UNIVERSIDADE DO EXTREMO SUL CATARINENSE - UNESC

CURSO DE CIÊNCIA DA COMPUTAÇÃO

ELIEL FRASSON WALTRICK

PROTÓTIPO DE UM FRAMEWORK DE COMPONENTES JAVASERVER FACES PARA CRIAÇÃO DE SISTEMAS DE INFORMAÇÕES GEOGRÁFICAS UTILIZANDO OPENLAYERS 3

CRICIÚMA

2017

ELIEL FRASSON WALTRICK

PROTÓTIPO DE UM FRAMEWORK DE COMPONENTES JAVASERVER FACES PARA CRIAÇÃO DE SISTEMAS DE INFORMAÇÕES GEOGRÁFICAS UTILIZANDO OPENLAYERS 3

Trabalho de Conclusão de Curso apresentado para obtenção do grau de Bacharel no curso de Ciência da Computação da Universidade do Extremo Sul Catarinense, UNESC.

Orientador: Prof. Esp. Fabrício Giordani

CRICIÚMA

2017

ELIEL FRASSON WALTRICK

PROTÓTIPO DE UM FRAMEWORK DE COMPONENTES JAVASERVER FACES PARA CRIAÇÃO DE SISTEMAS DE INFORMAÇÕES GEOGRÁFICAS UTILIZANDO OPENLAYERS 3

Trabalho de Conclusão de Curso aprovado pela Banca Examinadora para obtenção do Grau de Bacharel, no Curso de Ciência da Computação da Universidade do Extremo Sul Catarinense, UNESC, com Linha de Pesquisa em Engenharia de Software

Criciúma, 21 de junho de 2017.

BANCA EXAMINADORA

Prof. Esp. Fabrício Giordani - UNESC - Orientador

Prof. MSc. Leila Laís Gonçalves - UNESC

Prof. Esp. Gilberto Vieira da Silva - UNESC

Dedico este trabalho à minha família, pelo apoio e incentivo, e a todos que contribuíram de alguma forma.

AGRADECIMENTOS

Agradeço aos meus pais, Pedro e Tereza, pelo apoio e incentivo durante todos os momentos que contribuíram para a minha formação pessoal e acadêmica.

Ao meu orientador professor Fabrício Giordani, por compartilhar o seu conhecimento que foi fundamental para a conclusão deste trabalho.

Aos colegas de universidade e a todos que contribuíram de maneira direta ou indireta durante esta etapa.

RESUMO

Nos dias atuais existe uma grande demanda por sistemas computacionais que possibilitam analisar e gerenciar o espaço físico. Tais sistemas, denominados Sistemas de Informações Geográficas, têm se tornado ferramenta de grande importância, tanto para profissionais da área quanto para cidadãos. No entanto, observa-se que a criação destes sistemas envolve um grau relevante de dificuldade, devido a grande quantidade de conceitos e tecnologias envolvidas. Portanto, foi proposta a criação de um framework que permita criar SIGs web utilizando componentes JSF, facilitando assim a criação de um sistema geoespacial ao utilizar um ambiente Java EE. Iniciou-se o trabalho fazendo um levantamento de quais as principais funcionalidades necessárias em um SIG. Concluída esta etapa, deu-se início à criação dos componentes, onde foram utilizados recursos da especificação JSF e da biblioteca OpenLayers. Foram então realizados testes para garantir o funcionamento do protótipo, que demonstraram resultados satisfatórios. Concluiu-se que foi possível atingir o objetivo proposto, tendo como resultado os componentes JSF que reduzem em até 70% a quantidade de linhas de código comparado com uma implementação JavaScript.

Palavras-chave: OpenLayers. JavaServer Faces. Framework. SIG.

ABSTRACT

There has been an increasing demand for systems that allow the management and analysis of the physical space in recent years. Such systems are called Geographic Information Systems, and they have become an important tool for both professionals and regular citizens. However, a certain degree of difficulty is observed in the development of these systems, due to the number of concepts and technologies of which knowledge is required. Therefore, it was proprosed the creation of a framework that enables the development of web GIS utilizing JSF components, thus facilitating the conception of geospatial systems while using a Java EE environment. First, a survey was conducted to determine the main features needed in a GIS. Subsequently, the components were built, wherein the OpenLayers library and the JSF specification were used. Tests were conducted to ensure the proper functioning of the prototype, which showed satisfactory results. In conclusion, the objectives were achieved, resulting in JSF components which require up to 70% less lines of code compared to a JavaScript implementation.

Keywords: OpenLayers. JavaServer Faces. Framework. GIS.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
GML	*Geography Markup Language*
HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transfer Protocol*
JEE	*Java Enterprise Edition*
JSF	*JavaServer Faces*
JVM	*Java Virtual Machine*
JPEG	*Joint Photographic Experts Group*
JSON	*Javascript Object Notation*
OGC	*Open Geospatial Consortium*
PNG	*Portable Network Graphics*
SFA	*Simple Feature Access*
SVG	*Scalable Vector Graphics*
SRS	*Spatial Reference System*
SGBD	Sistema de Gerenciamento de Banco de Dados
SIG	Sistema de Informação Geográfica
SQL	*Structured Query Language*
TIFF	*Tagged Image File Format*
WKT	*Well-known Text*
WKB	*Well-known Binary*
WMS	*Web Map Service*
WFS	*Web Feature Service*
WCS	*Web Coverage Service*
XML	*Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

O crescimento do poder de processamento dos computadores, aliado aos progressos nas tecnologias de monitores gráficos, permitiram, há algumas décadas atrás, o desenvolvimento dos primeiros Sistemas de Informações Geográficas (SIG). Desde então, os SIG têm se demonstrado cada vez mais importantes em diversas áreas do conhecimento humano, como gerenciamento de recursos territoriais, planejamento urbano, transportes, marketing, entre outros (BONHAM-CARTER, 2014, tradução nossa).

A integração das tecnologias da Web 2.0 com os SIG resultaram no advento do *web mapping*, que se tornou presente em grande parte dos serviços online. Fontes abertas de dados geográficos, como o projeto colaborativo OpenStreetMap, em conjunto com demais projetos *open source*, contribuíram para o desenvolvimento de inúmeras novas tecnologias para criação de mapas em ambiente web, resultando assim em sistemas cada vez mais modernos e complexos (BALLATORE et al, 2011, tradução nossa).

A biblioteca OpenLayers é uma das mais importantes tecnologias *client-side* para *web mapping* por ser uma das mais completas e estáveis (LV, 2016).

Entretanto, o desenvolvimento de aplicações geoespaciais geralmente torna-se complexo para os desenvolvedores devido a grande quantidade de tecnologias envolvidas. Desta maneira, ao criar uma aplicação utilizando JavaServer Faces (JSF) e OpenLayers é necessário criar métodos de integração entre estas duas tecnologias, que podem vir a ser complexos e de difícil manutenção.

Levando em consideração a complexidade da construção de um SIG web, o objetivo deste trabalho é criar um *framework* de componentes JSF para criação de aplicações geoespaciais com a biblioteca OpenLayers 3, com o intuito de facilitar e agilizar o desenvolvimento de projetos que utilizam estas tecnologias.

1.1 OBJETIVO GERAL

Desenvolver um protótipo de um *framework* de componentes JSF sobre a biblioteca OpenLayers 3 com o intuito de facilitar o desenvolvimento de SIGs web ao integrar estas tecnologias.

1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos desta pesquisa consistem em:

a) estudar o processo de desenvolvimento de um SIG;
b) compreender o funcionamento da biblioteca OpenLayers aplicado ao desenvolvimento de um SIG;
c) compreender o processo de criação de componentes JSF;
d) definir os componentes a serem criados para o protótipo;
e) desenvolver protótipo do *framework*.

1.3 JUSTIFICATIVA

Cada vez mais os gestores precisam de ferramentas e recursos eficazes para auxiliar na tomada de decisões diminuindo, dessa forma, os riscos inerentes a esse processo. Nesse sentido, os sistemas de informação geográfica são de suma relevância em cenários nos quais a localização ou propriedades geográficas determinam o sucesso ou não de determinados empreendimentos (OLIVEIRA, 2009).

Contudo, o desenvolvimento de SIGs Web é trabalhoso, devido à quantidade de conceitos e tecnologias envolvidas. São necessários softwares especializados para tratamento de dados, extensões para bancos de dados, bibliotecas, além de conhecimentos na área de cartografia. Além disso, cada área de aplicação possui diferentes requisitos em relação à construção do software (BONHAM-CARTER, 2014, tradução nossa).

Portanto, ao desenvolver um SIG Web utilizando JSF o desenvolvedor precisa criar seus próprios métodos para integrar as tecnologias, trabalho este que seria facilitado ao utilizar um *framework* já pronto e devidamente testado.

Atualmente existem *frameworks* que possibilitam a integração entre JSF e OpenLayers, porém os mesmos utilizam versões antigas desta biblioteca, sendo que existem versões mais atuais que utilizam técnicas inovadoras e garantem uma melhor experiência tanto ao usuário final quanto ao desenvolvedor.

Além disso, a atualização da tecnologia existente não é tarefa trivial, visto que não existe retrocompatibilidade entre as versões da biblioteca OpenLayers (GRATIER; SPENCER; HAZZARD, 2015).

Desta forma, a criação de um protótipo de um *framework* de componentes JSF sobre a biblioteca OpenLayers versão 3 torna-se mais viável do que uma possível atualização dos *frameworks* existentes.

A integração entre JSF e OpenLayers 3 permitirá o desenvolvimento de SIGs Web através de componentes JSF, abstraindo assim boa parte do código JavaScript necessário, além de ajudar a diminuir a quantidade de erros e aumentar a eficiência no desenvolvimento dos projetos.

1.4 ESTRUTURA DO TRABALHO

Este trabalho é constituído por oito capítulos onde são abordados os conceitos e metodologias utilizados. No primeiro capítulo é apresentada uma introdução acerca do tema de pesquisa, são apresentados em seguida os objetivos e por fim a justificativa para o desenvolvimento deste trabalho.

O segundo capítulo aborda de maneira geral os Sistemas de Informações Geográficas e os principais conceitos que servem de fundamento para a existência de tal tecnologia.

O terceiro capítulo trata sobre a representação computacional de dados geográficos, nele são apresentadas as técnicas utilizadas para representar objetos geográficos em um ambiente computacional.

No capítulo quatro são introduzidas as principais tecnologias e padrões utilizados no desenvolvimento de Sistemas de Informações Geográficas.

No capítulo cinco são apresentadas as tecnologias que serviram de base para o desenvolvimento deste trabalho.

O capítulo seis trata dos trabalhos que possuem algum tipo de relação a este projeto de pesquisa.

No capítulo sete é apresentado o protótipo desenvolvido, incluindo as etapas de desenvolvimento, testes, e resultados obtidos.

Finalmente, o capítulo oito traz as conclusões sobre o tema aqui abordado. São relatadas as dificuldades encontradas e sugestões para trabalhos futuros.

2 SISTEMAS DE INFORMAÇÕES GEOGRÁFICAS

Desde as primeiras sociedades humanas a confecção de mapas sempre foi uma atividade muito importante. A localização geográfica de rios, terras aráveis, animais, e determinados pontos de referência foram essenciais para garantir a sobrevivência destas sociedades primitivas. Evoluções nas áreas de Cartografia e Ciência da Computação permitiram, em meados dos anos sessenta, o surgimento dos primeiros mapas digitais e, consequentemente, o surgimento dos primeiros Sistemas de Informações Geográficas.

Um Sistema de Informação Geográfica pode ser definido como um sistema de informação que armazena informações não somente de eventos, atividades, e coisas, mas também onde estes eventos, atividades, e coisas existem ou acontecem (LONGLEY, 2005, tradução nossa). Estes sistemas proporcionam, então, a integração de dados de diversas fontes, como mapas, fotos aéreas, dados de censos, além de possibilitar a realização de análises complexas para soluções de problemas de diversas áreas.

Segundo Konecny (2014), Sistemas de Informações Geográficas surgiram das atividades de quatro áreas:

a) cartografia, que tentou automatizar o processo manual de elaboração de mapas ao substituir os desenhos manuais pela digitalização vetorial;
b) computação gráfica, que tinha muitas aplicações de dados vetoriais digitais à parte da cartografia, especialmente na modelagem de edifícios, máquinas, e outros tipos de instalações;
c) bancos de dados, que criaram uma estrutura matemática que permitiu que os problemas da Computação Gráfica e da Cartografia Digital pudessem ser resolvidos;
d) sensoriamento remoto, que gerou grandes quantidades de imagem digitais que precisavam ser georreferenciadas e analisadas.

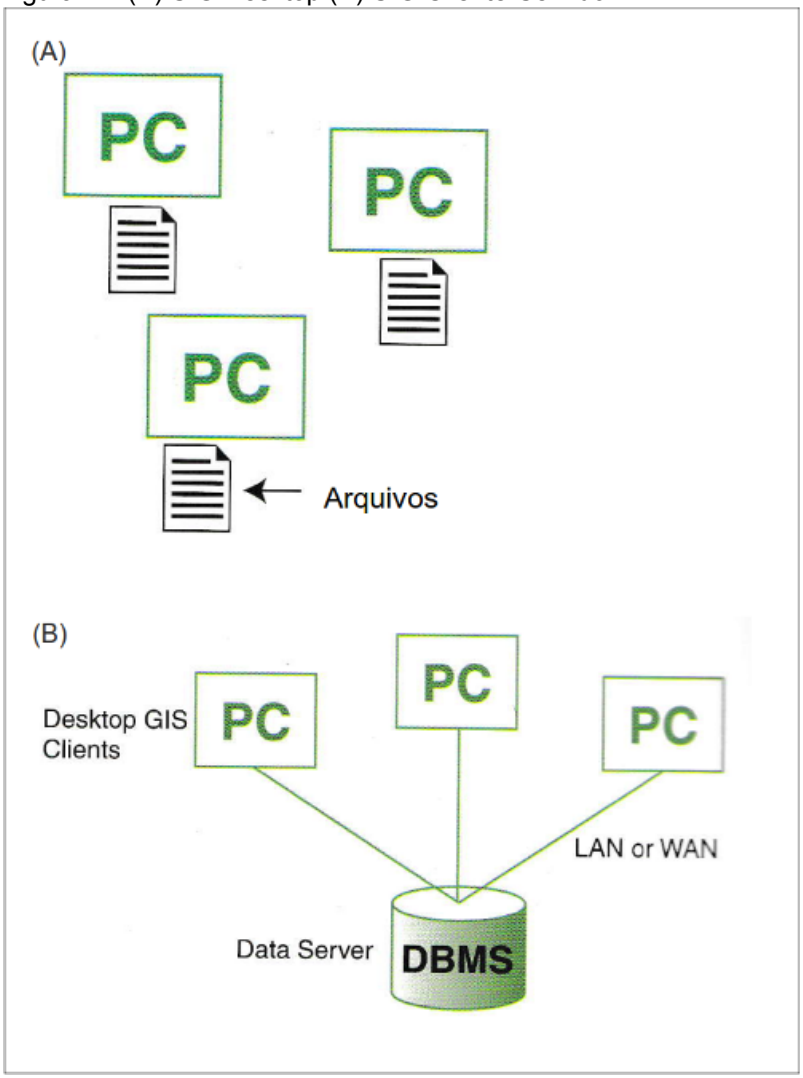
Neste capítulo serão apresentados os conceitos básicos de Sistemas de Informações Geográficas e as técnicas que tornam possíveis a representação de mapas em ambiente computacional.

