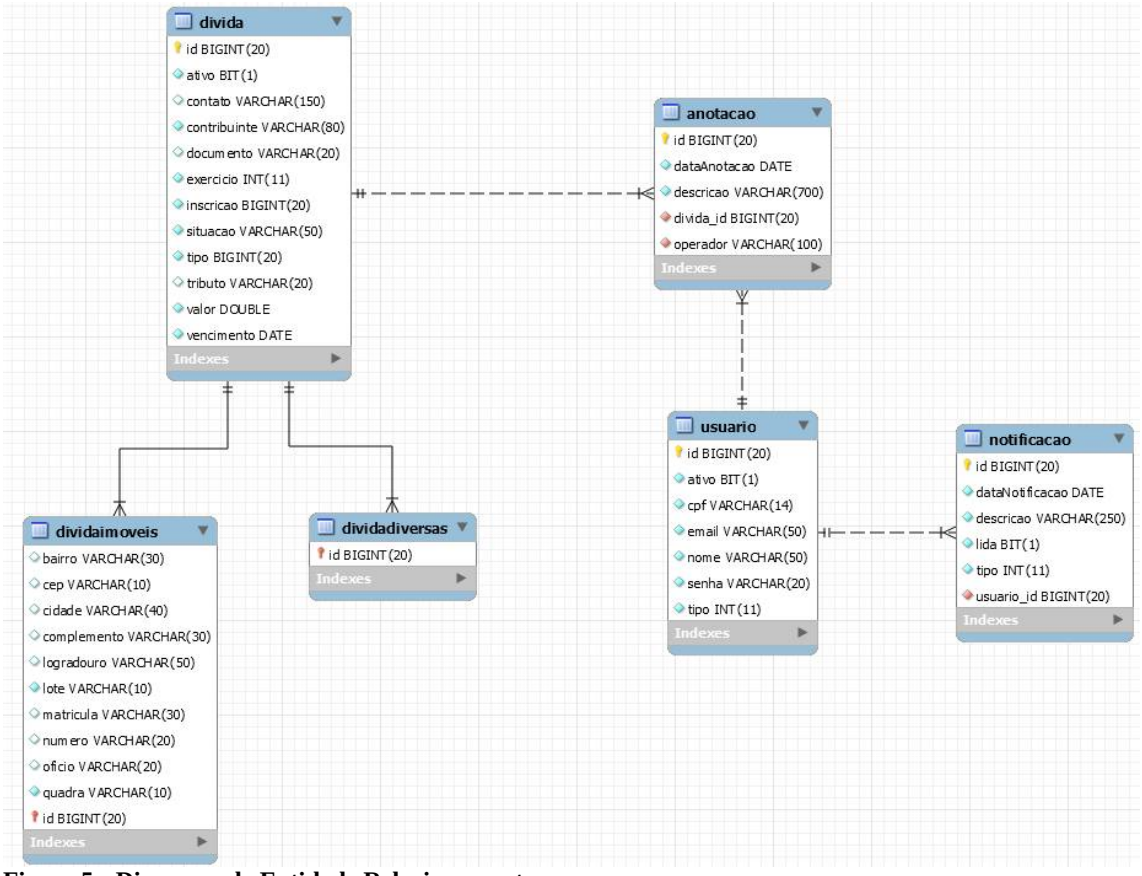

Figura 5 - Diagrama de Entidade Relacionamento

3.3 APRESENTAÇÃO DO SISTEMA

A Figura 6 apresenta a tela de *login* do sistema cujo acesso é realizado por meio dos campos CPF e senha.

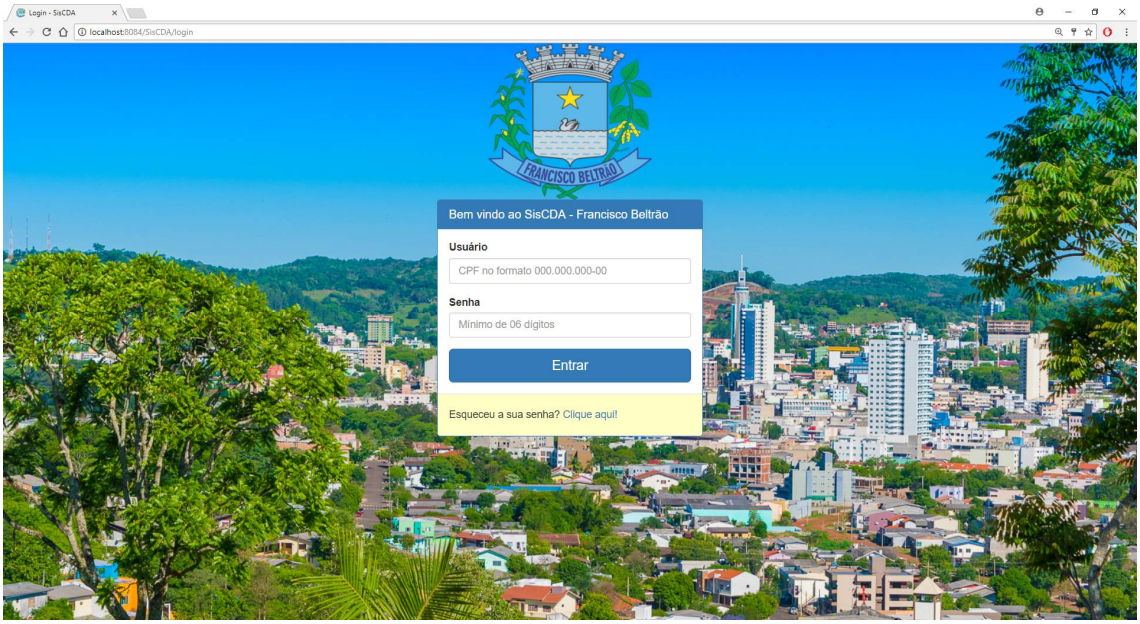

Figura 6 - Tela de login

A Figura 7 apresenta a tela inicial do sistema para o operador da tributação. Na lateral esquerda há uma barra de navegação na forma de menu lateral e outra barra na parte superior que apresenta informações sobre o sistema, usuário, notificações e alertas.

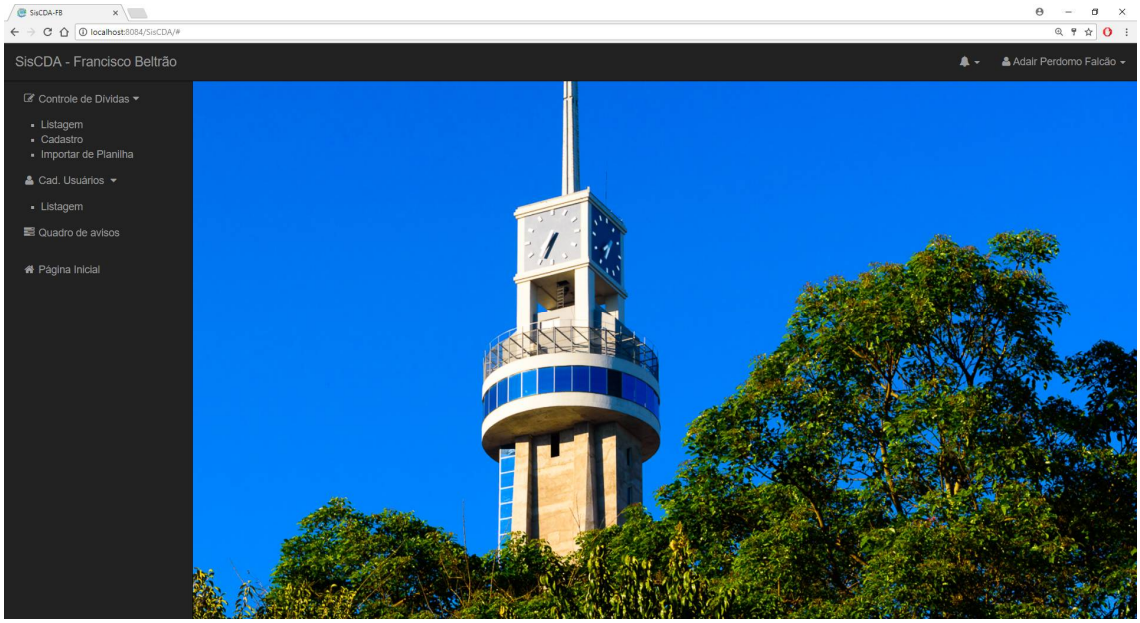

Figura 7 - Tela inicial do sistema

A transição da interface durante a navegação pelas telas é apresentada na Figura 8. Foi utilizado para navegar nas páginas um sistema de requisição assíncrona via *Asynchronous JavaScript and XML* (AJAX) em que é renderizada somente a parte da página de conteúdo principal sem a necessidade de recarregar novamente conteúdos desnecessários ou que já se encontram na página, como as barras de navegação.