2.1 ESTRUTURA DE UM SIG

Os três componentes básicos de um SIG são: a interface do usuário, as rotinas de processamento, e o banco de dados. A interface do usuário consiste em uma coleção de menus, botões, campos para entrada de dados, e outros controles que fornecem acesso às rotinas de processamento. Estas, por sua vez, possuem funções para manipulação e processamento dos dados geográficos, além de permitirem acesso ao banco de dados. Por fim, o banco de dados possui tipos de dados projetados especificamente para o armazenamento dos objetos geográficos. Estes três componentes constituem uma arquitetura de três camadas denominadas camada de apresentação, camada de lógica de negócio, e camada de armazenamento de dados (LONGLEY, 2005, tradução nossa).

Um SIG pode ser configurado de diversas maneiras. As configurações mais utilizadas atualmente são *desktop* e cliente-servidor. Na configuração *desktop*, todas as camadas são instaladas em um único computador. Uma das principais vantagens desta configuração é a maior rapidez no processamento dos dados geográficos, visto que ela não é dependente de conexões de rede. Na configuração cliente-servidor, os usuários possuem apenas a camada de apresentação e os dados se encontram em um servidor remoto, garantindo assim maior flexibilidade do que a configuração *desktop*, pois os dados podem ficar centralizados em um único local acessível por todos os usuários. Na figura 1 é possível observar os dois tipos de configuração. Na primeira todos os recursos necessários para o funcionamento do SIG encontram-se em cada computador. Na segunda, os computadores possuem apenas o cliente SIG, e o banco de dados se encontra em local remoto.

Figura 1 – (A) SIG Desktop (B) SIG Cliente-Servidor



Fonte: Longley (2005).

Para que a camada de apresentação possa exibir cada objeto geográfico em sua posição correta é necessário que adote-se uma referência geográfica que seja comum a todos os dados. Este processo é conhecido como georreferenciamento.

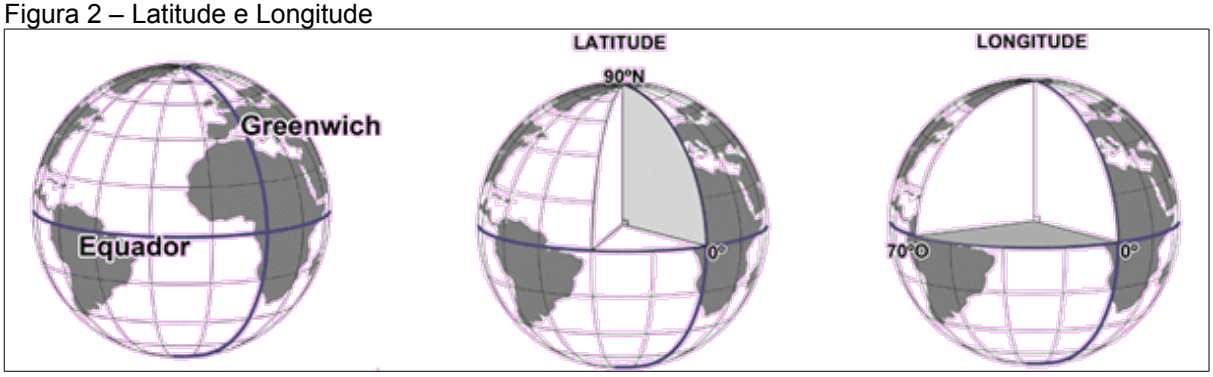
2.2 GEORREFERENCIAMENTO

Para que um objeto geográfico possa existir no contexto de um SIG, é preciso que esteja atrelado a ele uma referência geográfica. Vários sistemas de referências geográficas foram inventados no decorrer da evolução da humanidade, sendo o mais básico de todos o sistema de nomes. O sistema de nomes é um exemplo de um sistema de baixa precisão, pois a partir dele é possível dar nomes a

áreas de grande extensão, como países, cidades, ruas, entre outros. Ao surgir a necessidade de localizar objetos com maior precisão, como um lote ou uma edificação, é preciso utilizar sistemas de alta precisão, sendo que um dos mais utilizados é o sistema de coordenadas em latitude e longitude (LONGLEY, 2005, tradução nossa).

O sistema de coordenadas em latitude e longitude possui quatro elementos essenciais: a Linha do Equador, o Meridiano de Greenwich, os Paralelos e os Meridianos. Os paralelos são linhas imaginárias paralelas ao Equador que circundam a superfície terrestre, e que vão de 0 a 90 graus para o norte e para o sul. A partir dos paralelos é possível obter a latitude de um ponto na superfície da Terra, que é definida como o ângulo entre o plano Equatorial e uma linha imaginária reta que passa entre o ponto e o centro da Terra. Os Meridianos são linhas imaginárias que ligam os polos norte e sul e cruzam com as latitudes, indo de 0 a 180 graus a partir do Meridiano de Greenwich. A partir do conceito de meridiano pode-se obter a longitude de um ponto, que é definida como o ângulo formado entre o próprio ponto e o meridiano de origem, variando entre 0 e 180 graus nas direções leste ou oeste (FITZ, 2008). Na figura 2 é possível observar a representação gráfica destes conceitos.

Figura 2 – Latitude e Longitude



Fonte: Quoos (2016).

2.3 PROJEÇÕES

Um problema fundamental na elaboração de mapas é a transformação da superfície curva e tridimensional da Terra em uma superfície plana e bidimensional, como papel ou a tela de um computador. Esta transformação sempre

implica em deformações, por isso pode-se dizer que um mapa é apenas uma representação aproximada de uma determinada área da superfície do planeta.

Desta maneira, determinados pontos na superfície da terra são transportados para o plano de projeção do mapa através de Projeções Cartográficas, que são apoiadas em funções matemáticas. Existe uma grande quantidade de diferentes projeções cartográficas, sendo assim, deve-se escolher a projeção que é mais adequada para o determinado fim, levando em conta as características que mantêm-se fiéis e as características que são deformadas (FITZ, 2008).

Segundo Fitz (2008), as projeções podem ser classificadas nos seguintes tipos: conformes, equivalentes, e equidistantes.

Projeções Conformes são aquelas que mantém as formas ou os ângulos de pequenos objetos. A manutenção dos ângulos acarreta na distorção do tamanho dos objetos. As projeções Mercator e Universal Transverse Mercator (UTM) possuem estas características.

Projeções Equivalentes são as que conservam as áreas. Como consequência, há deformação dos ângulos. São consideradas as mais adequadas para uso em SIG. Exemplos: Azimutal de Lambert.

Projeções Equidistantes são aquelas que conservam a proporção entre as distâncias em determinadas direções. Exemplo: Projeção Cilíndrica Equidistante.

Foram aqui abordados conceitos básicos de projeções para melhor entendimento do presente trabalho. Para maior aprofundamento sobre o tema de projeções cartográficas recomenda-se a leitura de Fitz (2008).

3 REPRESENTAÇÃO COMPUTACIONAL DE DADOS GEOGRÁFICOS

Para abordar o problema da representação de dados geográficos em ambiente computacional, Câmara (2005) utiliza o paradigma dos quatro universos: universo ontológico, universo formal, universo estrutural, e universo implementado, onde cada universo é um nível de abstração para representar conceitos do mundo real.

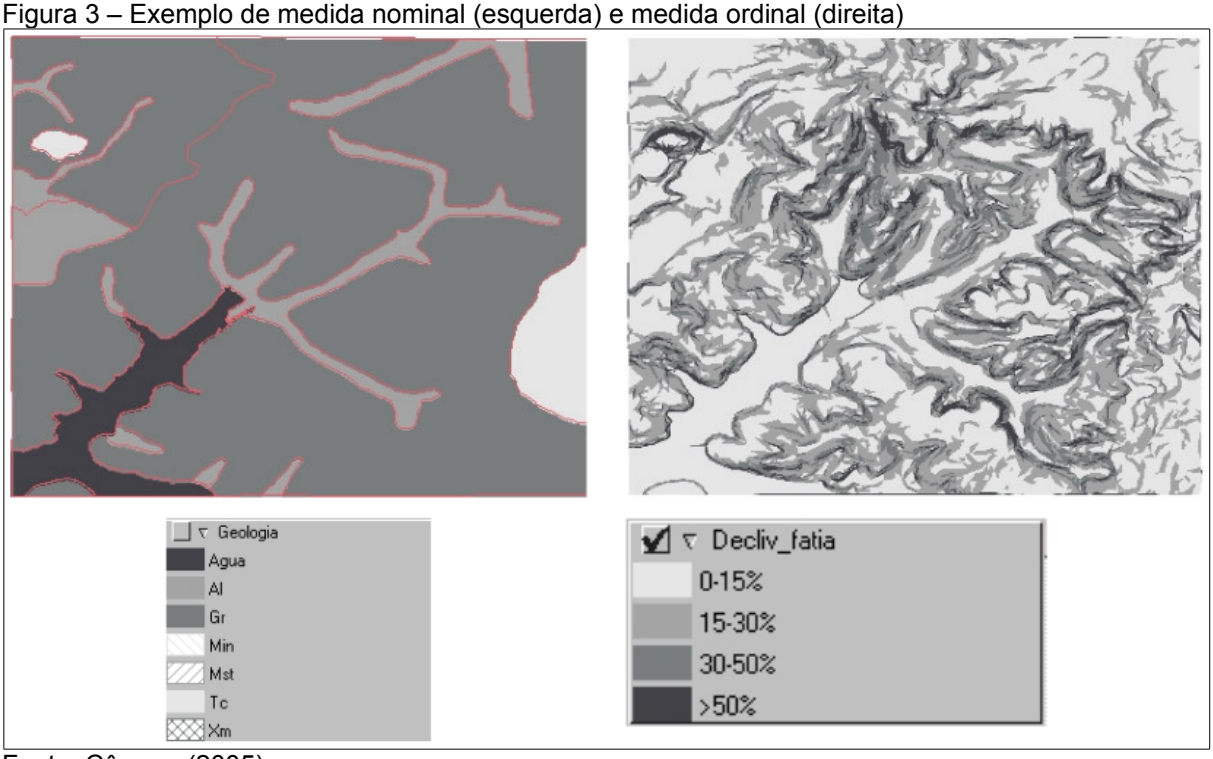
3.1 PARADIGMA DOS QUATRO UNIVERSOS

O universo ontológico trata de conceitualizar objetos e fenômenos do mundo real. Os conceitos associados a entidades geográficas podem ser divididos em dois tipos: físicos e sociais. Dentre os conceitos físicos estão aqueles que possuem fronteiras bem definidas na natureza, como montanhas e vales. Se encaixam também dentro dos conceitos físicos as topografias físicas, como altimetria e declividade. Já os conceitos sociais são aqueles que são criados por ações humanas, como lotes, municípios, países, entre outros. Dentro deste universo podem ser respondidas questões como: Quais entidades podem ser utilizadas para descrever o problema? (CÂMARA, 2005).

No universo formal, modelos lógicos e matemáticos são utilizados para formalizar os conceitos do universo ontológico. Este universo é uma espécie de intermediário entre a imprecisão do mundo real e a estrutura rígida dos computadores, portanto, uma das questões centrais é como medir os fenômenos do mundo real. Stevens (1946 apud Câmara, 2005) propôs quatro escalas de mensuração: nominal, ordinal, intervalo, e razão.

As escalas nominal e ordinal são ditas temáticas, pois são associados rótulos a determinados objetos, separando-os em classes distintas. Na escala nominal as classes não possuem nenhuma ordem específica, já a escala ordinal possui uma ordem inerente, como é demonstrado na figura 3.

Figura 3 – Exemplo de medida nominal (esquerda) e medida ordinal (direita)



Fonte: Câmara (2005).

Em contraste com as escalas temáticas, as escalas por intervalo e por razão são baseadas em números reais. A escala por intervalo, como o próprio nome sugere, possui intervalos regulares, possui um zero definido por convenção, e possui uma faixa de medidas que vai do infinito negativo ao infinito positivo. Como exemplo temos a escala Celsius. Por outro lado, a escala por razão não possui um zero arbitrário, mas sim um zero que representa a ausência do valor a ser medido. O peso de um objeto, por exemplo, faz parte de uma escala por razão, pois não existe peso negativo (KIRCH, 2008, tradução nossa).

Após os modelos serem definidos no universo formal, a próxima abstração é o universo estrutural, onde algoritmos são aplicados aos modelos formais para mapeá-los a estruturas de dados definidas. Estruturas de dados Vetoriais e Matriciais incluem-se neste universo.

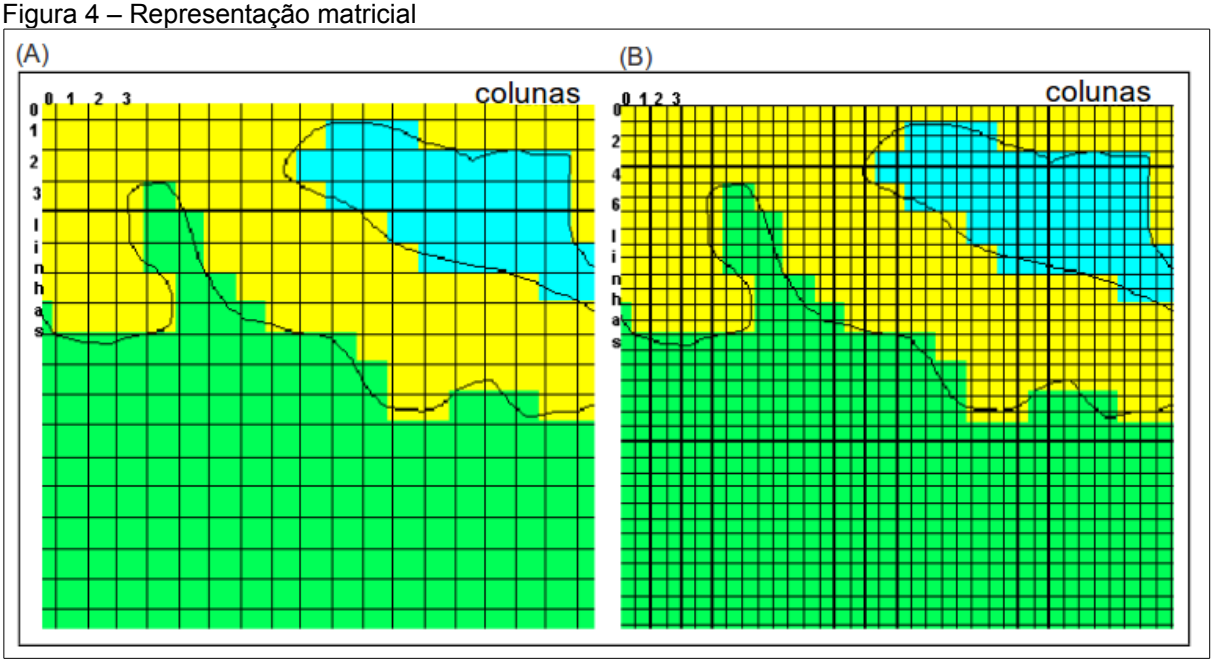
Por fim, o universo implementado trata da implementação dos sistemas, abordando questões como a arquitetura e as linguagens de programação a serem utilizadas.

3.2 REPRESENTAÇÃO VETORIAL E MATRICIAL

Existem duas estruturas fundamentais utilizadas para representar dados geográficos em ambiente computacional: Representação Matricial e Representação Vetorial.

Na representação matricial, o espaço é dividido em uma matriz de células retangulares, onde cada célula, ou *pixel,* representa uma pequena porção do terreno, como é possível visualizar na figura 4.

Figura 4 – Representação matricial

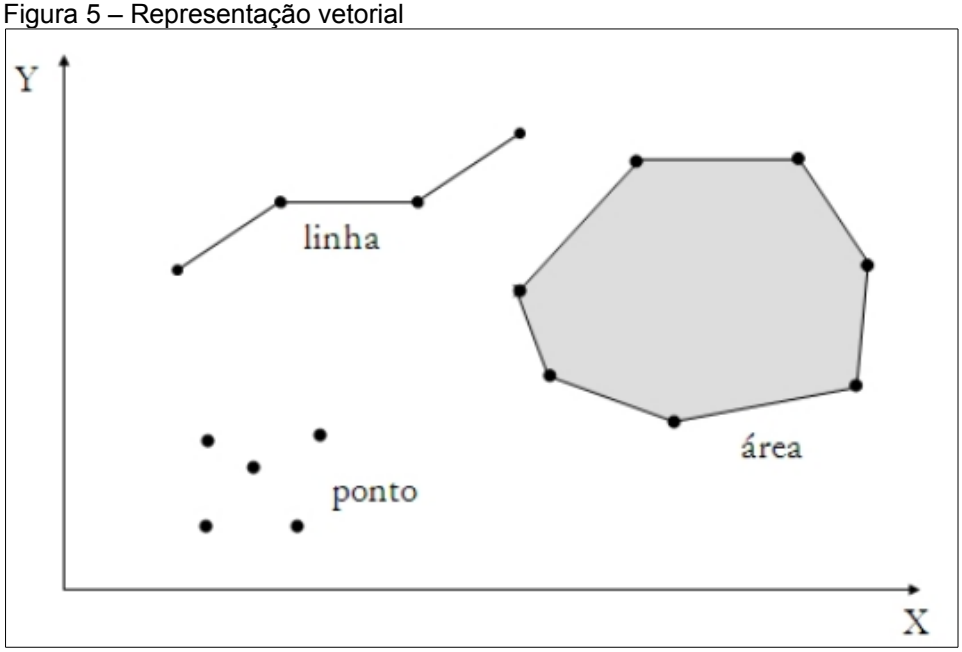


Fonte: Câmara (2005).

Na figura A é possível observar que quanto menor a área coberta por uma única célula, maior será a resolução da imagem, o que possibilita uma maior exatidão nas avaliações de áreas e distâncias. No entanto, quanto maior a resolução, mais espaço de armazenamento será necessário (CÂMARA, 2005).

Já na representação vetorial, pares de coordenadas são utilizados para representar a localização dos objetos geográficos. São três elementos gráficos principais: ponto, linha, e polígono (área), como é mostrado na figura 5.

Figura 5 – Representação vetorial



Fonte: Câmara (2005).

O ponto, o elemento gráfico mais básico, é definido por um par ordenado (x,y) de coordenadas. Pontos são geralmente utilizados para identificar fenômenos que não tem necessariamente uma forma geométrica, como a localização de um evento ou de uma cena de crime, de determinados grupos de animais, entre outros. As linhas são formadas através da conexão de um conjunto pontos e podem ser utilizadas para representar ruas, rios, entre outros. Os polígonos são as áreas resultantes da conexão de um conjunto de linhas, de tal maneira que o último ponto de uma linha esteja exatamente na mesma localização do primeiro ponto da linha seguinte. Polígonos podem ser adotados para a representação de municípios, bairros, lotes, edificações, ou qualquer entidade que possua uma forma poligonal (OLIVEIRA, 2009).

3.2.1 Formatos de dados vetoriais

Diversos formatos de dados vetoriais foram criados ao longo da evolução dos sistemas de informação geográfica, atualmente podem ser citados como os mais proeminentes os formatos *Shapefile*, GeoJSON, e *Geography Markup Language* (GML).

Shapefile é um formato de arquivo utilizado para armazenamento de dados vetoriais que desenvolvido pela empresa ESRI em 1997 e atualmente é um dos formatos mais populares para armazenamento de feições espaciais.

Um *Shapefile* armazena as geometrias e os atributos das feições espaciais de um conjunto de dados. A geometria de uma feição é armazenada utilizando um conjunto de coordenadas e os atributos são armazenados em um arquivo de formato dBASE. Pelo fato de não ser uma estrutura de dados topológica, este tipo de arquivo possui algumas vantagens sobre outras fontes de dados, como maior rapidez no desenho, capacidade editar as feições, e requerem menos espaço de armazenamento. *Shapefiles* suportam pontos, linhas, e polígonos (ESRI, 1998).

Shapefiles são compostos de vários arquivos, sendo que três são obrigatórios e os demais opcionais:

a) .shp – o arquivo principal, que armazena as geometrias das feições;
b) .shx – armazena um índice das feições para facilitar a busca;
c) .dbf – arquivo dBASE que armazena os atributos das feições;
d) .prj – armazena informações sobre o sistema de coordenadas (opcional);
e) .sbx ou .sbn – armazena um índice espacial das feições (opcional);
f) .fbx ou .fbn – armazena um índice espacial em modo somente leitura (opcional);
g) .shp.xml – metadados em formato XML (opcional).

Para a especificação completa deste tipo de arquivo recomenda-se a leitura de ESRI (1998).

GeoJSON é um padrão aberto e uma variante do padrão JSON. Este formato permite a representação de qualquer geometria, incluindo geometrias complexas como *MultiLineString*, *MultiPolygon* e *GeometryCollection*. Também é possível armazenar os atributos não-geográficos das feições e as projeções cartográficas associadas (WESTRA, 2015, tradução nossa). Na figura 6 é dado um exemplo de uma geometria em formato GeoJSON.

Figura 6 – Representação de um ponto em formato GeoJSON

```
{ "type": "Point",
    "coordinates": [30, 10]
}
```

Fonte: Do autor.

Geography Markup Language (GML) é um formato baseado em XML para armazenar feições espaciais em forma de texto. Este é um formato sofisticado que foi desenvolvido pelo *Open Geospatial Consortium* (OGC) e que recentemente passou a fazer parte dos padrões ISO (WESTRA, 2015, tradução nossa). Na figura 7 é dado um exemplo de uma geometria no formato GML.

Figura 7 – Representação de um ponto no formato GML

```
<gml:Point>
    <gml:pos>40.728185 -73.976411</gml:pos>
</gml:Point>
```

Fonte: Do autor.

3.2.2 Formatos de dados matriciais

Alguns dos formatos mais populares para armazenamento de dados matriciais (ou *raster*) georreferenciados são: ECW, JPEG2000, e GeoTIFF, sendo que GeoTIFF é o mais notável atualmente.

GeoTIFF é uma variante do formato *Tagged Image File Format* (TIFF) utilizado para armazenar dados matriciais. A especificação deste formato define um conjunto de *tags* para descrever informações cartográficas associadas com imagens TIFF provenientes de imagens de satélite, fotos aéreas, modelos de elevação, entre outros. Sendo assim, o propósito deste formato é associar uma referência espacial ou projeção cartográfica a uma imagem *raster*. (POON, 2007, tradução nossa).

4 TECNOLOGIAS PARA DESENVOLVIMENTO DE SIG

As tecnologias apresentadas a seguir são essenciais para o desenvolvimento de SIGs pois permitem armazenar, manipular, e disponibilizar os dados geográficos para visualização. Todas as tecnologias descritas neste capítulo fazem uso de padrões abertos, garantindo assim interoperabilidade e fácil integração.

4.1 PADRÕES OPEN GEOSPATIAL CONSORTIUM

O *Open Geospatial Consortium* (OGC) é a principal organização responsável pelo desenvolvimento e manutenção de padrões abertos para utilização em *softwares* geoespaciais. Fundado em 1994 por 8 colaboradores, atualmente conta com 523 membros que representam instituições governamentais, empresas privadas, e instituições acadêmicas (OGC, 2016).

Existem dezenas de padrões OGC que abrangem as mais diversas situações e necessidades. Para os fins propostos neste trabalho, três padrões são de suma importância: *Simple Feature Access* (SFA), que é uma especificação para bancos de dados geográficos; *Web Map Service* (WMS), um padrão para servidores de mapas; e *Web Feature Service* (WFS), um padrão para servidores de feições.

4.1.1 Simple Feature Access

Simple Feature Access (SFA) é um padrão para armazenamento e manipulação de dados geográficos. Este padrão especifica tipos de dados, formatos para armazenamento e operações em feições geográficas.

A classe *Geometry* serve de base para todas as feições que podem ser armazenadas em um banco de dados geográfico. Esta classe é abstrata e portanto não pode ser instanciada. Desta forma, todas as feições são instâncias de subclasses de *Geometry*. Todas as instâncias devem possuir um Sistema de Referência Espacial (SRS) associado a elas (GRONNING, 2013, tradução nossa).

Conforme Gronning (2013), quatro subclasses herdam diretamente de *Geometry: Point, Curve, Line, e Polygon*. As classes são descritas a seguir:

a) ***point***: a classe *Point* representa a geometria mais simples do padrão SFA. Ela consiste de uma coordenada x e y. É possível vincular duas coordenadas adicionais caso isso seja exigido pelo SRS associado. Os métodos disponíveis nesta classe são apenas aqueles que foram herdados da classe *Geometry*. Um objeto desta classe é adequado para representar coordenadas bidimensionais, como um par de coordenadas em latitude e longitude;

b) ***curve***: a classe *Curve* é uma classe abstrata que possibilita a representação de curvas através de uma coleção de pontos com um algoritmo de interpolação. Esta classe fornece métodos para medição do comprimento total da curva, além de métodos que permitem avaliar se a curva é fechada ou se ela é um anel;

c) ***line***: as classes *Line* e *LineString* representam linhas. A classe *LineString* é uma subclasse de *Curve*, e especifica uma interpolação linear para os seus pontos. *Line* é um caso especial de *LineString*: ela possui apenas dois pontos;

d) ***polygon***: a classe *Polygon* permite a representação de áreas ou polígonos. Um *Polygon* é uma superfície planar que possui uma única fronteira exterior e zero ou mais fronteiras interiores. Cada fronteira interior é um buraco no polígono. Para que um *Polygon* seja válido ele deve ser topologicamente fechado. Esta classe é adequada para representação de territórios como países ou municípios, além de outras entidades que possuem forma poligonal.

O Padrão SFA também define um conjunto de métodos para a classe *Geometry*. Estes métodos são divididos em três categorias (OGC, 2011):

a) **métodos básicos:** são os métodos que descrevem as características da geometria, como a sua dimensão, o seu SRS, o seu retângulo envolvente, entre outros. Também existem métodos para descrever a geometria em *Well Known Text* (WKT) ou *Well Known Binary* (WKB). WKT e WKB serão introduzidos na seção 4.1.1.1;

b) **métodos para testes de relação espacial:** estes métodos testam a relação espacial entre dois objetos geométricos, como igualdade, intersecção, sobreposição, entre outros;

c) **métodos para análise espacial:** esta categoria de métodos fornece análises espaciais como a distância entre duas geometrias, a diferença entre elas, o fecho convexo de uma ou mais geometrias, etc. A precisão das análises depende da precisão do SRS associado.

4.1.1.1 Well Known Text e Well Known Binary

Dois formatos são especificados pelo padrão SFA para armazenamento de dados: *Well Known Text* (WKT) e *Well Known Binary* (WKB). Estes formatos descrevem geometrias em forma textual e em forma binária.

O formato WKT é definido utilizando um Formalismo de Backus-Naur cuja especificação está além do escopo deste texto e portanto não será aqui apresentada. Para melhor entendimento deste formato, alguns exemplos podem ser visualizados na figura 8.

O formato WKB não é legível por seres humanos e por isso não serão apresentados exemplos. A especificação deste formato também está além do escopo deste trabalho e portanto não será apresentada. As especificações completas dos formatos WKT e WKB podem ser encontradas em OGC (2011).

Figura 8 – Exemplos de geometrias em formato WKT

Descrição	**Representação WKT**
Um Ponto na posição (10,10)	Point (10 10)
Uma LineString com três pontos	LineString (10 10, 20 20, 30 30)
Um Polygon com 1 fronteira exterior e 0 fronteiras interiores	Polygon (10 10, 10 20, 20 20, 20 15, 10 10)
Um MultiPolygon com 2 polígonos	MultiPolygon (((10 10, 10 20, 20 20, 20 15, 10 10)), ((60 60, 70 70, 80 65, 60 60)))

Fonte: Do autor.

4.1.2 Web Map Service

Web Map Service é um protocolo para transferência de mapas geoespaciais pela internet desenvolvido pelo OGC em 1999. Este protocolo faz uso do protocolo *Hypertext Transfer Protocol* (HTTP) para transferir os dados do servidor para o cliente. WMS é altamente eficiente pois transfere somente a porção do mapa que o usuário está visualizando, sem precisar transferir todo o mapa (CHHAJED, 2016, tradução nossa).

O protocolo WMS define três operações: *GetCapabilities*, *GetMap*, e *GeatFeatureInfo*. As operações são explicadas a seguir (GRONNING, 2013):

a) *GetCapabilities*: esta é a operação mais básica. Quando esta operação é executada, o servidor responderá com uma lista de metadados que descrevem as "habilidades" do servidor. Assim, o cliente poderá visualizar quais os dados geográficos estão disponíveis para visualização, as operações disponíveis, entre outros;
b) *GetMap*: esta é a principal operação do WMS. Ao executar esta operação o servidor produzirá um mapa que será enviado ao cliente. Através de parâmetros o cliente pode especificar as camadas que deseja, o SRS, o retângulo envolvente, o tamanho do mapa, o formato, entre outros;
c) *GetFeatureInfo*: esta operação permite que o cliente solicite ao servidor informações sobre uma determinada feição. Os parâmetros i e j, que são as coordenadas da feição, são necessários para esta operação.

Na figura 9 é possível observar um exemplo de uma operação *GetMap*. Neste exemplo o servidor WMS encontra-se na máquina local, denominada *localhost*. É possível observar vários parâmetros, sendo que: *request* é a operação a ser realizada, *service* é o tipo de serviço, *version* é a versão do protocolo WMS, *layers* são as camadas, *styles* são os estilos associados às camadas, srs é o sistema de referência espacial, bbox é o retângulo envolvente, *width* é a largura do mapa, *height* é a altura do mapa, e *format* é o formato da imagem.

Figura 9 – Exemplo de requisição GetMap

```
http://localhost/wms?
request=GetMap
&service=wms
&version=1.3.0
&layers=edificacoes
&styles=
&srs=EPSG%3A31982
&bbox=-130.0 ,18.2 , -36.0 ,41.3
&width=600
&height=400
&format=image%2Fpng
```

Fonte: Do autor.

Portanto, a operação da figura 9 retornará um mapa de edificações que se encontra dentro das coordenadas -130.0, 18.2, -36.0, 41.3, e que cujo sistema de referência espacial é EPSG:31982. O tamanho do mapa será 600x400 e o formato da imagem será PNG.

4.1.3 Web Feature Service

Web Feature Service (WFS) é um protocolo para transferência de feições. Ao passo que o protocolo WMS transfere mapas em formato JPEG ou PNG, o protocolo WFS transfere feições codificadas no formato GML. Isso o torna ideal para usuários que necessitam realizar análises mais complexas sobre os mapas, em contraste ao protocolo WMS que só possibilita a visualização dos mapas.