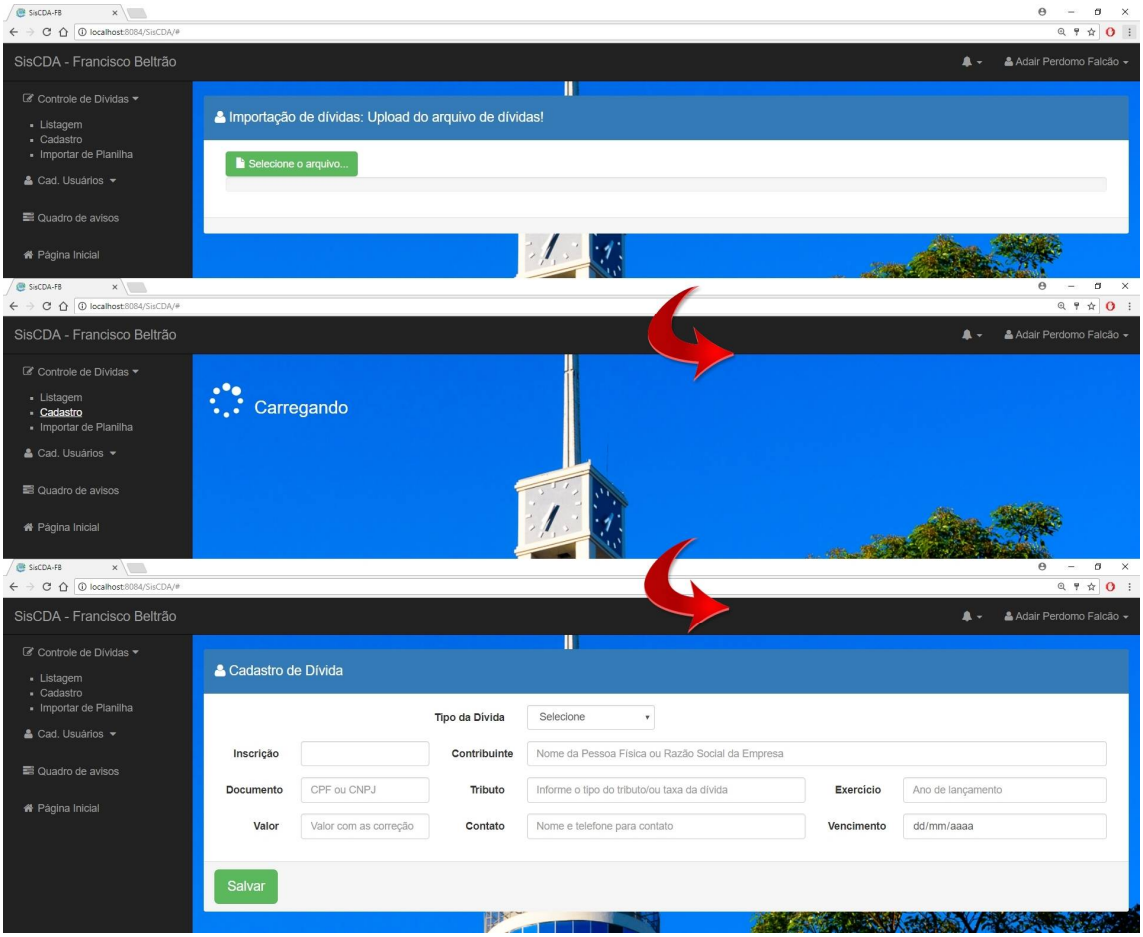

Figura 8 - Tela de transição entre as páginas

A Figura 9 exibe o cadastro de dívidas. Para a criação dessa tela foram utilizados recursos do Bootstrap, o alinhamento dos rótulos e dos campos foi elaborado com o uso do sistema de *grid* do Bootstrap com base nas classes para dispositivos de tamanho médio, "col-md-*". O principal uso do sistema é por meio de dispositivos *desktops* e a validação dos campos do formulário foi realizada por meio dos recursos do *Hypertext Markup Language* 5 (HTML5). O campo de Documento possui uma máscara para se organizar de acordo com o tamanho dos dados inseridos alterando-se entre o formato de CPF e CNPJ. O campo valor e CEP também possuem máscaras e foram implementas em JavaScript por meio do *framework* JQueryMask.

As dívidas cadastradas são apresentadas na tela de listagem de dívidas da Figura 10. Para a construção dessa tela utilizou-se o *plugin* DataTables a fim de organizar e apresentar os dados no formato de uma tabela dinâmica.

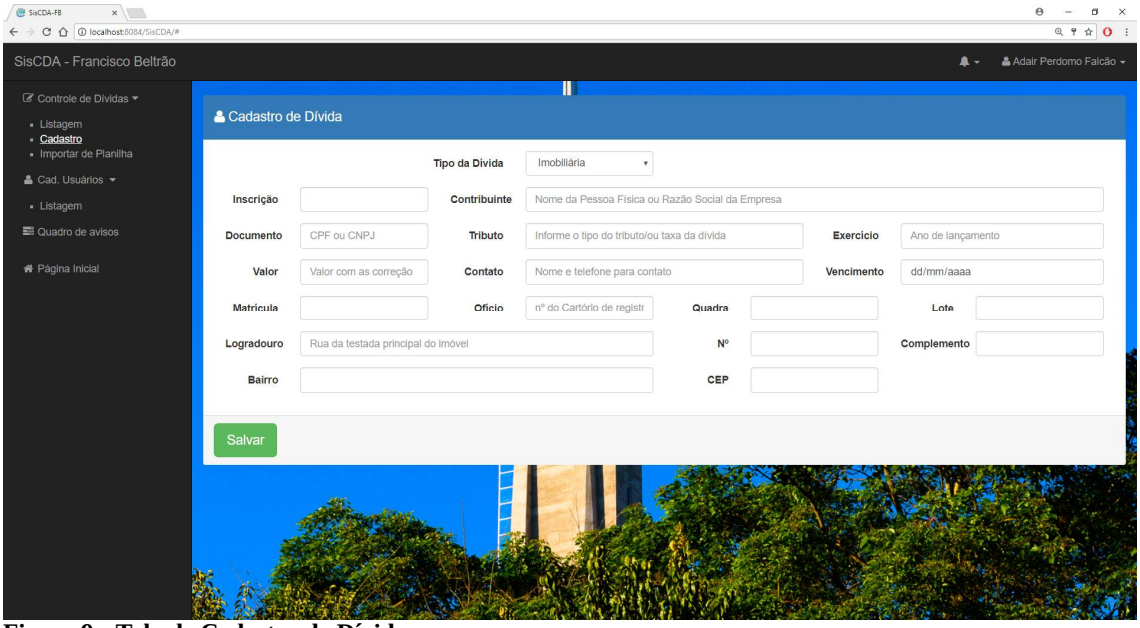

Figura 9 - Tela de Cadastro de Dívidas

O botão Carregar serve para requisitar ao servidor da aplicação as informações sobre as dívidas. Para diminuir a carga dessas informações o usuário pode filtrá-las para serem carregadas na tabela pelos filtros posicionados na parte superior. Todos os dados da tabela são requisitados por meio de uma chamada assíncrona utilizando JavaScript e AJAX e retornadas via objeto *JavaScript Object Notation* (JSON) que é organizado e distribuído na tabela.