Para os fins propostos neste trabalho, duas operações WFS são relevantes:

a) *GetCapabilities*: esta operação é equivalente à *GetCapabilities* do WMS;

b) *GetFeature*: esta é a operação principal do WFS. Ela retorna uma feição ou um conjunto de feições de acordo com os parâmetros especificados. Alguns dos parâmetros principais são: typeName, featureID, maxFeatures. Sendo que typeName é o nome da camada, featureID é código de identificação da feição, e maxFeatures é a quantidade máxima de feições a serem retornadas (GRONNING, 2013).

4.2 SERVIDOR DE DADOS GEOESPACIAIS

Os servidores de dados geoespaciais têm a função de disponibilizar os dados solicitados pelos clientes através de padrões estabelecidos. Nesta seção serão expostos os principais servidores de dados geoespaciais que existem atualmente no mercado.

4.2.1 Geoserver

Geoserver é um servidor de mapas de código-fonte aberto escrito em Java que permite compartilhamento de dados geoespaciais. Foi projetado para interoperabilidade e possibilita a publicação de dados geoespaciais através de padrões abertos. (GEOSERVER, 2016).

Por ser um projeto mantido pela *Open Source Geospatial Foundation* (OSGeo), o Geoserver é desenvolvido e testado por uma grande comunidade de indivíduos e organizações. Atualmente serve como a implementação de referência do padrão WFS, além de possuir implementações dos padrões WMS e *Web Coverage Service* (WCS).

Destacam-se entre as principais características do Geoserver (SILVA, 2011):

a) interface de administração via web (figura 10);
b) suporte às principais fontes de dados espaciais, como PostGIS, Oracle Spatial, Shapefile, entre outros;
c) formatos JPEG, GIF, PNG, SVG e GML para mapas WMS;
d) suporte ao padrão de estilos SLD do OGC;
e) suporte a WFS-T;
f) fácil integração com extensões e *plugins*.

Figura 10 – Interface de administração do GeoServer

Fonte: Do autor.

4.2.2 MapServer

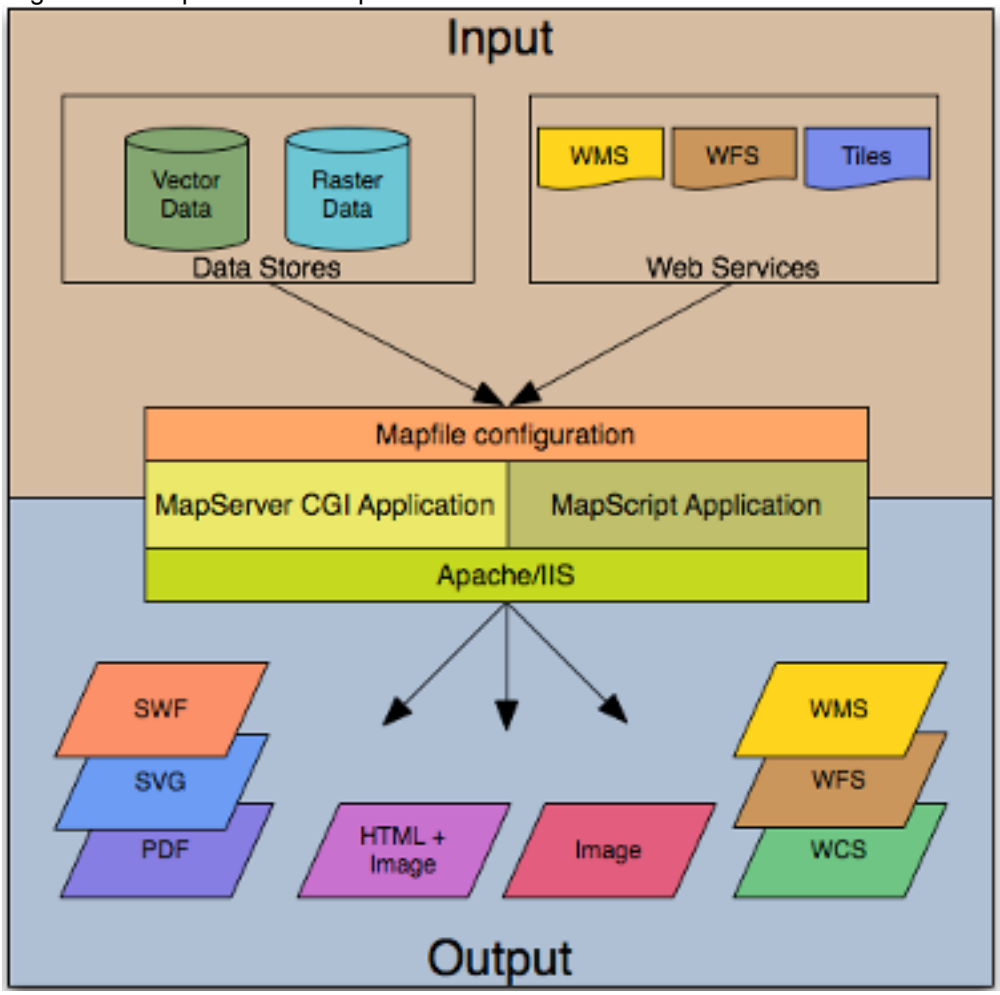
MapServer é um ambiente de desenvolvimento e servidor de mapas de código aberto para criação de aplicações geoespaciais em ambiente web. É flexível e confiável e possui capacidade de integração com diversas aplicações SIG. Inicialmente desenvolvido pela Universidade de Minnesota, atualmente é desenvolvido e mantido por uma comunidade de desenvolvedores ao redor do mundo (OSGEO, 2016).

MapServer é multiplataforma e pode ser rodado na maioria dos servidores web. Suas principais características são (OSGEO, 2016):

a) MapScript: um ambiente para *scripting* que suporta linguagens populares como PHP, Python, Perl, C# e Java;
b) suporte aos principais padrões OGC;
c) grande variedade de consultas espaciais;
d) possibilita integração com as principais tecnologias geoespaciais de código aberto, como GDAL/OGR, PostGIS e Proj.4.

A arquitetura do MapServer permite utilizar as mais variadas fontes de entrada de dados, incluindo arquivos em formatos vetoriais, bancos de dados geográficos, e *web services*. Já os formatos de saída incluem SVG, PDF, formatos de imagem como JPG e PNG, e padrões OGC como WMS e WFS (figura 11).

Figura 11 – Arquitetura do MapServer



Fonte: MapServer (2016).

4.3 CLIENTE DE DADOS GEOSPACIAIS

Os clientes de dados geoespaciais têm a função de disponibilizar os mapas de maneira gráfica na tela do usuário. Os dados são solicitados ao servidor através de padrões estabelecidos. Nesta seção serão expostos os principais clientes de dados geoespaciais existentes atualmente no mercado.

4.3.1 OpenLayers

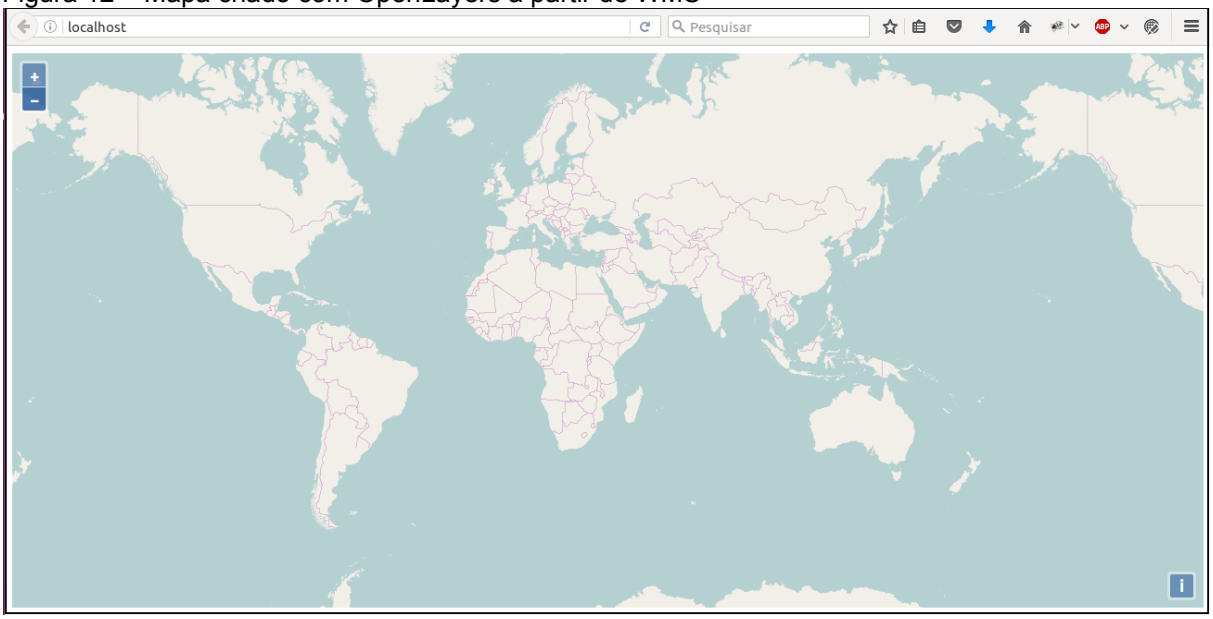
OpenLayers é uma biblioteca JavaScript de código aberto para criação de mapas interativos na web. Foi inicialmente desenvolvida pela empresa MetaCarta

em 2006 como uma alternativa *open source* ao Google Maps. Atualmente é mantida por uma comunidade de desenvolvedores e encontra-se na versão 3.18.

OpenLayers possibilita a construção de aplicações geoespaciais na web através de uma *Application Programming Interface* (API) JavaScript, que permite a customização de vários aspectos do mapa, como estilos, camadas, eventos, entre outros. Possui também alto grau de compatibilidade com padrões OGC, como WMS e WFS (GRATIER; SPENCER; HAZZARD, 2015).

Na figura 12 é possível visualizar um mapa criado a partir do serviço público de WMS do OpenStreetMap.

Figura 12 – Mapa criado com OpenLayers a partir de WMS



Fonte: Do autor.

O código presente na figura 13 foi utilizado neste exemplo será explicado em seguida.

Figura 13 – Código de exemplo OpenLayers

```
1  <!doctype html>
2  <html lang="en">
3      <head>
4          <link rel="stylesheet" href="http://openlayers.org/en/v3.18.2/css/ol.css" type="text/css">
5          <script src="http://openlayers.org/en/v3.18.2/build/ol.js" type="text/javascript"></script>
6          <title>OpenLayers 3 Example</title>
7      </head>
8      <body>
9          <div id="map" style="height: 600px; width: 100%"></div>
10         <script type="text/javascript">
11             var map = new ol.Map({
12                 target: 'map',
13                 layers: [
14                     new ol.layer.Tile({
15                         source: new ol.source.OSM()
16                     })
17                 ],
18                 view: new ol.View({
19                     center: ol.proj.fromLonLat([37.41, 8.82]),
20                     zoom: 4
21                 })
22             });
23         </script>
24     </body>
25 </html>
```

Fonte: Do autor.

Dentro da *tag head* são carregados o arquivo de estilos css (linha 4) e a biblioteca OpenLayers (linha 5). Em seguida, dentro do corpo da página, é criada uma *tag div* (linha 9), na qual o mapa estará contido. Na linha 11, uma variável de nome *map* recebe um mapa que é criado a partir do construtor *ol.Map.* Na linha 13 são especificadas as camadas que farão parte do mapa, e a fonte de dados é especificada na linha 15, que neste caso é um serviço WMS do projeto OpenStreetMap. Por fim, a posição do centro do mapa é definida na linha 19 e o nível de *zoom* na linha 20.

4.3.2 Leaflet

Leaflet é uma biblioteca JavaScript de código aberto para mapas interativos com suporte a dispositivos móveis. Possui uma extensa API que atende a grande parte das necessidades para criação de mapas web. Leaflet foi projetada para simplicidade, performance, e usabilidade. Funciona nas principais plataformas *desktop* e *mobile* e permite integração com diversos *plugins*, além de implementar padrões OGC (LEAFLET, 2016). Um exemplo de um mapa criado com Leaflet é dado na figura 14.

Figura 14 – Exemplo de mapa criado com Leaflet

Fonte: Do autor.

4.3.3 Google maps

Google Maps surgiu em 2005 e foi um dos primeiros serviços de web mapping abertos para o público geral. Foi desenvolvido utilizando JavaScript, XML e AJAX. Este serviço disponibiliza uma API JavaScript que permite aos desenvolvedores integrarem os serviços em suas páginas web. É livre para uso, porém possui limites de uso diário, a partir dos quais passa a ser cobrado (GOOGLE, 2017).

Através de sua API JavaScript, é possível integrar diversas fontes de dados, como serviços online de mapas, camadas vetoriais, imagens de satélite entre outros. Também é possível realizar geocodificação, que é o serviço de transformar coordenadas espaciais em endereços postais. Google Maps também integra aos seus serviços o Street View, que permite visualizar imagens reais de ruas em todo o planeta (GOOGLE, 2017).

A figura 15 mostra um mapa criado com a API do Google Maps.

Figura 15 – Exemplo de mapa criado com API do Google Maps

Fonte: Do autor.

4.4 BANCO DE DADOS

O padrão SFA, apresentado anteriormente, é importante para a construção de um banco de dados geográfico, pois é ele que possibilita a compatibilidade entre os diversos sistemas gerenciadores de banco de dados (SGBD) distribuídos por diferente fabricantes.

Atualmente existem vários bancos de dados com capacidade para armazenar e manipular dados geográficos. Entre os *softwares open source* destaca-se o PostGIS como um dos mais utilizados no mercado (RUZICKA, 2016). Entre os *softwares* proprietários destaca-se o Oracle Spatial.

4.4.1 PostGIS

PostGIS é uma extensão espacial para o Sistema de Gerenciamento de Banco de Dados (SGBD) PostgreSQL. PostGIS fornece tipos de dados e funções adicionais ao PostgreSQL, tornando-o habilitado para manipular e armazenar dados geográficos. É um projeto *open source* e totalmente compatível com os padrões

OGC, sendo atualmente o banco de dados geográfico mais utilizado por diversas instituições ao redor do mundo. São as principais características do PostGIS (POSTGIS, 2016):

a) processamento e análise de dados vetoriais e matriciais através da linguagem *Structured Query Language* (SQL);
b) álgebra para dados matriciais para processamento de imagens de alta resolução;
c) suporte a *Shapefiles* e demais formatos de dados vetoriais;
d) suporte a feições 3D;
e) suporte a topologia.

4.4.2 Oracle Spatial

Oracle Spatial é um extensão espacial para o SGBD Oracle. É compatível com o padrão SFA é possui compatibilidade com todos os principais fornecedores de *software* SIG. Oracle Spatial possui alguns componentes fundamentais que habilitam o banco de dados Oracle a manipular dados geográficos. São eles (OLIVEIRA, 2009):

a) *Data Model:* cria o tipo de dados SDO_GEOMETRY para representar atributos geoespaciais;
b) *Location-Enabling*: possibilita a criação de tabelas contendo a coluna do tipo SDO_GEOMETRY;
c) *Spatial query and analysis*: adicionais funções que permitem manipular os dados armazenados na coluna SDO_GEOMETRY;
d) *Advanced Spatial Engine*: incorpora funções avançadas para construção de sistemas de informação geográfica;
e) *Visualization*: exibe os dados geoespaciais através de um servidor de mapas.