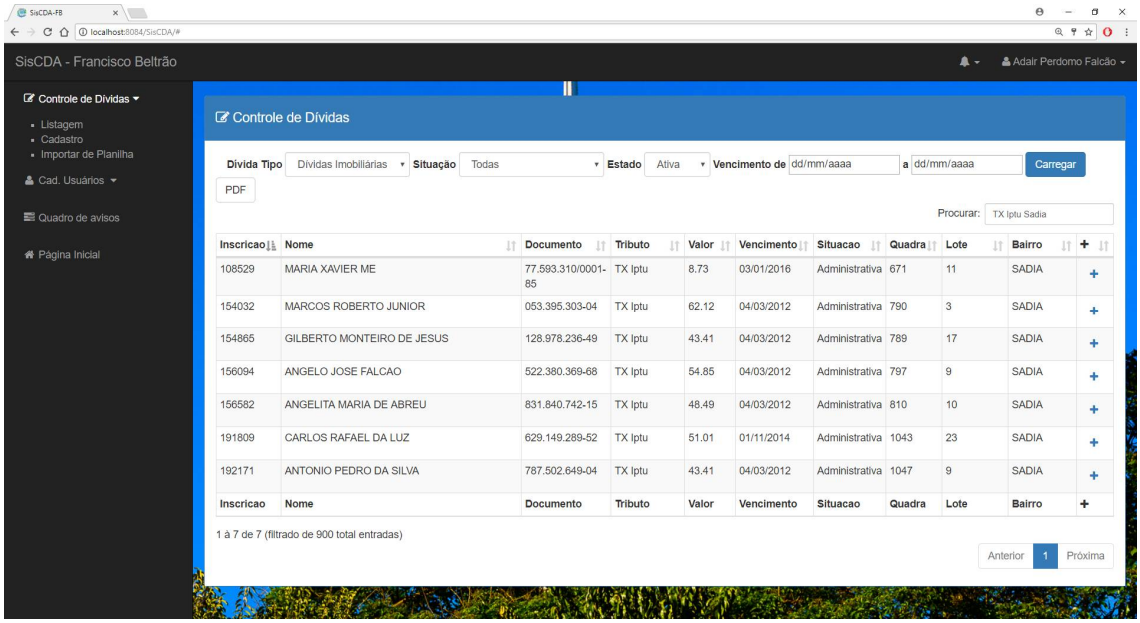

Figura 10 - Tela de listagem de dívidas

Na tela apresentada na Figura 10, o campo Procurar é utilizado para filtrar as linhas por qualquer informação que esteja na tabela. Também é possível organizar os dados em ordem crescente ou decrescente por qualquer uma das colunas. Com o botão *Portable Document Format* (PDF) é possível gerar arquivos do tipo PDF dos dados da tabela. Essa funcionalidade serve para gerar relatórios de dívidas. O botão na forma de um "+" é utilizado para acessar os dados completos da dívida incluindo a visualização e adição de anotações como mostra a Figura 11. É por meio dessa tela que o funcionário da tributação pode editar os dados de uma dívida.

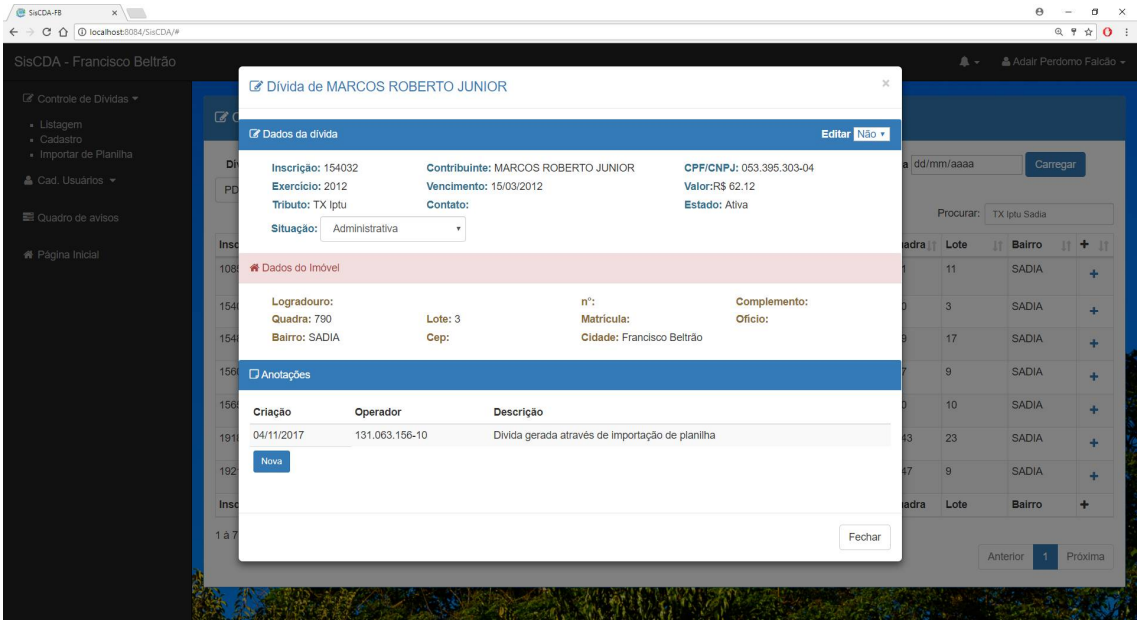

Figura 11 - Tela da descrição da dívida

A importação de dívida começa pelo *upload* da planilha, conforme apresentado na Figura 12. Após enviar o arquivo para importação, o sistema lê os cabeçalhos da planilha e gera uma tela de importação renderizada de forma dinâmica (Figura 13) de acordo com o tipo de dívida e das colunas da planilha, oferecendo ao usuário múltiplas opções para fazer a importação e, ao mesmo tempo, possibilitando a importação de dados a partir de planilhas com diferentes organizações de dados.

O início da importação está vinculado à seleção do mínimo de informação necessárias para funcionamento adequado do sistema, ou seja, é obrigatório que o usuário faça o vínculo dos seguintes atributos da dívida: inscrição, contribuinte, exercício, valor e vencimento, acrescidos de quadra e lote quando o tipo de dívida for imobiliário. Além disso, o sistema bloqueia o início da importação em caso de colunas vinculadas a mais de um atributo ao mesmo tempo.

Mesmo com essa validação, por se tratar de uma grande remessa de dados, antes de iniciar cada importação o sistema gera automaticamente um *backup* do banco de dados e da própria planilha utilizada para importação.

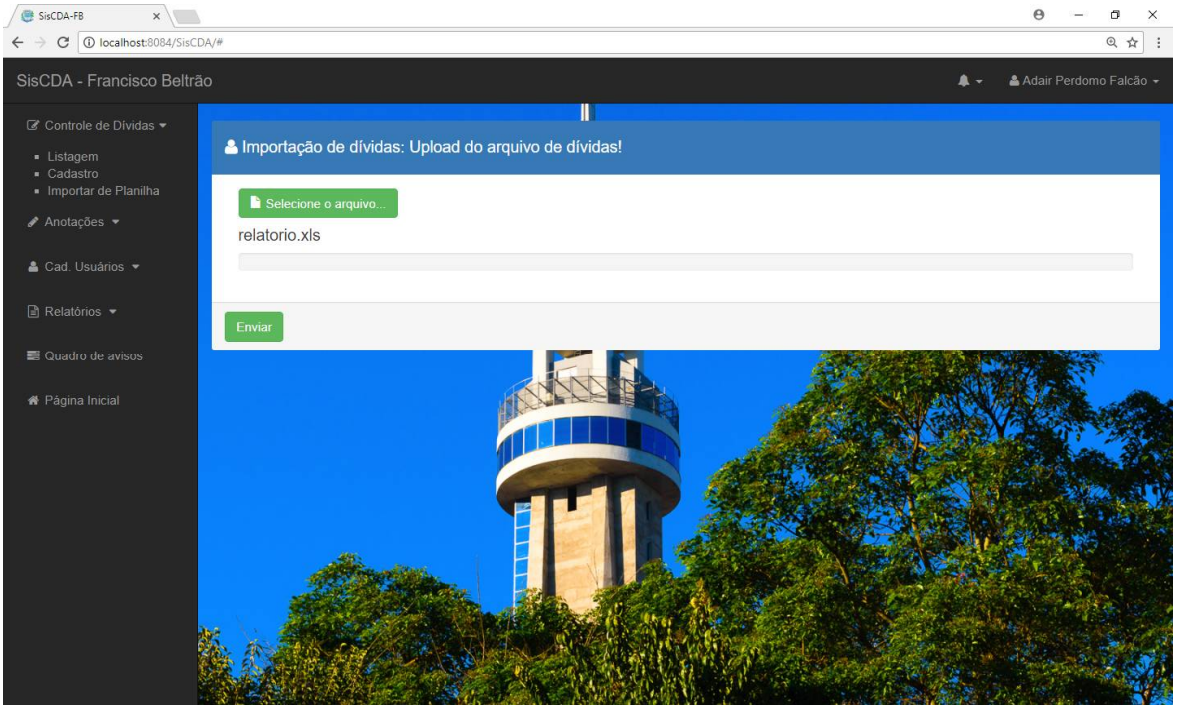

Figura 12 - Tela de *upload* do arquivo de importação de dívidas

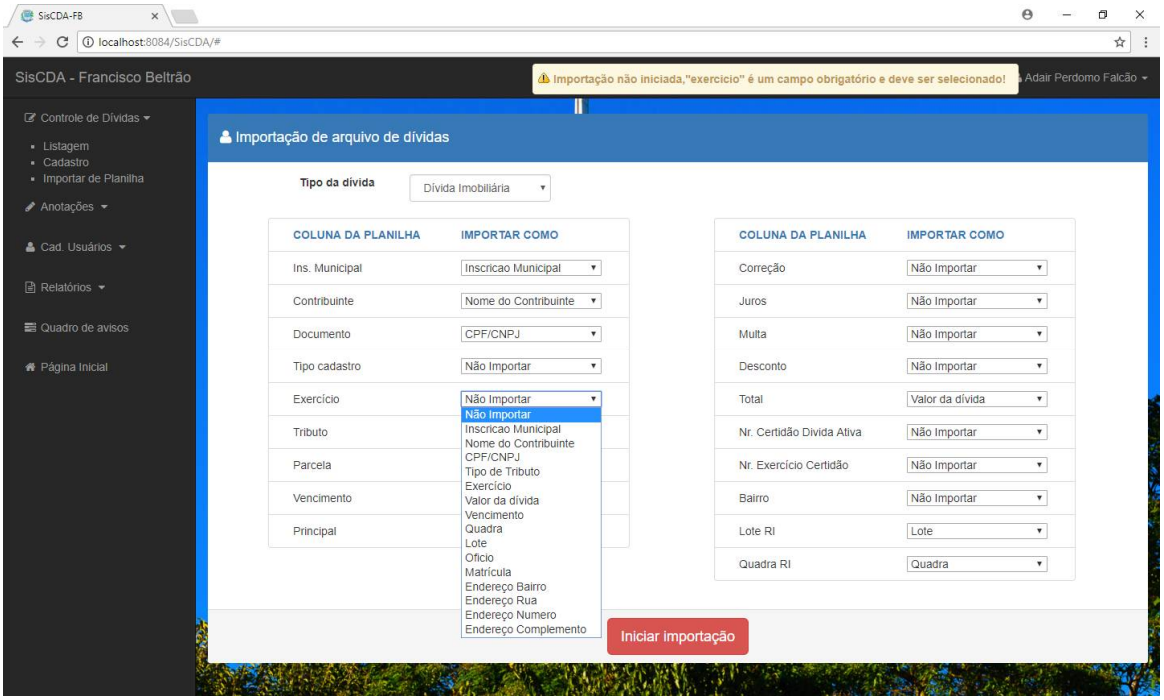

Figura 13 - Seleção de itens da importação de dívidas

A Figura 14 apresenta o sistema de notificação. No canto superior direito está o uso de mensagens informativas e de acesso rápido. Na janela principal está o quadro de avisos onde é apresentada a listagem do histórico de notificações por ordem cronológica da mais recente para a mais antiga. Algumas notificações são geradas automaticamente pelo sistema ou quando um usuário exerce uma ação específica no sistema e que seja de interesse para outros usuários que são notificados em tempo real por meio de uma mensagem enviada por um servidor *WebSocket* desde que estejam conectados.

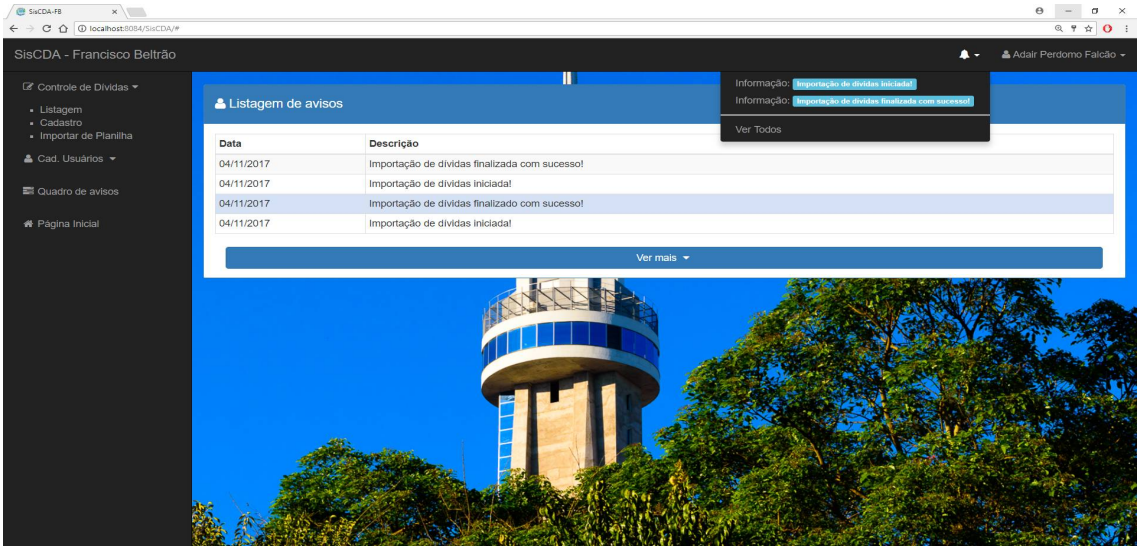

Figura 14 - Notificação e quadro de avisos

As Figuras 15 e 16 apresentam, respectivamente, as telas de cadastro e listagem de usuários que possuem funcionalidades específicas do administrador. O usuário poderá ativar e inativar usuários cadastrados e cadastrar novos usuários a partir do formulário de cadastro.