5 TECNOLOGIAS UTILIZADAS NO DESENVOLVIMENTO DO FRAMEWORK

Neste capítulo serão apresentadas as principais tecnologias utilizadas no desenvolvimento do trabalho proposto.

5.1 JAVA

Java é uma linguagem de programação de alto nível, orientada a objetos, e projetada para ser multi-plataforma. Foi desenvolvida nos anos 90 na empresa Sun Microsystems. É reconhecida por possuir alta performance, segurança, portabilidade e robustez (ORACLE, 2016).

Na linguagem Java, o código é primeiro escrito em um simples arquivo de texto que possui a extensão *.java*. Estes arquivos são então compilados para arquivos com extensão *.class* pelo compilador. Porém, o compilador Java não gera código de máquina, mas sim bytecodes, que é o código interpretado pela *Java Virtual Machine* (JVM).

A JVM está disponível para vários sistemas operacionais, por isso, o mesmo programa Java pode ser executado no Microsoft Windows, Solaris OS, Linux, ou Mac OS.

Por fazer uso extenso de conceitos de programação orientada a objetos, serão aqui definidos os conceitos mais importantes, como classe, objeto, e herança.

5.1.1 Classe

Uma classe é um modelo que define como um objeto será criado um objeto. A classe especifica portanto quais os atributos e os métodos estarão disponíveis a um objeto. Desta maneira, pode-se dizer que um objeto é uma instância de uma classe. Uma classe pode ser criada na linguagem Java com a palavra-chave *class* (MARQUES, 2012).

5.1.2 Objeto

Objetos são um conceito chave para o entendimento da programação orientada a objetos. Todos objetos compartilham duas características: estado e

comportamento. Um objeto armazena o seu estado em atributos (também conhecidos como variáveis) e expõe seu comportamento através de métodos (também conhecidos como funções). Métodos manipulam o estado interno de um objeto e são importantes para a comunicação entre dois objetos. O ato de esconder o estado interno de um objeto e somente permitir a sua manipulação através de seus próprios métodos é chamado de encapsulamento (ORACLE, 2016).

5.1.3 Herança

Herança é um conceito que possibilita a reutilização de código através do compartilhamento do que há de comum entre duas classes.

Sendo assim, uma classe (chamada de subclasse) herda os atributos e os métodos de outra classe existente (chamada de superclasse). Este conceito proporciona economia de tempo e maior eficiência no desenvolvimento de programas. Na linguagem Java a herança é feita através da palavra-chave *extends* (MARQUES, 2012).

5.2 JAVASERVER FACES

JavaServer Faces é uma especificação e um *framework* de componentes *server-side* para construção de aplicações web com a linguagem Java. A tecnologia JSF consiste de (ORACLE, 2016):

a) uma API para representar componentes e gerenciar os seus estados, manipular eventos, validação *server-side*, conversão de dados, definir estrutura de navegação de páginas, suporte a internacionalização e acessibilidade;
b) bibliotecas de *tags* para adicionar componentes às páginas e conectar estes componentes a objetos no lado do servidor.

Por fazer parte das tecnologias *Java Enterprise Edition* (JEE), o JSF possui alto grau de integração com ambientes de desenvolvimento e desta forma facilita o desenvolvimento de aplicações web ao proporcionar recursos como *drag-and-drop* e autocomplemento de código. Além disso, fornece componentes pré-fabricados e permite criação de componentes customizados, que agilizam o desenvolvimento de páginas web (GEARY; HORSTMANN, 2012).

A declaração de componentes JSF é feita utilizando tags no formato *Extensible Markup Language* (XML), sendo assim de maneira muito similar à declaração de tags *Hypertext Markup Language* (HTML). A figura 16 apresenta um exemplo básico de uma página JSF com um campo para entrada de dados e um botão para enviar os dados.

Figura 16 – Exemplo de código JSF

```
1 <html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/html">
2     <body>
3         <h:form>
4             Nome: <h:inputText value="#{controlador.nome}" />
5             Senha: <h:inputText value="#{controlador.senha}" />
6             <h:commandButton value="Enviar Dados" action="#{controlador.enviaDados}" />
7         </h:form>
8     </body>
9 </html>
```

Fonte: Do autor.

Na linha 1 é declarada uma biblioteca de *tags* que possuirá o prefixo *h.* Na linha 2 é declarado o corpo da página, seguido da declaração de um formulário na linha 3. Nas linhas 4 e 5 são declarados dois componentes *inputText,* que são os campos que receberão o nome e a senha. Em seguida, na linha 6, é declarado um *commandButton* que servirá para enviar os dados contidos nos campos. Nas linhas 7, 8, e 9 as *tags* são fechadas, constituindo assim um formato válido XML. O exemplo produzirá a página apresentada na figura 17.

Figura 17 – Exemplo de página JSF

Fonte: Do autor.

No lado do servidor, a classe *Controlador,* apresentada na figura 18, é responsável por controlar a página e receber os dados submetidos.

Figura 18 – Classe Controlador

```
 6  @ManagedBean
 7  @ViewScoped
 8  public class Controlador {
 9      private String nome;
10      private String senha;
11
12      public String getNome() {
13          return nome;
14      }
15      public void setNome(String nome) {
16          this.nome = nome;
17      }
18      public String getSenha() {
19          return senha;
20      }
21      public void setSenha(String senha) {
22          this.senha = senha;
23      }
24
25      public void enviaDados(){
26
27      }
28  }
```

Fonte: Do autor.

Ao clicar no botão *Enviar Dados,* os dados serão enviados diretamente para as variáveis *nome* e *senha*, mostradas nas linhas 9 e 10.

5.3 JAVASCRIPT

JavaScript é uma linguagem de programação interpretada de alto nível. É considerada a linguagem de programação da web. Atualmente todos os navegadores de internet possuem suporte nativo a esta linguagem, tornando-a um dos pilares do desenvolvimento web junto com HTML e CSS. Possui uma API mínima para manipulação de texto, *arrays*, datas e expressões regulares, porém não inclui funcionalidades para entrada e saída de dados (FLANAGAN, 2011, tradução nossa).

JavaScript foi projetada com o objetivo de introduzir conteúdo dinâmico a páginas web, que inicialmente eram estáticas. Portanto, através de APIs fornecidas pelos navegadores, é possível manipular o conteúdo de páginas web com esta linguagem.

6 TRABALHOS CORRELATOS

Diversas ferramentas para desenvolvimento de SIG em ambiente web têm surgido recentemente devido à relevância deste tema de pesquisa. Neste capítulo serão apresentados trabalhos que possuem conteúdo semelhante e que serviram de base para esta pesquisa.

6.1 UM SISTEMA DE INFORMAÇÃO GEOGRÁFICA EM PLATAFORMA WEB BASEADO NO GEOSERVER

Este artigo de Sousa et al. foi publicado em 2013 no *World Congress on Systems Engineering and Information Technology*. O artigo teve como objetivo demonstrar um SIG web que tem como base o GeoServer.

O sistema desenvolvido, denominado EMAPI, é um sistema web, de código aberto, que possui as principais funcionalidades encontradas em SIGs desktop. Também foram utilizados OpenLayers e JSF para o desenvolvimento da camada de apresentação. O *software* permite gerar mapas automaticamente a partir de serviços web ou de arquivos dos formatos mais populares. Também possui suporte a diversos banco de dados, como PostGIS, OracleSpatial, DB2, entre outros (SOUSA et al., 2013).

6.2 CRIAÇÃO DE UM FRAMEWORK OPEN SOURCE PARA CONSTRUÇÃO DE SISTEMAS GEOESPACIAIS

Esta monografia foi apresentada à Universidade Tiradentes por Robert Anderson Nogueira de Oliveira, como requisito para obtenção do grau de bacharel em Sistemas de Informação. Este trabalho teve como objetivo desenvolver um *framework open source* para criação de SIG utilizando a plataforma JEE.

Para o desenvolvimento do *framework* OL4JSF foram utilizados as tecnologias JSF e OpenLayers 2 para a camada de visualização, e o *software* Maven para auxiliar no processo de compilação e gerenciamento de dependências. Foram criados componentes JSF para renderização de mapas em ambiente web utilizando como base a biblioteca OpenLayers 2. Foram criadas ainda classes

utilitárias para manipulação de objetos geográficos, com o intuito de facilitar a transmissão de dados entre o servidor e o cliente (OLIVEIRA, 2009).

6.3 DESIGN AND IMPLEMENTATION OF WIRELESS SENSOR NETWORK MANAGEMENT SYSTEM BASED ON WEBGIS

Neste artigo de Haibo e Fang, publicado em 2013 no *Journal of Theoretical and Applied Information Technology*, foi desenvolvido um SIG web para gerenciamento de uma Rede de Sensores Sem Fio.

As tecnologias utilizadas no desenvolvimento foram JSF e OpenLayers para a interface de usuário, e PostgreSQL para armazenamento de dados.

O sistema desenvolvido permite monitorar e analisar os dados coletados pelos sensores, exibição de dados em tempo real, gerenciamento da rede de maneira gráfica, e envio de comandos para os sensores de maneira remota (HAIBO; FANG, 2013).

6.4 ESTUDO DE TECNOLOGIAS PARA DESENVOLVIMENTO DE SISTEMAS DE INFORMAÇÃO GEOGRÁFICA EM AMBIENTE WEB

Este Trabalho de Conclusão de Curso foi apresentado à Universidade Tecnológica Federal do Paraná por Jean Paulo da Silva, como requisito parcial para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

O trabalho teve por objetivo expor de forma detalhada as principais tecnologias utilizadas para o desenvolvimento de SIG em ambiente web. Foram estudados os bancos de dados geográficos PostGIS e Oracle Spatial, os servidores de mapas Geoserver a Mapserver, e foi feito um estudo de caso utilizando o *framework* OL4JSF. O estudo de caso consistiu no desenvolvimento de um editor de geometrias que tem como camada base o Google Maps. O editor permite relacionar dados espaciais e descritivos, que ficam disponíveis em ambiente web (SILVA, 2011).

7 PROTÓTIPO DE UM FRAMEWORK DE COMPONENTES JAVASERVER FACES PARA CRIAÇÃO DE SISTEMAS DE INFORMAÇÕES GEOGRÁFICAS UTILIZANDO OPENLAYERS 3

Tendo como base o conhecimento adquirido durante a pesquisa, foi desenvolvido o protótipo de um *framework* de componentes JavaServer Faces, utilizando como base a biblioteca OpenLayers 3, com o objetivo de criar uma ferramenta que facilite o desenvolvimento de sistemas de informações geográficas utilizando a plataforma JEE.

7.1 METODOLOGIA

Para a realização deste projeto de pesquisa foram empregadas as seguintes etapas metodológicas: levantamento bibliográfico; definição dos componentes JSF a serem criados; desenvolvimento do protótipo do *framework*.

Na etapa de levantamento bibliográfico foi obtido o conteúdo teórico necessário para o desenvolvimento do protótipo. Foram estudados os conceitos de cartografia necessários para o bom entendimento das tecnologias geoespaciais utilizadas. Em seguida foi estudado o processo de desenvolvimento de um SIG, incluindo as tecnologias necessárias. O próximo passo foi estudar o processo de criação de componentes JSF.

Na etapa de definição dos componentes a serem criados foram levantados quais os componentes mais utilizados em sistemas geoespaciais. Foram definidos os seguintes componentes: mapa, camadas, e controles. Além disso, foi decidido criar uma API para maior eficiência na manipulação de dados.

Por fim, foi iniciado o desenvolvimento do protótipo. Os testes foram realizados durante e após o desenvolvimento para garantir o funcionamento correto dos componentes.

7.1.1 Estrutura do Framework

Para o desenvolvimento deste trabalho foram pesquisadas as principais funcionalidades necessárias em um SIG. Também foi criada uma API para facilitar a integração entre *front-end* e *back-end*.

As funcionalidades pesquisadas foram representadas através dos seguintes componentes: mapa, camadas, *popup*, e controles.

O componente mapa trata-se do *container* dentro do qual serão inseridos os demais componentes. Para o seu correto funcionamento é obrigatório preencher alguns atributos, como a coordenada de centro do mapa e a projeção cartográfica.

Em seguida, as camadas devem ser inseridas no mapa. Os componentes de camadas desenvolvidos foram camada WMS, camada OpenStreet Map, e camada de entrada de dados vetoriais.

Por fim, foram desenvolvidas funcionalidades utilitárias como a *popup*, que trata-se de uma pequena janela flutuante que surge assim que o usuário clica no mapa, além de outros demais controles.

7.1.2 Desenvolvimento

Nesta seção será apresentado o processo de desenvolvimento de cada um dos componentes.

7.1.2.1 Map

Map é o principal objeto da biblioteca OpenLayers, pois ele serve de *container* para todos os demais objetos. Neste *framework* o objeto *Map* é renderizado através da classe *Map*, que é mostrada na figura 19.

Figura 19 – Componente Map

```
12 @FacesComponent(value = "org.ol3jsf.components.Map",
13                 createTag = true,
14                 namespace = "http://ol3jsf.org/core",
15                 tagName = "map")
16 public class Map extends UIInput {
17
18     @Override
19     public void encodeBegin(FacesContext context) throws IOException {
20         ResponseWriter writer = context.getResponseWriter();
21
22         String mapVar = getJsVariable();
23         String id = this.getClientId();
24         String center = getCenterLonLat();
25         String zoom = getZoom();
26         String transformationSource = getTransformationSource();
27         String transformationTarget = getTransformationTarget();
28         Properties.setTransformationSource(transformationSource);
29         Properties.setTransformationTarget(transformationTarget);
30
31         writer.startElement("div", this);
32         writer.writeAttribute("id", id, "id");
33         writer.writeAttribute("class", "map", null);
34         writer.endElement("div");
35
36         writer.startElement("script", this);
37         writer.writeAttribute("type", "text/javascript", null);
38
39         writer.write("var " + mapVar + " = new ol.Map({"
40                             + "target: '" + id + "',"
41                             + "view: new ol.View({"
42                                 + "center: ol.proj.fromLonLat([" + center + "]),"
43                                 + "zoom: " + zoom
44                             + "})"
45                         + "});\n");
```

Fonte: Do autor.

Na linha 12, é possível observar a declaração do componente para reconhecimento pelo JSF. A opção *value* define a classe que será utilizada para renderização. A opção *namespace* define o *namespace* que deverá ser incluído na página JSF. Por fim, a opção *tagName* define o nome da *tag*.

O método *encodeBegin*, na linha 19, é responsável por iniciar a codificação do componente. Os atributos da *tag* são recebidos e armazenados em variáveis, como é exibido nas linhas 22 a 29. Em seguida é iniciado a declaração da div que servirá de *container* para o mapa.

Na linha 36, é iniciado o código javascript que renderizará de fato o objeto *Map* da biblioteca OpenLayers. Finalmente, na linha 39, o objeto é declarado, sendo passado para ele as opções advindas dos atributos da *tag*.

7.1.2.2 OSMLayer

OSMLayer utiliza o serviço gratuito *OpenStreetMap*, que é um projeto colaborativo para criar um mapa do mundo. Esta camada possui principalmente o

desenho das ruas de um determinado local, mas possui também vegetações e pontos importantes como aeroportos e pontos turísticos. Geralmente é utilizada como camada base. A figura 20 mostra a classe que renderiza este componente.