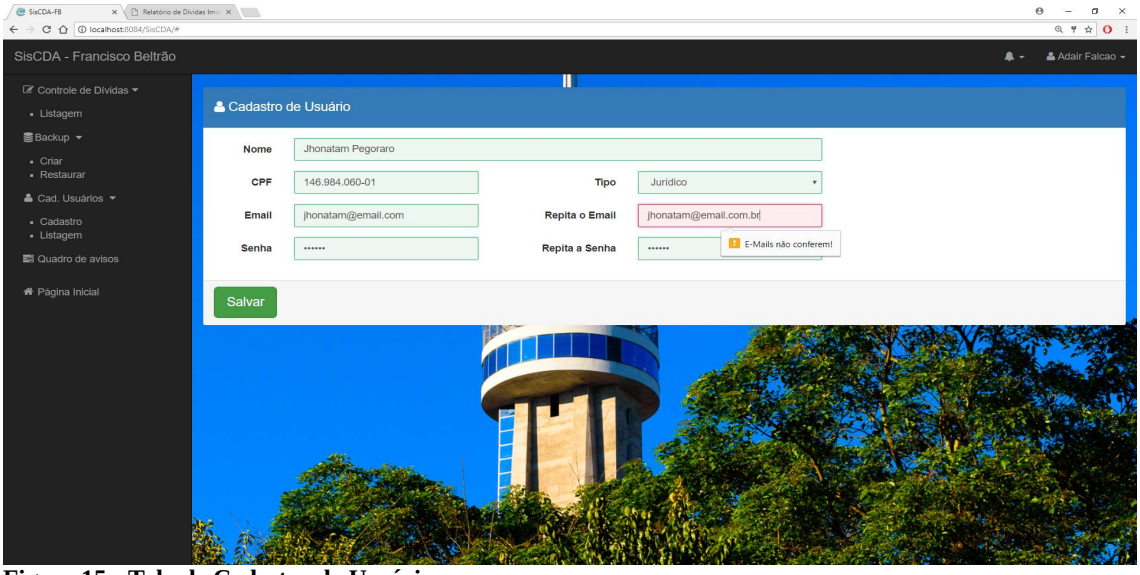

Figura 15 - Tela de Cadastro de Usuário

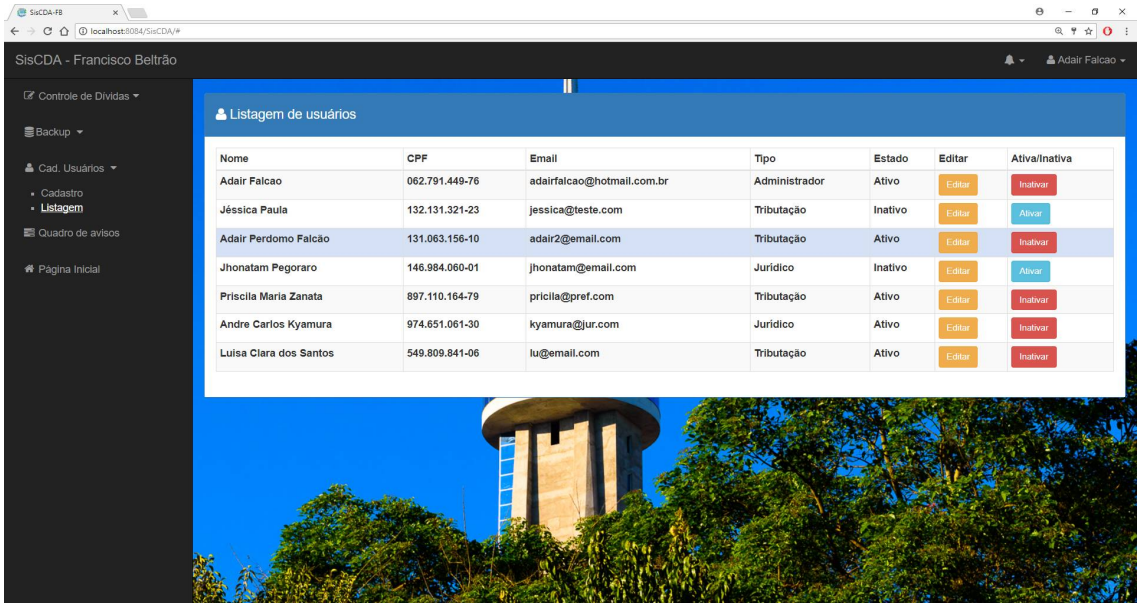

Figura 16 - Tela de Listagem de Usuários

O administrador, também poderá gerar *backups* e recuperar dados. A figura 17 mostra a tela utilizada para recuperar os dados.

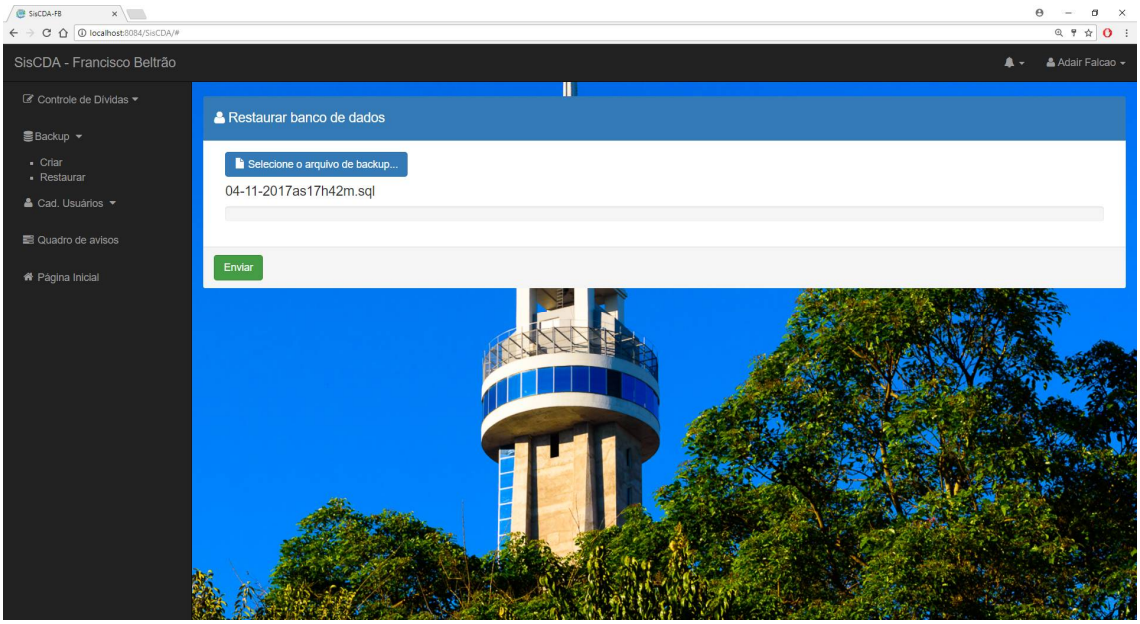

Figura 17 - Tela de recuperação de dados

3.4 IMPLEMENTAÇÃO DO SISTEMA

Para a codificação do sistema, foi realizada a instalação e configuração do ambiente de desenvolvimento juntamente com os recursos utilizados (*frameworks*, *plugins*, etc.). Durante a instalação do Netbeans, foi realizada a instalação e configuração do servidor ApacheTomcat. Para iniciar o desenvolvimento do sistema foi criado um projeto Maven que foi configurado conforme os procedimentos técnicos apresentados no item 2.2.

Como servidor de banco de dados foi instalado e configurado o MySQL e para criar e manipular o banco de dados (utilizando a linguagem SQL), para ser acessado e manipulado pelo Hibernate, foi utilizado o MySQL Workbench. Para a configuração do *framework* Hibernate, primeiramente foi necessário adicionar as dependências necessárias ao POM, conforme apresentado no Apêndice B. Para baixar as bibliotecas necessárias foi adicionado uma nova unidade de persistência por meio da criação de um arquivo XML com o nome de persistence.xml e configurado o Hibernate como provedor de persistência. Os dados deste arquivo encontram-se no Apêndice A.

Após a configuração do Hibernate foram criadas as classes de modelo conforme o diagrama de entidade e relacionamento (apresentado na Figura 5), juntamente com as devidas anotações necessárias utilizadas pelo Hibernate para realizar o mapeamento das classes, como demonstrado no Listagem 5.

```
@Entity
@Table(name = "notificacao")
public class Notificacao implements Serializable {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "id")
   private Long id;

   @Column(name = "descricao", length = 150, nullable = false)
   private String descricao;

   @Column(name = "dataNotificacao", nullable = false)
   @Convert(converter=LocalDateConverter.class)
   private LocalDate dataNotificacao;

  @Column(name = "lida", nullable = false)
   private Boolean lida;
 //getters e setters
   ...
}
```

Listagem 5 - Classe do modelo Notificação

Para que o Hibernate saiba quais classes são entidades, além das anotações, deve ser realizado o mapeamento das classes do modelo no arquivo de configuração da unidade de persistência. Um detalhe importante é que se for passar uma instância de entidade como um objeto individual a classe da entidade deve implementar a interface *Serializable.* Com essas configurações o Hibernate pode criar no todas as tabelas e seus relacionamentos, desde que esteja habilitado.

Para realizar a persistência de dados no banco, foi necessário criar uma classe de controle para configurar as conexões e sessões com o banco de dados (Listagem 6). Essa classe fica responsável por criar uma fábrica de sessões que abrem novas conexões com o banco de dados de acordo com a configuração da unidade de persistência.

```
@ApplicationScoped
public class DatabaseConnection {
   private static DatabaseConnection dbConnection;
   private EntityManagerFactory emf;

   private DatabaseConnection(){
      emf = Persistence.createEntityManagerFactory("SisCDA_PU");
   }

   public static DatabaseConnection getInstace(){
      if (dbConnection == null){
         dbConnection = new DatabaseConnection();
      }
      return dbConnection;
   }

   public EntityManagerFactory getEntityManagerFactory(){
      return emf;
   }

   public EntityManager getEntityManager(){
      return emf.createEntityManager();
   }
   @PreDestroy
   public void closeEntityManager(){
      if(this.emf.isOpen() ){
         this.emf.close();
      }
   }
}
```

Listagem 6 - Classe de controle de conexões com o banco de dados

Na Listagem 6, a anotação @ApplicationScoped serve para indicar que é uma classe de “aplicação”, ou seja, que deve ser instanciada uma única vez durante a execução da aplicação.

O uso de tipos genéricos em Java possibilita a reutilização de códigos, evitando redundâncias como, por exemplo, os vários métodos similares de consulta e manipulação do banco de dados. Para evitar essa redundância foi criada uma classe genérica de acesso ao banco de dados contendo as principais operações, gravar, editar, buscar por *id*, buscar todos e excluir, como mostrado na Listagem 7.

```
public abstract class GenericDao<T, ID extends Serializable> {
   …
   private Class<T> persistedClass;
   protected GenericDao(Class<T> persistedClass) {
      this.persistedClass = persistedClass;
   }
   public void insert(T object) throws Exception {
      em = DatabaseConnection.getInstace().getEntityManager();
      em.getTransaction().begin();
      try {
         em.persist(object);
         em.getTransaction().commit();
      } catch (Hibernate Exception ex) {
         throw new Exception("Erro ao inserir! " + ex.getMessage());
      } finally {
         if (em != null && em.getTransaction().isActive()) {
            em.getTransaction().rollback();
         }
      }
   }
   public List<T> findAll() {
      em = DatabaseConnection.getInstace().getEntityManager();
      return em.createQuery("from " + persistedClass.getName()).getResultList();
   }
   public void update(T object) throws Exception {    …    }
   public void remove(ID id) throws Exception {    …    }
   public T findById(ID id) {    …    }
}
```

Listagem 7 - Classe genérica de manipulação do banco de dados

A Listagem 8 contém o código da classe *UsuarioDao* que faz uso da classe genérica e implementa a classe genérica *GenericDao*. Além disso, contém o método *getUsuario*, necessário para uma consulta específica ao banco de dados. Neste caso, a consulta de verificação do *login* e senha do usuário.

```
public class UsuarioDao extends GenericDao<Usuario, Long> {

   public UsuarioDao() {
      super(Usuario.class);
   }

   public Usuario getUsuario(String cpf, String senha) throws Exception {
      Usuario usuario = null;
      em = DatabaseConnection.getInstace().getEntityManager();
      Query query = em.createQuery("from Usuario where cpf = :cpf and senha= :senha");
      query.setParameter("cpf", cpf);
      query.setParameter("senha", senha);
      try {
         usuario = (Usuario)query.getSingleResult();
      } catch (Hibernate Exception ex) {
         throw new Exception("Erro no login! " + ex.getMessage());
      }
      return usuario;
   }
}
```

Listagem 8 - Classe de acesso ao banco de dados do modelo Usuário

O *framework* Vraptor IV torna desnecessário a implementação de *servlets* para realizar a comunicação entre a camada de visualização e a camada de controle, bastando, para isso, anotar as classes de controle com *@Controller*. Dessa forma, os métodos públicos dessa classe ficam disponíveis para receber requisições *web*. Também possibilita o uso de injeção de dependências nas classes por meio da anotação *@Inject* no atributo ou no construtor. Neste caso é necessário adicionar, também, o construtor padrão e anotá-lo com *@Deprecated* para funcionamento correto do CDI, com isso o VRaptor utilizando o CDI se encarrega de criar ou localizar essas dependências não tendo diferença usar a injeção via construtor ou diretamente no atributo da classe.

Os métodos públicos dos controladores são acessíveis por meio da *Uniform Resource Locator* (URL) http://EndereçoDoServidor/Sistema/NomeDaClasse/NomeDoMétodo, os parâmetros recebidos no método são populados automaticamente pelo VRaptor de acordo com a requisição e o tipo de dado. Por convecção do VRaptor, as classes controladoras devem conter o sufixo *Controller* e o próprio *framework* se encarrega de excluir o sufixo para definir a URL de acesso.

Ao chamar um método o VRaptor redireciona a requisição para o *Java Server Pages* (JSP) de nome correspondente na forma /WEB-INF/jsp/NomeDaClasse/NomeDoMétodo.jsp. Essas convenções podem ser configuradas e modificadas por meio do uso de anotações, ou pelo objeto da classe *Result* que será apresentado mais abaixo.

A Listagem 9 apresenta uma parte da classe *LoginController*.java demostrando o uso de injeção de dependências pela anotação *@Inject* no construtor. Dessa forma, não é preciso instanciar e popular os atributos, outro uso dos padrões do *framework* pode ser visto nas anotações acima dos métodos públicos, que indicam o método de requisição conforme a semântica dos métodos do *HyperText Transfer Protocol* (HTTP) (G*et, Post, Put, Patch, Delete, Head, Options, Connect* e *Trace*), requisições enviadas de outro tipo que não o da anotação são rejeitadas automaticamente.

```
@Controller
public class LoginController {
   private UsuarioLogado logado;
   private Result result;
   @Inject
   public LoginController(UsuarioLogado logado, Result result) {
      this.logado = logado;
      this.result = result;
   }
   @Path("/login")
   public void login() {
      if (logado.isLogado()) {
         logado.deslogar();
      }
   }
  @Post("/login/logar")
   public void logar(String cpf, String senha) {
      UsuarioDao dao = new UsuarioDao();
      Usuario usuario = dao.getUsuario(cpf, senha);
      if (usuario == null) {
         result.include("msgRetorno", "Login ou Senha Invalidos!");
         result.redirectTo(LoginController.class).login();
      } else {
         logado.loga(usuario);
         result.redirectTo(IndexController.class).index();
      }
   }
}
```

Listagem 9 - Classe controladora do login

Na Listagem 9, o atributo “logado” age como um objeto “reservatório” que mantém os dados do usuário que está logado durante toda sessão. Isso é possível pois a classe

UsuarioLogado está anotada com *@SessionScoped*, indicando que seus objetos devem permanecer durante toda a sessão. O método *public void login*( ), serve para o *framework* redirecionar todas as requisições de *urls /login* para a *view* correspondente: *login.jsp*. O método logar(*String* cpf, *String* senha), recebe dois parâmetros enviados por método "*post*" a partir de uma requisição na URL /servidor/SisCDA/login/logar, esses parâmetros são utilizados para fazer uma busca no banco de dados, caso a busca resulte em nulo, a requisição é retornada para a página de login juntamente com uma mensagem informando o erro de login, caso a busca retorne uma pessoa, é efetuado o login dela armazenando seus dados na classe UsuarioLogado pelo método logado.login(usuario), e redirecionando a requisição para o método index() do controlador *IndexController* e consequentemente para a *view* inicial do sistema o *index.jsp*.

O atributo/objeto *result*, da classe *Result* é de grande importância para o desenvolvimento dos controladores. Ele tem como principais funções encaminhar a requisição para outras lógicas, redirecionar para outra *view*, retornar status de requisições e adicionar atributos na requisição ou objetos complexos para a *view* no formato JSON.

Um dos requisitos do sistema é o acesso do sistema apenas por usuários cadastrados, sejam administradores, funcionários da tributação ou do jurídico. Com isso, foi necessário desenvolver um sistema de autenticação de usuários juntamente com restrições de acesso. O VRaptor torna essa tarefa bastante simples por meio do uso da anotação *@Intercepts* que possibilita criar uma classe que intercepta requisições específicas executando a lógica apropriada para cada uma delas.

A Listagem 10 apresenta parte do código de uma classe que intercepta as requisições e verifica o estado do usuário, ou seja, se é um usuário ativo e também se o usuário já está logado. A verificação ocorre no método anotado com *@AroundCall*. A anotação *@Accepts* serve para indicar métodos ou classes que podem ser aceitas sem interceptar e direcionar à alguma lógica e, neste caso, apenas as requisições destinadas a classe *LoginController* são aceitas sem interceptar.

```
@Intercepts
public class LoginInterceptor {

   private HttpServletRequest request;
   private Result result;
   private UsuarioLogado logado;

   @Inject
```

```
public LoginInterceptor(HttpServletRequest request, Result result, UsuarioLogado logado)
{ … }
…
@AroundCall
public void intercept(SimpleInterceptorStack stack) {
   if (logado != null && logado.isLogado() && logado.getUsuario().getAtivo()) {
      stack.next();
   } else {
      result.redirectTo(LoginController.class).login();
   }
}
@Accepts
public boolean restritos(ControllerMethod method) {
   return !method.getController().getType().equals(LoginController.class);
}
```

Listagem 10 - Classe que intercepta requisições

O código da Listagem 10 mostra como uso do *framework* VRaptor agilizou o desenvolvimento do sistema, fornecendo uma forma de implementar os controladores de maneira ágil e prática por meio do uso de anotações facilitando especialmente a codificação de acesso aos métodos por requisições em URLs específicas. Além disso, o uso da classe *Result*, para redirecionar ou adicionar objetos na camada de visualização de acordo com a lógica dos controladores se mostrou de grande ajuda.