Figura 20 – Componente OSMLayer

```
11 @FacesComponent(value = "org.ol3jsf.components.OSMLayer",
12                 createTag = true,
13                 namespace = "http://ol3jsf.org/core",
14                 tagName = "osmLayer")
15 public class OSMLayer extends UIComponentBase {
16
17     @Override
18     public void encodeAll(FacesContext context) throws IOException {
19         ResponseWriter writer = context.getResponseWriter();
20         UIComponent parent = this;
21
22         while (!(parent instanceof Map)) {
23             parent = parent.getParent();
24         }
25
26         Map mapComponent = (Map) parent;
27         String mapVar = mapComponent.getJsVariable();
28
29         String isBaseLayer = getIsBaseLayer();
30         String newLayer = "var newOsmLayer = new ol.layer.Tile({  source: new ol.source.OSM()});\n";
```

Fonte: Do autor.

Na linha 11, o componente é declarado para reconhecimento pelo JSF, como já explicado anteriormente. Na linha 22 é criado um *loop* para identificar qual a *tag* mãe da *tag* atual. Na linha 30 é declarado o código Javascript que gerará a camada.

7.1.2.3 WMSLayer

WMSLayer é um tipo de camada que recebe dados de um *Web Map Service.* Geralmente os dados retornados são imagens em formato JPEG ou PNG. A classe *WMSLayer*, exibida na figura 21, renderiza este componente.

Figura 21 – Componente WMSLayer

```
11 @FacesComponent(value = "org.ol3jsf.components.WMSLayer",
12                 createTag = true,
13                 namespace = "http://ol3jsf.org/core",
14                 tagName = "wmsLayer")
15 public class WMSLayer extends UIComponentBase {
16
17     @Override
18     public void encodeAll(FacesContext context) throws IOException {
19         ResponseWriter writer = context.getResponseWriter();
20         UIComponent parent = this;
21         .
           .
           .
33         String newLayer = "var newLayer = new ol.layer.Image({"
34                 + "          title: '" + name + "', "
35                 + "          source: new ol.source.ImageWMS({url: '" + url + "',  "
36                 + "            params: {'LAYERS': '" + layer
37                 + "                  ', 'VERSION': '1.1.1'},"
38                 + "            serverType: 'geoserver' "
39                 + "          }) "
40                 + "        });\n";
41
42         writer.write(newLayer + mapVar + ".addLayer(newLayer);\n");
43     }
```

Fonte: Do autor.

O código presente nas linhas 22 a 31 é similar em todas as classes e a partir de agora será omitido. Na linha 33, é declarado o código Javascript que gera a camada. Um objeto *ol.layer.Image* é criado, que receberá opções como o nome da camada, a URL do servidor onde os dados estão disponíveis, o tipo de servidor, entre outros. Finalmente, a camada é adicionada ao mapa na linha 36.

7.1.2.4 Input Vector Layer

Input Vector Layer é uma camada utilizada para entrada e visualização de dados vetoriais. Ao receber um objeto geográfico, esta camada desenha-o na tela, permitindo assim destacá-lo dos demais. A classe que renderiza o componente é exibida na figura 22.

Figura 22 – Componente Input Vector Layer

```
13 @FacesComponent(value = "org.ol3jsf.components.InputVectorLayer",
14                 createTag = true,
15                 namespace = "http://ol3jsf.org/core",
16                 tagName = "inputVectorLayer")
17 public class InputVectorLayer extends UIInput {
18
19     @Override
20     public void encodeAll(FacesContext context) throws IOException {
21         ResponseWriter writer = context.getResponseWriter();
22         String features = ComponentUtils.getValueToRender(context, this);
23         UIComponent parent = this;
24         .
           .
           .
36         writer.write(
37                 "var vector = new ol.layer.Vector({"
38                 + "  source: new ol.source.Vector(), "
39                 + "  style: new ol.style.Style({"
40                 + "    fill: new ol.style.Fill({"
41                 + "      color: 'rgba(50,30,230, 0.3)'"
42                 + "    }),"
43                 + "    stroke: new ol.style.Stroke({"
44                 + "      color: 'rgba(50,25,180, 1)',"
45                 + "      width: 2"
46                 + "    })"
47                 + "  })"
48                 + "});\n");
49
50         writer.write(mapVar + ".addLayer(vector);\n");
51         writer.write("vector.getSource().addFeatures(" + features + ");\n");
52     }
```

Fonte: Do autor.

Na linha 22, é recebido o valor que se encontra dentro do atributo *value* da *tag*. O valor consiste de objetos geográficos em formato WKT. Na linha 36, uma camada de vetores é criada. Os objetos geográficos são então repassados à camada de vetores na linha 39, fazendo com que eles sejam desenhados na tela.

7.1.2.5 Popup

O componente *Popup* trata-se de uma pequena janela que surge quando o usuário efetua um clique único no mapa. Geralmente é utilizado para apresentar informações em forma de texto sobre a geometria que recebeu o clique. A classe *Popup* é mostrada na figura 23.

Figura 23 – Componente Popup

```
13 @FacesComponent(value = "org.ol3jsf.components.Popup",
14                 createTag = true,
15                 namespace = "http://ol3jsf.org/core",
16                 tagName = "popup")
17 public class Popup extends UIInput {
18
19     @Override
20     public void encodeAll(FacesContext context) throws IOException {
21         ResponseWriter writer = context.getResponseWriter();
22         UIComponent parent = this;
23         .
           .
           .
49         writer.write("var overlay = new ol.Overlay(({"
50                 + "     element: popupContainer, "
51                 + "     autoPan: true, "
52                 + "     autoPanAnimation: {duration: 250}"
53                 + "   }));\n");
54
55         writer.write("popupCloser.onclick = function() {"
56                 + "     overlay.setPosition(undefined); "
57                 + "     popupCloser.blur(); "
58                 + "     return false; "
59                 + "   };\n");
60
61         writer.write(mapVar + ".addOverlay(overlay);\n");
62         writer.write(mapVar + ".on('singleclick', function(evt) { "
63                 + "               var coordinate = evt.coordinate; "
64                 + "               overlay.setPosition(coordinate); "
65                 + "            });\n");
66
67         writer.write("popupContainer.appendChild(popupCloser);\n");
68         writer.write("popupContainer.appendChild(popupContent);\n");
69
70     }
```

Fonte: Do autor.

O código Javascript que renderiza o componente é iniciado na linha 49. O objeto *ol.Overlay* é o recurso da biblioteca OpenLayers utilizado para exibir conteúdo por cima do mapa, que neste caso será a *popup*. Na linha 55 um evento é criado para que seja possível fechar a *popup* ao clicar em um botão. Da linha 61 em diante o *overlay* é adicionado ao mapa e um evento *singleclick* é criado, permitindo assim que a janela apareça na mesma coordenada onde o usuário efetuou o clique.

7.1.2.6 OverviewMap

OverviewMap é um pequeno mapa que fica localizado no canto da tela e fornece uma visão mais ampla da área onde o usuário está navegando. A Classe *OverviewMap* é exibida na figura 24.

Figura 24 – Componente OverviewMap

```
11 @FacesComponent(value = "org.ol3jsf.components.OverviewMap",
12                  createTag = true,
13                  namespace = "http://ol3jsf.org/core",
14                  tagName = "overviewMap")
15 public class OverviewMap extends UIComponentBase {
16
17     @Override
18     public void encodeAll(FacesContext context) throws IOException {
19         ResponseWriter writer = context.getResponseWriter();
20         UIComponent parent = this;
21         .
           .
           .
29         writer.write(mapVar + ".addControl(new ol.control.OverviewMap());\n");
30     }
```

Fonte: Do autor.

O controle *OverviewMap* da biblioteca OpenLayers é criado na linha 29 e é adicionado ao mapa na mesma linha.

7.1.2.7 Rotate

Rotate é um controle que permite rotacionar o mapa ao segurar a tecla *shift* e arrastar o mapa. A classe *Rotate* é exibida na figura 25.

Figura 25 – Componente Rotate

```
11 @FacesComponent(value = "org.ol3jsf.components.Rotate",
12                  createTag = true,
13                  namespace = "http://ol3jsf.org/core",
14                  tagName = "rotate")
15 public class Rotate extends UIComponentBase {
16
17     @Override
18     public void encodeAll(FacesContext context) throws IOException {
19         ResponseWriter writer = context.getResponseWriter();
20         UIComponent parent = this;
           .
           .
           .
29         writer.write(mapVar + ".addInteraction(new ol.interaction.DragRotateAndZoom());\n");
30     }
```

Fonte: Do autor.

O controle *DragRotateAndZoom* da biblioteca OpenLayers é criado na linha 29 e é adicionado ao mapa na mesma linha.

7.1.2.8 ZoomSlider

ZoomSlider é um controle visual que permite dar zoom no mapa. A classe *ZoomSlider* é exibida na figura 26.

Figura 26 – Componente ZoomSlider

```
11 @FacesComponent(value = "org.ol3jsf.components.ZoomSlider",
12                  createTag = true,
13                  namespace = "http://ol3jsf.org/core",
14                  tagName = "zoomSlider")
15 public class ZoomSlider extends UIComponentBase {
16
17     @Override
18     public void encodeAll(FacesContext context) throws IOException {
19         ResponseWriter writer = context.getResponseWriter();
20         UIComponent parent = this;
           .
           .
           .
29         writer.write(mapVar + ".addControl(new ol.control.ZoomSlider());\n");
30     }
```

Fonte: Do autor.

O controle *ZoomSlider* da biblioteca OpenLayers é criado na linha 29 e é adicionado ao mapa na mesma linha.

7.1.2.9 Classes da API

As classes *Feature* e *WKTFeaturesCollection* foram desenvolvidas para auxiliar o usuário na manipulação de dados geográficos advindos do banco de dados. Portanto o usuário pode recuperar uma geometria em formato WKT do banco de dados e criar uma nova *Feature*, que estará pronta para utilização pela biblioteca OpenLayers. O usuário poderá também criar uma coleção de *features* através da classe *WKTFeaturesCollection*, facilitando assim a manipulação de grandes quantidades de feições geométricas. Na figura 27 é mostrado um exemplo de uso destas classes.

Figura 27 – Classes da API

```
32     public void pesquisar() {
33         Feature f1 = new Feature("POLYGON((-49.323065 -28.525524, "
34                                          + "-49.322884 -28.525541, "
35                                          + "-49.322931 -28.525897, "
36                                          + "-49.323108 -28.525876))");
37
38         Feature f2 = new Feature("POLYGON((-49.322065 -28.525524, "
39                                          + "-49.321884 -28.525541, "
40                                          + "-49.320931 -28.525897, "
41                                          + "-49.323108 -28.525876))");
42
43         WKTFeaturesCollection features = new WKTFeaturesCollection();
44         features.addFeature(f1);
45         features.addFeature(f2);
46         setFeatures(features.toMap());
47     }
```

Fonte: Do autor.

São criados dois polígonos em formato WKT que são passados como parâmetro para o construtor da classe *Feature*, resultando assim em uma nova *feature* compatível com a biblioteca OpenLayers. Em seguida é criada uma *WKTFeaturesCollection* que serve para armazenar as *features*. Finalmente, na linha 46 é executado o método toMap() que organizará as feições de maneira que a biblioteca OpenLayers possa reconhecê-las e transformá-las em polígonos na tela do usuário.

7.1.3 Testes

Concluída a etapa de desenvolvimento, foram efetuados testes para verificar o funcionamento dos componentes. A figura 28 mostra as *tags* criadas na etapa de desenvolvimento, que serão testadas em seguida.

Figura 28 – Tags desenvolvidas

Tag	Função
<m:map />	Cria o mapa, que servirá de container para os outros componentes
<m:osmLayer />	Camada OpenStreet Map
<m:wmsLayer />	Camada WMS
<m:inputVectorLayer />	Camada de entrada de dados vetoriais
<m:popup />	Popup para exibição de textos, etc.
<m:overviewMap />	Componente que mostra a visão geral do mapa
<m:rotate />	Funcionalidade para rotação do mapa
<m:zoomSlider />	Componente para ajuste de zoom

Fonte: Do autor.

Para testar o *framework* foi criado um projeto web com a IDE Eclipse. Após a criação do projeto, o arquivo JAR resultante do presente trabalho, denominado ol3jsf.jar, foi adicionado ao projeto. Foram adicionados também a implementação de referência do JSF, Mojarra 2.2, e a biblioteca Primefaces 5.3.

Em seguida foi criada uma página web denominada teste.xhtml. Nesta página foram adicionados os *namespaces* e *scripts* necessários para o correto funcionamento dos componentes. Em seguida foram adicionados os componentes, como é mostrado na figura 29.

Figura 29 – Página teste

teste.xhtml

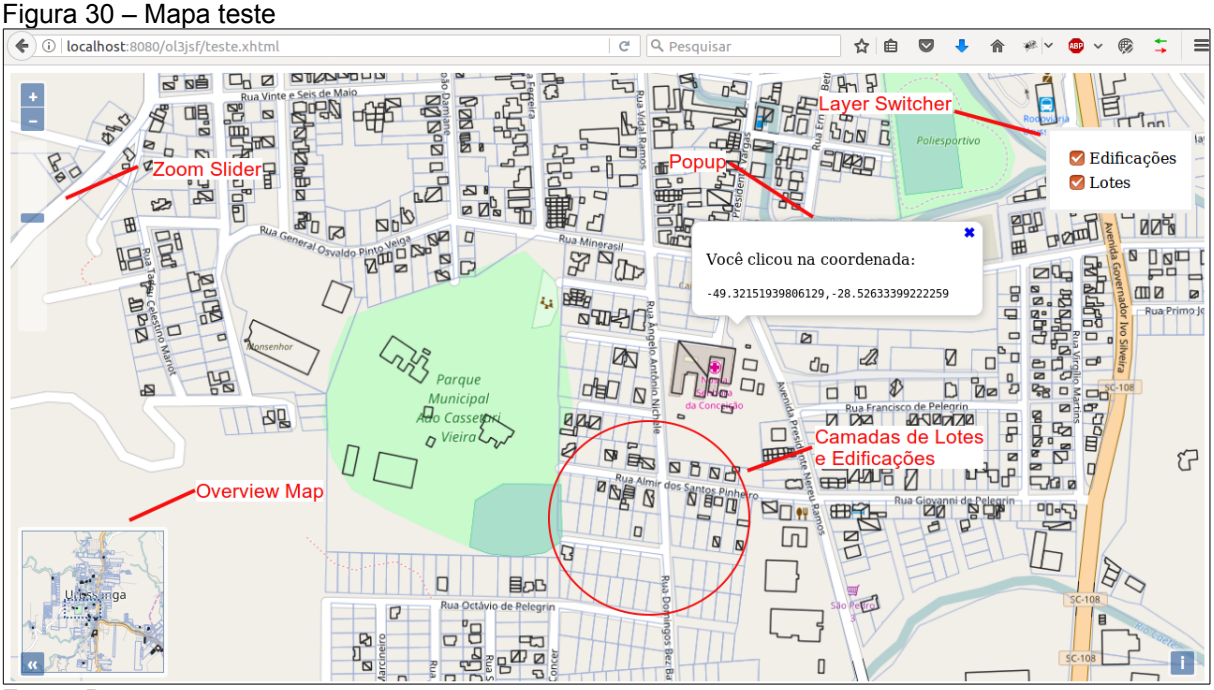
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:m="http://ol3jsf.org/core"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:p="http://primefaces.org/ui">
    <h:head>
        <title>OL3JSF</title>
        <script src="#{request.contextPath}/ol3/ol.js" type="text/javascript"></script>
        <script src="#{request.contextPath}/ol3/ol3-layerswitcher.js" type="text/javascript"></script>
        <link rel="stylesheet" href="#{request.contextPath}/ol3/ol.css" type="text/css" />
        <link rel="stylesheet" href="#{request.contextPath}/css/style.css" type="text/css" />
        <link rel="stylesheet" href="#{request.contextPath}/ol3/ol3-layerswitcher.css" type="text/css" />
    </h:head>