A codificação das interfaces de visualização do cliente foi realizada utilizando as tecnologias HTML5, CSS e JavaScript agregados a arquivos do tipo JSP. Para manipular informações em um arquivo JSP foi utilizado uma "biblioteca de *tags*" denominada *JavaServer Pages Standard Tag Library* (JSTL) que possibilita manipular arquivos JSP que são renderizados em HTML e enviados para visualização pelo cliente.

A Listagem 11 apresenta (nos trechos em azul) o uso do JSTL para renderizar uma lista dos usuários cadastrados ao mesmo tempo que restringe uma parte para visualização apenas para o usuário do tipo administrador. Neste caso, as duas últimas colunas só aparecem quando o usuário logado é do tipo administrador, uma vez que possibilitam formas de edição das informações dos usuários cadastrados, funcionalidade restrita ao administrador.

```
<tbody>
    <c:forEach items="${userList}" var="usuario">
        <tr>
            <td>${usuario.nome}</td>
            <td>${usuario.cpf}</td>
```

```
                <td>${usuario.email}</td>
                <td>
                    <c:choose >
                        <c:when test="${usuario.tipo == 1}">
                            Tributação
                        </c:when>
                        <c:when test="${usuario.tipo == 2}">
                            Jurídico
                        </c:when>
                        <c:when test="${usuario.tipo == 3}">
                            Administrador
                        </c:when>
                    </c:choose>
                </td>
                <td>…</td>
                <c:if test="${logado.tipo == 3}">
                    <td>
                        <button type="button" class="btn btn-sm btn-warning" data-
                        toggle="modal" data-target="#modalEditar" data-id="${usuario.id}">
                            Editar
                        </button>
                    </td>
                    <td>…</td>
                </c:if>
            </tr>
        </c:forEach>
    </tbody>
```

Listagem 11 - Parte do JSP da listagem de usuários

A navegação entre as telas do sistema ocorre por meio de requisições assíncronas com o uso de JavaScript e AJAX. Isso possibilita transições mais fluidas e sem necessidade de recarregar informações que se encontram na tela. Isso resulta em menos informações sendo transferidas pela rede e uma maior agilidade nas transições. A Listagem 12 mostra a função da requisição realizada para acessar a página de cadastro de dívidas que demonstra o uso do *framework* JQuery, como, por exemplo, o uso de $(*'#id'*) para manipular os componentes da interface.

```
function dividaCadastro() {
        carregando('main-content');
        $.ajax({
            url: '/SisCDA/divida/cadastro'
        }).done(function (retorno) {
```

```
            $('#main-content').empty();
            $('#main-content').hide().html(retorno).fadeIn();
        }).fail(function (data) {
            $('#main-content').html('Erro ao carregar a página!');
        });
};
```

Listagem 12 - Código JavaScript para navegação entre as páginas

Para facilitar o cadastro de dívidas o sistema traz uma forma de importar dívidas para o sistema a partir de uma planilha de extensão XLS. Para isso, foi utilizado a API Apache POI. Como o processamento da importação pode ser demorado, uma vez que depende diretamente da quantidade de registros da planilha, foi desenvolvido uma rotina de importação com o uso de *threads*. A Listagem 13 mostra o construtor da classe ImportacaoThread que recebe os atributos necessários para iniciar a importação e o método *run()* que é chamado ao iniciar o processamento.

```
public ImportacaoThread(HSSFWorkbook wb, List<String> selecionados, HttpServletRequest req,
String usuario) {
    this.wb = wb;
    this.selecionados = selecionados;
    this.req = req.getServletContext();
    this.usuario = usuario;
}
@Override
public void run() {
    if (this.wb != null) {
        try {
            notificacao.notificarImportacao(1);
            if (selecionados.get(0).equals("1")) {
                importarDividasImobiliarias();
            } else if (selecionados.get(0).equals("2")) {
                importarDividasDeReceitasDiversas();
            }
        } catch (IOException ex) {
            System.out.println("Erro na importacao: "+ex.getMessage());
            ex.printStackTrace();
        }
}
```

Listagem 13 - Parte do código da thread de importação de dívidas

O sistema de notificações é umas das principais funcionalidades do sistema, pois informa aos usuários sobre a situação de uma dívida sem a necessidade de uma conferência diária por parte dos funcionários. As principais notificações são realizadas quando um usuário altera a situação de uma dívida, algumas são menos relevantes, como a notificação de início e fim da importação de dados. As notificações que servem para informar sobre o prazo de prescrição são as mais importantes, pois servem para evitar prejuízos com a falta de execução. Isso evita processos de auditoria e de renúncia de receita por parte do ministério público e do tribunal de contas. A Listagem 14 mostra o recebimento e processamento de uma mensagem de notificação no lado do cliente.

```
ws.onmessage = function (event) {
    var qutNotificacoes = $('#qut_notificacao');
    var a = (parseInt(qutNotificacoes.text()) + 1);
    var listaNotificacoes = $('#lista_notificacoes');
    var mensagem = event.data.split(":");
    var tipo = mensagem[0];
    var descricao = mensagem[1];
    if (a > 4) {
       $('#lista_notificacoes').find('li:first').remove();
    }
    if (tipo == "1") {
                $("<li>  <a href=\"#\" class=\"notificacoes\" \">Sucesso: <span class=\"label label-
                 success\">" + descricao + "</span></a> </li>").insertBefore('#divisor');
    } else if (tipo == "2") {
                $("<li>  <a href=\"#\" class=\"notificacoes\" \">Informação: <span class=\"label label-
                info\">" + descricao + "</span></a> </li>").insertBefore('#divisor');
    } else if (tipo == "3") {
                $("<li>  <a href=\"#\" class=\"notificacoes\" \">Cuidado: <span class=\"label label-
                warning\">" + descricao + "</span></a> </li>").insertBefore('#divisor');
    } else if (tipo == "4") {
                $("<li>  <a href=\"#\" class=\"notificacoes\" \">Perigo: <span class=\"label label-
                danger\">" + descricao + "</span></a> </li>").insertBefore('#divisor');
    }
    qutNotificacoes.text(a);
    qutNotificacoes.show();
  };
```

Listagem 14 - Código JavaScript para que mostra a notificação

As notificações de prescrição de dívidas são realizadas por meio de uma rotina diária agendada para executar à 01:00h e que é repetida diariamente no mesmo horário. Essa rotina tem a função de verificar e notificar dívidas que ainda não foram executadas e que estão para prescrever. A notificação ocorre de duas formas: diariamente para dívidas que estão para prescrever nos próximos 15 dias e, duas vezes no mês, no primeiro e décimo quinto dia do

mês, para dívidas que estão para prescrever nos próximos 6 meses. Após a consulta no banco de dados das dívidas é gerado uma notificação para todos os usuários da tributação e do jurídico de cada dívida que está para prescrever.

A Listagem 15 apresenta o método agendarTarefa que utiliza as classe *Timer* para agendar uma tarefa. Essa classe realiza o agendamento de uma tarefa que seja uma instância da classe *TimerTask* que possui vários métodos de agendamento de tarefas, podendo definir a data do início da tarefa e um tempo em milissegundos para a repetição dessa tarefa. A classe *TimerTask* implementa a interface *Runnable*, usada para execução de *threads*. Para criar uma tarefa deve-se estender a Classe *TimerTask* na classe da tarefa e implementar o método *run()*, que deve conter a lógica da tarefa, pois é ele que é chamado quando der o tempo definido pela agendamento da classe *Timer*.

```
public void agendarTarefa() {
    if (estaAgentado != null && !estaAgentado) {
        TimerTaskNotificacoes task = new TimerTaskNotificacoes();
        timer = new Timer();
        Date date = new Date();
        Calendar hoje = Calendar.getInstance();
        hoje.setTime(date);
        hoje.add(Calendar.DAY_OF_MONTH, +1);
        int ano = hoje.get(Calendar.YEAR);
        int mes = hoje.get(Calendar.MONTH);
        int dia = hoje.get(Calendar.DAY_OF_MONTH);
        hoje.set(ano, mes, dia, 1, 0);
        timer.scheduleAtFixedRate(task, hoje.getTime(), 86400000L);
        estaAgentado = true;
    }
}
```

Listagem 15 - Método de agendar tarefa

A implantação do sistema ocorreu juntamente com o apoio do departamento de Central de Processamento de Dados (CPD) da prefeitura municipal de Francisco Beltrão. O departamento ficou responsável pela instalação do servidor ApacheTomcat e do servidor de banco de dados MySqlServer. O próprio departamento forneceu ao desenvolvedor o endereço e senha de acesso ao banco de dados que foram configurados no sistema. Para implantação gerou-se um arquivo compilado no formato .war do código fonte do sistema que foi utilizado pelo CPD para fazer o *deploy* no servidor.