    <h:body>
        <m:map id="map" jsVariable="map" centerLonLat="#{testeController.center}"
        transformationSource="EPSG:4326" transformationTarget="EPSG:3857" zoom="#{testeController.zoom}">
            <m:osmLayer isBaseLayer="true" />
            <m:wmsLayer name="Lotes"
                        url="http://localhost:8181/geoserver/wms"
                        layer="urussanga:lotes"/>
            <m:wmsLayer name="Edificações"
                        url="http://localhost:8181/geoserver/wms"
                        layer="urussanga:edificacoes"/>
            <m:inputVectorLayer value="#{testeController.features}" />
            <m:popup id="popup" styleClass="ol-popup"/>
            <m:overviewMap />
            <m:rotate />
            <m:zoomSlider />
        </m:map>
    </h:body>
</html>
```

Fonte: Do autor.

É possível observar nas linha 3 a 7 a declaração dos *namespaces* necessários, destacando-se na linha 5 o *namespace* do presente *framework*. Em seguida os *scripts* e folhas de estilo da biblioteca OpenLayers são declarados nas linha 10 a 14. Na linha 18 é declarado o componente <m:map>, que servirá de *container* para o mapa. Em seguida vem a camada base, que utiliza o serviço *OpenStreetMap* para fornecer um mapa de fundo. Logo após vem as camadas WMS, que neste exemplo representam os lotes e edificações da cidade de Urussanga. Na linha 27 é declarada a camada de entrada de dados vetoriais. Finalmente, são declarados os componentes utilitários, como *popup*, *overviewMap*, e *zoomSlider*.

Este código irá gerar a página mostrada na figura 30.

Figura 30 – Mapa teste



Fonte: Do autor.

7.2 RESULTADOS OBTIDOS

A pesquisa inicial permitiu determinar quais as principais funcionalidades necessárias em um SIG web. Deu-se início então ao estudo da biblioteca OpenLayers 3 e do processo de criação de componentes JSF, visando criar um framework que facilitasse o máximo possível o desenvolvimento de SIGs web com estas tecnologias.

O desenvolvimento do protótipo do *framework* foi completado de acordo com os objetivos estabelecidos no início do projeto. Os componentes desenvolvidos permitem criar mapas utilizando tags JSF, minimizando assim o uso de código Javascript. A partir dos atributos das *tags* é possível receber valores diretamente no componente, minimizando também a necessidade de funções auxiliares, e possibilitando, desta forma, um alto nível de integração entre *front-end* e *back-end*.

Foi constatado que a implementação utilizando componentes ocupou 14 linhas de código, enquanto a implementação JavaScript ocupou 52 linhas de código. Observou-se, portanto, uma redução de pelo menos 70% do número de linhas de código ao utilizar o presente *framework,* o que contribui também para melhorar a manutenibilidade do projeto.

O protótipo foi testado utilizando dados da base cartográfica do município de Urussanga, que disponibiliza estes dados de maneira pública. Durante a fase de testes foi verificado que o protótipo tem comportamento estável e não apresenta *bugs*. Deve-se ressaltar, porém, que ainda cabem melhorias ao código.

O código-fonte e exemplos de uso do protótipo estão disponíveis em https://github.com/elielwaltrick/ol3jsf.

Sendo assim, todos os objetivos propostos foram atingidos.

8 CONCLUSÃO

A demanda por sistemas computacionais que possibilitam analisar e gerenciar a superfície terrestre vem crescendo com o passar do tempo. Tais sistemas, denominados Sistemas de Informações Geográficas, têm se tornado ferramenta de grande importância, tanto para profissionais da área quanto para cidadãos.

Contudo, percebe-se que a criação destes sistemas envolve um grau relevante de dificuldade, pois requer conhecimentos de múltiplas áreas e o uso de diversas tecnologias. Portanto, foi proposto neste trabalho a criação de um protótipo de um *framework* de componentes JSF que facilitasse a criação de mapas na web utilizando os recursos da biblioteca OpenLayers 3.

A partir do estudo bibliográfico foi possível estudar e compreender os conceitos e tecnologias necessárias para o desenvolvimento do trabalho, destacando-se como tema central do trabalho os componentes JSF e a biblioteca OpenLayers.

Durante a pesquisa foram encontradas algumas dificuldades, principalmente em relação a documentação das tecnologias envolvidas. Porém estas questões foram sendo resolvidas ao longo do desenvolvimento com o auxílio de comunidades *online* cujo conteúdo é direcionado a tais tecnologias. Outra dificuldade encontrada foi na criação do componente *Popup*. Durante a etapa de desenvolvimento teve-se a ideia de criar um atributo *value* para permitir que o texto da *popup* pudesse ser definido no *Managed Bean*. No entanto, observou-se que esta implementação seria complexa, e, devido ao tempo limitado, optamos por não criar este atributo.

Conclui-se, portanto, que foi possível atingir o objetivo geral e os objetivos específicos, resultando em um protótipo de um *framework* que permite, a partir de componentes JSF, o desenvolvimento de SIGs web de maneira simples, utilizando poucas linhas de código, e seguindo os padrões JSF.

Durante o desenvolvimento do trabalho foram observadas algumas possibilidades para trabalhos futuros, como a continuação do desenvolvimento do framework, adicionando novos componentes e funcionalidades, bem como atualização do código à medida que novas versões da biblioteca OpenLayers surgem. Outra sugestão seria fazer um comparativo entre as diversas bibliotecas

para *web mapping* e analisar questões como desempenho, funcionalidades, acessibilidade, entre outras questões.

REFERÊNCIAS

BALLATORE, Andrea et al. **A comparison of open source geospatial technologies for web mapping.** International Journal of Web Engineering and Technology, 2011. Disponível em: <http://dx.doi.org/10.1504/IJWET.2011.043440> Acesso em: 17 set. 2016.

BONHAM-CARTER, Graeme F. **Geographic information systems for geoscientists: modelling with GIS.** Tarrytown: Elsevier, 2014.

CAMARA, Gilberto. 2005. **Bancos de Dados Geográficos.** Disponível em: <http://www.dpi.inpe.br/gilberto/livro/bdados> Acesso em: 14 set. 2016.

CHHAJED, Tejashree. **Deploying contrail forecasting service to reduce the impact of aviation on Environment.** 2016**.** 96f. Dissertação (Mestrado) – Curso de Engenharia de Software, TU CHEMNITZ, Chemnitz, 2016.

ESRI. Environmental Systems Research Institute, Inc. **ESRI Shapefile technical description.** 1998. Disponível em: <https://www.esri.com/library/whitepapers/pdfs/shapefile.pdf> Acesso em: 10 out. 2016.

FITZ, Paulo Roberto. **Cartografia básica.** São Paulo: Oficina de Textos, 2008. 143 p.

FLANAGAN, David. **JavaScript: The definitive guide: Activate your web pages.** Sebastopol: O'Reilly Media, Inc., 2011.

GEARY, David; HORSTMANN, Cay S. **Core JavaServer Faces.** Rio de Janeiro: Alta Books, 2012. 636 p.

GEOSERVER. **GeoServer User Manual.** 2016. Disponível em: <http://docs.geoserver.org/stable/en/user/.> Acesso em: 25 set. 2016.

GOODCHILD, Michael et al. **Interoperating geographic information systems.** Santa Barbara: Springer Science & Business Media, 2012.

GOOGLE. **Google Maps**. 2017. Disponível em: <https://developers.google.com/maps/> Acesso em: 25 jun. 2017.

GRATIER, Thomas; SPENCER, Paul; HAZZARD, Erik. **OpenLayers 3: Beginner's Guide.** Packt Publishing Ltd, 2015.

GRONNING, Torgeir M. **Data structure, Access and Presentation in Web-GIS for marine research.** 2013. 126f. Dissertação (Mestrado) – Curso de Ciência da Computação, University of Bergen, Bergen, 2013. Disponível em: <http://bora.uib.no/handle/1956/6784> Acesso em: 14 out. 2016.

KIRCH, Wilhelm. **Encyclopedia of Public Health.** 2 Vol. Springer Science & Business Media, 2008.

KONECNY, Gottfried. **Geoinformation: remote sensing, photogrammetry and geographic information systems.** Boca Raton: CRC Press, 2014.

LEAFLET. **Leaflet Documentation.** 2016. Disponível em <http://www.leafletjs.com> Acesso em 18 nov. 2016.

LONGLEY, Paul. **Geographical infomation systems and science.** 2nd ed Chichester: Wiley, 2005. 517 p.

LV, Dekui et al. **A WebGIS platform design and implementation based on Open Source GIS middleware.** Geoinformatics 24th International Conference, 2016.

MAPSERVER. **MapServer Documentation.** 2016. Disponível em <http://www.mapserver.org> Acesso em 19 nov. 2016.

MARQUES, Juliano. **Comparação de desempenho e consumo de memória entre frameworks de mapeamento objeto-relacional java Hibernate e Eclipselink.** 2012. 79f. TCC (Graduação) – Curso de Ciência da Computação, UNESC, Criciúma, 2012.

OLIVEIRA, Robert A. N. **Criação de um framework open source para construção de sistemas geoespaciais.** 2009. 82f. TCC (Graduação) – Curso de Sistemas de Informação, Universidade Tiradentes, Aracaju, 2009.

OGC, Inc. **OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 1: Common architecture.** 2011. Disponível em: <http://www.opengeospatial.org/standards/sfa> Acesso em: 28 set. 2016.

OGC, Inc. **About OGC**. 2016. Disponível em: <http://www.opengeospatial.org> Acesso em: 25 set. 2016.

ORACLE. **The Java Tutorials.** 2016. Disponível em <http://docs.oracle.com/javase/tutorial/java/index.html> Acesso em: 28 set. 2016.

OSGEO. **Open Source Geospatial Foundation.** 2016. Disponível em <http://www.osgeo.org/mapserver> Acesso em 19 nov. 2016.

POON, Joanne. **Spatial information generation from high-resolution satellite imagery.** 2007. 160f. Dissertação (Doutorado) – Curso de Geomática, The University of Melbourne, Melbourne, 2007. Disponível em: <http://www.crcsi.com.au/assets/Resources/5374d5c8-c248-4b12-a04b-b008923f489a.pdf> Acesso em: 10 out. 2016.

POSTGIS. **PostGIS Documentation.** 2016. Disponível em <http://postgis.net> Acesso em 17 nov. 2016.

QUOOS, João H. **O que é latitude e longitude?** 2016. Disponível em: <http://coral.ufsm.br/cartografia/> Acesso em: 19 nov. 2016.

RUZICKA, Jan. **Comparing speed of Web Map Service with GeoServer on ESRI Shapefile and PostGIS.** Geoinformatics FCE CTU, 2016. Disponível em: <https://ojs.cvut.cz/ojs/index.php/gi/article/download/gi.15.1.1/3634> Acesso em: 21 mai. 2017

SILVA, Jean P. **Estudo de tecnologias para desenvolvimento de sistemas de informação geográfica em ambiente web.** 2011. 79f. Monografia – Curso de Tecnologia em Análise e Desenvolvimento de Sistemas, UTFPR, Medianeira, 2011.

WESTRA, Erik. **Python Geospatial Analysis Essentials.** Packt Publishing Ltd, 2015. 200p.

APÊNDICE(S)

APÊNDICE A – ARTIGO CIENTÍFICO

Protótipo de um Framework de Componentes JavaServer Faces para Criação de Sistemas de Informações Geográficas Utilizando OpenLayers 3

Eliel Frasson Waltrick[1], Fabrício Giordani[1]

[1]Curso de Ciência da Computação
Universidade do Extremo Sul Catarinense (UNESC) – Criciúma, SC – Brazil

{elielwaltrick,fgiordani}@gmail.com

Abstract. *There has been an increasing demand for Geographic Information Systems in recent years, since they have become an important tool in many areas. However, a certain degree of difficulty is observed in the development of these systems. Therefore, it was proprosed the creation of a framework that enables the development of web GIS utilizing JSF components. First, a survey was conducted to determine the main features needed in a GIS. Subsequently, the components were built, wherein the OpenLayers library and the JSF specification were used. Tests were conducted to ensure the proper functioning of the prototype, which showed satisfactory results. In conclusion, the objectives were achieved, resulting in JSF components which require up to 70% less lines of code compared to a JavaScript implementation.*

Resumo. *Nos dias atuais existe uma grande demanda por Sistemas de Informações Geográficas. No entanto, observa-se que a criação destes sistemas envolve um grau relevante de dificuldade. Portanto, foi proposta a criação de um framework que permita criar SIGs web utilizando componentes JSF. Foi feito um levantamento das principais funcionalidades necessárias em um SIG. Em seguida foram criados os componentes, utilizando recursos JSF e OpenLayers. Foram realizados testes para garantir o funcionamento do protótipo, que demonstraram resultados satisfatórios. Concluiu-se que foi possível atingir o objetivo proposto, tendo como resultado os componentes JSF que reduzem em até 70% a quantidade de linhas de código comparado com uma implementação JavaScript.*

1. Introdução

O crescimento do poder de processamento dos computadores, aliado aos progressos nas tecnologias de monitores gráficos, permitiram, há algumas décadas atrás, o desenvolvimento dos primeiros Sistemas de Informações Geográficas (SIG). Desde então, os SIG têm se demonstrado cada vez mais importantes em diversas áreas do conhecimento humano, como gerenciamento de recursos territoriais, planejamento urbano, transportes, marketing, entre outros (BONHAM-CARTER, 2014, tradução nossa).

A integração das tecnologias da Web 2.0 com os SIG resultaram no advento do *web mapping*, que se tornou presente em grande parte dos serviços online. Fontes abertas de dados geográficos, como o projeto colaborativo OpenStreetMap, em conjunto com demais projetos *open source*, contribuíram para o desenvolvimento de inúmeras novas tecnologias para

criação de mapas em ambiente web, resultando assim em sistemas cada vez mais modernos e complexos (BALLATORE et al, 2011, tradução nossa).

A biblioteca OpenLayers é uma das mais importantes tecnologias *client-side* para *web mapping* por ser uma das mais completas e estáveis (LV, 2016).

Entretanto, o desenvolvimento de aplicações geoespaciais geralmente torna-se complexo para os desenvolvedores devido a grande quantidade de tecnologias envolvidas. Desta maneira, ao criar uma aplicação utilizando JavaServer Faces (JSF) e OpenLayers é necessário criar métodos de integração entre estas duas tecnologias, que podem vir a ser complexos e de difícil manutenção.

Levando em consideração a complexidade da construção de um SIG web, o objetivo deste trabalho é criar um *framework* de componentes JSF para criação de aplicações geoespaciais com a biblioteca OpenLayers 3, com o intuito de facilitar e agilizar o desenvolvimento de projetos que utilizam estas tecnologias.

2. Metodologia

Para a realização deste projeto de pesquisa foram empregadas as seguintes etapas metodológicas: levantamento bibliográfico; definição dos componentes JSF a serem criados; desenvolvimento do protótipo do *framework*.

Na etapa de levantamento bibliográfico foi obtido o conteúdo teórico necessário para o desenvolvimento do protótipo. Foram estudados os conceitos de cartografia necessários para o bom entendimento das tecnologias geoespaciais utilizadas. Em seguida foi estudado o processo de desenvolvimento de um SIG, incluindo as tecnologias necessárias. O próximo passo foi estudar o processo de criação de componentes JSF.

Na etapa de definição dos componentes a serem criados foram levantados quais os componentes mais utilizados em sistemas geoespaciais. Foram definidos os seguintes componentes: mapa, camadas, e controles. Além disso, foi decidido criar uma API para maior eficiência na manipulação de dados.

Por fim, foi iniciado o desenvolvimento do protótipo. Os testes foram realizados durante e após o desenvolvimento para garantir o funcionamento correto dos componentes.

2.1 Estrutura do framework

Para o desenvolvimento deste trabalho foram pesquisadas as principais funcionalidades necessárias em um SIG. Também foi criada uma API para facilitar a integração entre *front-end* e *back-end*.

As funcionalidades pesquisadas foram representadas através dos seguintes componentes: mapa, camadas, *popup*, e controles.

O componente mapa trata-se do *container* dentro do qual serão inseridos os demais componentes. Para o seu correto funcionamento é obrigatório preencher alguns atributos, como a coordenada de centro do mapa e a projeção cartográfica.

Em seguida, as camadas devem ser inseridas no mapa. Os componentes de camadas desenvolvidos foram camada WMS, camada OpenStreet Map, e camada de entrada de dados vetoriais.

Por fim, foram desenvolvidas funcionalidades utilitárias como a *popup*, que trata-se de uma pequena janela flutuante que surge assim que o usuário clica no mapa, além de outros demais controles.

2.2 Desenvolvimento dos componentes

Nesta seção será apresentado o processo de desenvolvimento de cada um dos componentes.

2.2.1 Map

Map é o principal objeto da biblioteca OpenLayers, pois ele serve de *container* para todos os demais objetos. Neste *framework* o objeto *Map* é renderizado através da classe *Map*, que é mostrada na figura 1.