4 CONSIDERAÇÕES FINAIS

Este trabalho apresentou o desenvolvimento de uma ferramenta *web* de apoio à cobrança da dívida ativa que utiliza um complexo gerenciamento de dívidas e notificações para facilitar a organização e controle da dívida de órgãos públicos, proporcionando a otimização da cobrança da dívida ativa e a redução da perda de receitas por meio da prescrição.

O *framework* VRaptor IV juntamente com o Hibernate se mostraram importantes ferramentas para o desenvolvimento ágil de um sistema *web* em Java. O auxílio de bibliotecas como o Bootstrap 3 e JQuery foi fundamental para superar a dificuldade no desenvolvimento de interfaces intuitivas e dinâmicas, agilizando, assim, o desenvolvimento do trabalho.

O projeto está sendo utilizado pela prefeitura de Francisco Beltrão e tem se demonstrado eficaz para agilizar e organizar a cobrança da dívida do município. A versão implantada encontra-se em teste e, com isso, obteve-se os objetivos propostos. Entretanto, ainda faltam realizar testes mais abrangentes, como, por exemplo, o uso simultâneo de mais usuários para verificar a sua estabilidade e concorrência.

Por se tratar de um sistema aplicado a gestão pública, em uma área que é pouco difundida entre os pesquisadores e desenvolvedores de sistemas e tecnologias, espera-se instigar mais pesquisas que procurem aliar tecnologia e gestão pública e que resultem no desenvolvimento de novas ferramentas que venham a suprir a grande necessidade de modernização do serviço público e, assim, facilitar os desafios do cotidiano dos servidores e, assim proporcionar aos servidores maior agilidade na prestação de serviços para a população.

REFERÊNCIAS

BRASIL. **Lei n. 5.172**, de 25 de outubro de 1966. Dispõe sobre o Sistema Tributário Nacional e institui normas gerais de direito tributário aplicáveis à União, Estados e Municípios. Disponível em: <http://www.planalto.gov.br/ccivil_03/leis/L5172Compilado.htm>. Acesso em: 05 nov. 2017.

BRASIL. **Lei n. 10.406**, de 10 de janeiro de 2002. Institui o Código Civil. Disponível em: <http://www.planalto.gov.br/ccivil_03/leis/2002/L10406.htm>. Acesso em: 05 nov. 2017.

CAVALCANTI, Lucas. **VRaptor**: Desenvolvimento Ágil para Web com Java. São Paulo: Casa do Código, 2014

PÁRANA, **Transparência**: Demonstrativo de Receitas, Disponível em: <www.transparencia.pr.gov.br>. Acesso em: 14 fev. 2017.

PARANÁ. **Lei n. 18.660**, de 22 de dezembro de 2015. Lei orçamentária anual exercício de 2016 do Estado do Paraná. Disponível em: <http://www.fazenda.pr.gov.br/arquivos/File/Orcamento/LOA/LOA2016.pdf>. Acesso em: 05 nov. 2017.

SANTANA, Rafael Gomes de. **Protesto de certidões de dívida ativa**. Revista Jus Navigandi, ISSN 1518-4862, Teresina, ano 15, n. 2534, 9 jun. 2010. Disponível em: <https://jus.com.br/artigos/14990>. Acesso em: 5 nov. 2017.

SOMMERVILLE, Ian. **Engenharia de software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011. 529 p. Tradução de: Ivan Bosnic e Kalinka G. de O. Gonçalves.

APÊNDICES

APÊNDICE A

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
   <persistence-unit name="SisCDA_PU" transaction-type="RESOURCE_LOCAL">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <class>br.com.apf.siscda.model.Anotacao</class>
      <class>br.com.apf.siscda.model.Divida</class>
      <class>br.com.apf.siscda.model.DividaDiversas</class>
      <class>br.com.apf.siscda.model.DividaImoveis</class>
      <class>br.com.apf.siscda.model.Notificacao</class>
      <class>br.com.apf.siscda.model.SMTPEmail</class>
      <class>br.com.apf.siscda.model.Usuario</class>
      <properties>
         <property name="javax.persistence.jdbc.url"
         value="jdbc:mysql://localhost:3306/siscdadb?zeroDateTimeBehavior=convertToNull"
         />
         <property name="javax.persistence.jdbc.user" value="root"/>
         <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
         <property name="javax.persistence.jdbc.password" value="123456"/>
         <property name="hibernate.cache.provider_class"
          value="org.hibernate.cache.NoCacheProvider"/>
         <property name="javax.persistence.schema-generation.database.action"
          value="create"/>
         <property name="hibernate.show_sql" value="true"/>
         <property name="hibernate.format_sql" value="true"/>
         <property name="use_sql_comments" value="true"/>
      </properties>
   </persistence-unit>
</persistence>
```

APÊNDICE B

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>br.com.apf</groupId>
  <artifactId>SisCDA</artifactId>
  <version>1.0</version>
  <packaging>war</packaging>
  <name>SisCDA</name>
  <properties>
    <endorsed.dir>${project.build.directory}/endorsed</endorsed.dir>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>br.com.caelum</groupId>
      <artifactId>vraptor</artifactId>
      <version>4.2.0-RC3</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.weld.servlet</groupId>
      <artifactId>weld-servlet-core</artifactId>
      <version>2.3.5.Final</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.weld</groupId>
      <artifactId>weld-core-impl</artifactId>
      <version>2.3.5.Final</version>
    </dependency>
    <dependency>
      <groupId>javax.el</groupId>
      <artifactId>el-api</artifactId>
      <version>2.2</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
      <version>5.1.0.Final</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-jpamodelgen</artifactId>
      <version>5.1.0.Final</version>
    </dependency>
    <dependency>
      <groupId>javassist</groupId>
      <artifactId>javassist</artifactId>
      <version>3.12.1.GA</version>
    </dependency>
    <dependency>
      <groupId>javax</groupId>
      <artifactId>javaee-web-api</artifactId>
      <version>7.0</version>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
```

```
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
    </dependency>
    <dependency>
        <groupId>jstl</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.7.21</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-validator-cdi</artifactId>
        <version>5.2.4.Final</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.44</version>
    </dependency>
    <dependency>
        <groupId>commons-fileupload</groupId>
        <artifactId>commons-fileupload</artifactId>
        <version>1.3.1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi</artifactId>
        <version>3.14</version>
    </dependency>
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-email</artifactId>
        <version>1.4</version>
    </dependency>
</dependencies>
<build>
    <finalName>SisCDA</finalName>
    <outputDirectory>src/main/webapp/WEB-INF/classes</outputDirectory>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.1</version>
            <configuration>
                <source>1.7</source>
                <target>1.7</target>
                <encoding>UTF-8</encoding>
                <compilerArguments>
                    <endorseddirs>${endorsed.dir}</endorseddirs>
                </compilerArguments>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>2.3</version>
```

```
            <configuration>
               <failOnMissingWebXml>false</failOnMissingWebXml>
            </configuration>
         </plugin>
         <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-dependency-plugin</artifactId>
            <version>2.6</version>
            <executions>
               <execution>
                  <phase>validate</phase>
                  <goals>
                     <goal>copy</goal>
                  </goals>
                  <configuration>
                     <outputDirectory>${endorsed.dir}</outputDirectory>
                     <silent>true</silent>
                     <artifactItems>
                        <artifactItem>
                           <groupId>javax</groupId>
                           <artifactId>javaee-endorsed-api</artifactId>
                           <version>7.0</version>
                           <type>jar</type>
                        </artifactItem>
                     </artifactItems>
                  </configuration>
               </execution>
            </executions>
         </plugin>
      </plugins>
   </build>
</project>
```