```
12 @FacesComponent(value = "org.ol3jsf.components.Map",
13                  createTag = true,
14                  namespace = "http://ol3jsf.org/core",
15                  tagName = "map")
16 public class Map extends UIInput {
17
18     @Override
19     public void encodeBegin(FacesContext context) throws IOException {
20         ResponseWriter writer = context.getResponseWriter();
21
22         String mapVar = getJsVariable();
23         String id = this.getClientId();
24         String center = getCenterLonLat();
25         String zoom = getZoom();
26         String transformationSource = getTransformationSource();
27         String transformationTarget = getTransformationTarget();
28         Properties.setTransformationSource(transformationSource);
29         Properties.setTransformationTarget(transformationTarget);
30
31         writer.startElement("div", this);
32         writer.writeAttribute("id", id, "id");
33         writer.writeAttribute("class", "map", null);
34         writer.endElement("div");
35
36         writer.startElement("script", this);
37         writer.writeAttribute("type", "text/javascript", null);
38
39         writer.write("var " + mapVar + " = new ol.Map({"
40                             + "target: '" + id + "',"
41                             + "view: new ol.View({"
42                                 + "center: ol.proj.fromLonLat([" + center + "]),"
43                                 + "zoom: " + zoom
44                             + "})"
45                         + "});\n");
```

Figura 1. Componente Map

Na linha 12, é possível observar a declaração do componente para reconhecimento pelo JSF. A opção *value* define a classe que será utilizada para renderização. A opção *namespace* define o *namespace* que deverá ser incluído na página JSF. Por fim, a opção *tagName* define o nome da *tag*.

O método *encodeBegin*, na linha 19, é responsável por iniciar a codificação do componente. Os atributos da *tag* são recebidos e armazenados em variáveis, como é exibido nas linhas 22 a 29. Em seguida é iniciado a declaração da div que servirá de *container* para o mapa.

Na linha 36, é iniciado o código javascript que renderizará de fato o objeto *Map* da biblioteca OpenLayers. Finalmente, na linha 39, o objeto é declarado, sendo passado para ele as opções advindas dos atributos da *tag*.

2.2.2 OSMLayer

OSMLayer utiliza o serviço gratuito *OpenStreetMap*, que é um projeto colaborativo para criar um mapa do mundo. Esta camada possui principalmente o desenho das ruas de um determinado local, mas possui também vegetações e pontos importantes como aeroportos e pontos turísticos. Geralmente é utilizada como camada base. A figura 2 mostra a classe que renderiza este componente.

```
11 @FacesComponent(value = "org.ol3jsf.components.OSMLayer",
12                 createTag = true,
13                 namespace = "http://ol3jsf.org/core",
14                 tagName = "osmLayer")
15 public class OSMLayer extends UIComponentBase {
16
17     @Override
18     public void encodeAll(FacesContext context) throws IOException {
19         ResponseWriter writer = context.getResponseWriter();
20         UIComponent parent = this;
21
22         while (!(parent instanceof Map)) {
23             parent = parent.getParent();
24         }
25
26         Map mapComponent = (Map) parent;
27         String mapVar = mapComponent.getJsVariable();
28
29         String isBaseLayer = getIsBaseLayer();
30         String newLayer = "var newOsmLayer = new ol.layer.Tile({  source: new ol.source.OSM()});\n";
```

Figura 2. Componente OSMLayer

Na linha 11, o componente é declarado para reconhecimento pelo JSF, como já explicado anteriormente. Na linha 22 é criado um *loop* para identificar qual a *tag* mãe da *tag* atual. Na linha 30 é declarado o código Javascript que gerará a camada.

2.2.3 WMSLayer

WMSLayer é um tipo de camada que recebe dados de um *Web Map Service*. Geralmente os dados retornados são imagens em formato JPEG ou PNG. A classe *WMSLayer*, exibida na figura 3, renderiza este componente.

```
11 @FacesComponent(value = "org.ol3jsf.components.WMSLayer",
12                 createTag = true,
13                 namespace = "http://ol3jsf.org/core",
14                 tagName = "wmsLayer")
15 public class WMSLayer extends UIComponentBase {
16
17     @Override
18     public void encodeAll(FacesContext context) throws IOException {
19         ResponseWriter writer = context.getResponseWriter();
20         UIComponent parent = this;
21         .
           .
           .
33         String newLayer = "var newLayer = new ol.layer.Image({"
34                 + "           title: '" + name + "', "
35                 + "           source: new ol.source.ImageWMS({url: '" + url + "',  "
36                 + "             params: {'LAYERS': '" + layer
37                 + "                   ', 'VERSION': '1.1.1'},"
38                 + "             serverType: 'geoserver' "
39                 + "           }) "
40                 + "         });\n";
41
42         writer.write(newLayer + mapVar + ".addLayer(newLayer);\n");
43     }
```

Figura 3. WMSLayer

O código presente nas linhas 22 a 31 é similar em todas as classes e a partir de agora será omitido. Na linha 33, é declarado o código Javascript que gera a camada. Um objeto *ol.layer.Image* é criado, que receberá opções como o nome da camada, a URL do servidor onde os dados estão disponíveis, o tipo de servidor, entre outros. Finalmente, a camada é adicionada ao mapa na linha 36.

2.2.4 InputVectorLayer

Input Vector Layer é uma camada utilizada para entrada e visualização de dados vetoriais. Ao receber um objeto geográfico, esta camada desenha-o na tela, permitindo assim destacá-lo dos demais. A classe que renderiza o componente é exibida na figura 4.

```
13 @FacesComponent(value = "org.ol3jsf.components.InputVectorLayer",
14                 createTag = true,
15                 namespace = "http://ol3jsf.org/core",
16                 tagName = "inputVectorLayer")
17 public class InputVectorLayer extends UIInput {
18
19     @Override
20     public void encodeAll(FacesContext context) throws IOException {
21         ResponseWriter writer = context.getResponseWriter();
22         String features = ComponentUtils.getValueToRender(context, this);
23         UIComponent parent = this;
24         .
           .
           .
36         writer.write(
37                 "var vector = new ol.layer.Vector({"
38                 + "  source: new ol.source.Vector(), "
39                 + "  style: new ol.style.Style({"
40                 + "    fill: new ol.style.Fill({"
41                 + "      color: 'rgba(50,30,230, 0.3)'"
42                 + "    }),"
43                 + "    stroke: new ol.style.Stroke({"
44                 + "      color: 'rgba(50,25,180, 1)',"
45                 + "      width: 2"
46                 + "    })"
47                 + "  })"
48                 + "});\n");
49
50         writer.write(mapVar + ".addLayer(vector);\n");
51         writer.write("vector.getSource().addFeatures(" + features + ");\n");
52     }
```

Figura 4. Componente InputVectorLayer

Na linha 22, é recebido o valor que se encontra dentro do atributo *value* da *tag*. O valor consiste de objetos geográficos em formato WKT. Na linha 36, uma camada de vetores é criada. Os objetos geográficos são então repassados à camada de vetores na linha 39, fazendo com que eles sejam desenhados na tela.

2.2.5 Popup

O componente *Popup* trata-se de uma pequena janela que surge quando o usuário efetua um clique único no mapa. Geralmente é utilizado para apresentar informações em forma de texto sobre a geometria que recebeu o clique. A classe *Popup* é mostrada na figura 5.

```
13 @FacesComponent(value = "org.ol3jsf.components.Popup",
14                 createTag = true,
15                 namespace = "http://ol3jsf.org/core",
16                 tagName = "popup")
17 public class Popup extends UIInput {
18
19     @Override
20     public void encodeAll(FacesContext context) throws IOException {
21         ResponseWriter writer = context.getResponseWriter();
22         UIComponent parent = this;
23         .
           .
           .
49         writer.write("var overlay = new ol.Overlay(({"
50                 + "     element: popupContainer, "
51                 + "     autoPan: true, "
52                 + "     autoPanAnimation: {duration: 250}"
53                 + "   }));\n");
54
55         writer.write("popupCloser.onclick = function() {"
56                 + "     overlay.setPosition(undefined); "
57                 + "     popupCloser.blur(); "
58                 + "     return false; "
59                 + "   };\n");
60
61         writer.write(mapVar + ".addOverlay(overlay);\n");
62         writer.write(mapVar + ".on('singleclick', function(evt) { "
63                 + "               var coordinate = evt.coordinate; "
64                 + "               overlay.setPosition(coordinate); "
65                 + "            });\n");
66
67         writer.write("popupContainer.appendChild(popupCloser);\n");
68         writer.write("popupContainer.appendChild(popupContent);\n");
69
70     }
```

Figura 5. Componente Popup

O código Javascript que renderiza o componente é iniciado na linha 49. O objeto *ol.Overlay* é o recurso da biblioteca OpenLayers utilizado para exibir conteúdo por cima do mapa, que neste caso será a *popup*. Na linha 55 um evento é criado para que seja possível fechar a *popup* ao clicar em um botão. Da linha 61 em diante o *overlay* é adicionado ao mapa e um evento *singleclick* é criado, permitindo assim que a janela apareça na mesma coordenada onde o usuário efetuou o clique.

2.2.6 OverviewMap

OverviewMap é um pequeno mapa que fica localizado no canto da tela e fornece uma visão mais ampla da área onde o usuário está navegando. A Classe *OverviewMap* é exibida na figura 6.

```
11 @FacesComponent(value = "org.ol3jsf.components.OverviewMap",
12                 createTag = true,
13                 namespace = "http://ol3jsf.org/core",
14                 tagName = "overviewMap")
15 public class OverviewMap extends UIComponentBase {
16
17     @Override
18     public void encodeAll(FacesContext context) throws IOException {
19         ResponseWriter writer = context.getResponseWriter();
20         UIComponent parent = this;
21         .
           .
           .
29         writer.write(mapVar + ".addControl(new ol.control.OverviewMap());\n");
30     }
```

Figura 6. Componente OverviewMap

O controle *OverviewMap* da biblioteca OpenLayers é criado na linha 29 e é adicionado ao mapa na mesma linha.

2.2.7 Rotate

Rotate é um controle que permite rotacionar o mapa ao segurar a tecla *shift* e arrastar o mapa. A classe *Rotate* é exibida na figura 7.

```
11 @FacesComponent(value = "org.ol3jsf.components.Rotate",
12                 createTag = true,
13                 namespace = "http://ol3jsf.org/core",
14                 tagName = "rotate")
15 public class Rotate extends UIComponentBase {
16
17     @Override
18     public void encodeAll(FacesContext context) throws IOException {
19         ResponseWriter writer = context.getResponseWriter();
20         UIComponent parent = this;
           .
           .
           .
29         writer.write(mapVar + ".addInteraction(new ol.interaction.DragRotateAndZoom());\n");
30     }
```

Figura 7. Componente Rotate

O controle *DragRotateAndZoom* da biblioteca OpenLayers é criado na linha 29 e é adicionado ao mapa na mesma linha.

2.2.8 ZoomSlider

ZoomSlider é um controle visual que permite dar zoom no mapa. A classe *ZoomSlider* é exibida na figura 8.

```
11 @FacesComponent(value = "org.ol3jsf.components.ZoomSlider",
12                 createTag = true,
13                 namespace = "http://ol3jsf.org/core",
14                 tagName = "zoomSlider")
15 public class ZoomSlider extends UIComponentBase {
16
17     @Override
18     public void encodeAll(FacesContext context) throws IOException {
19         ResponseWriter writer = context.getResponseWriter();
20         UIComponent parent = this;
           .
           .
           .
29         writer.write(mapVar + ".addControl(new ol.control.ZoomSlider());\n");
30     }
```

Figura 8. Componente ZoomSlider

O controle *ZoomSlider* da biblioteca OpenLayers é criado na linha 29 e é adicionado ao mapa na mesma linha.

3. Testes

Concluída a etapa de desenvolvimento, foram efetuados testes para verificar o funcionamento dos componentes. Foi criada uma página web denominada teste.xhtml. Nesta página foram adicionados os *namespaces* e *scripts* necessários para o correto funcionamento dos componentes. Em seguida foram adicionados os componentes, como é mostrado na figura 9.

teste.xhtml

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3  <html xmlns="http://www.w3.org/1999/xhtml"
4      xmlns:h="http://java.sun.com/jsf/html"
5      xmlns:m="http://ol3jsf.org/core"
6      xmlns:f="http://java.sun.com/jsf/core"
7      xmlns:p="http://primefaces.org/ui">
8      <h:head>
9          <title>OL3JSF</title>
10         <script src="#{request.contextPath}/ol3/ol.js" type="text/javascript"></script>
11         <script src="#{request.contextPath}/ol3/ol3-layerswitcher.js" type="text/javascript"></script>
12         <link rel="stylesheet" href="#{request.contextPath}/ol3/ol.css" type="text/css" />
13         <link rel="stylesheet" href="#{request.contextPath}/css/style.css" type="text/css" />
14         <link rel="stylesheet" href="#{request.contextPath}/ol3/ol3-layerswitcher.css" type="text/css" />
15     </h:head>
16
17     <h:body>
18         <m:map id="map" jsVariable="map" centerLonLat="#{testeController.center}"
19         transformationSource="EPSG:4326" transformationTarget="EPSG:3857" zoom="#{testeController.zoom}">
20             <m:osmLayer isBaseLayer="true" />
21             <m:wmsLayer name="Lotes"
22                         url="http://localhost:8181/geoserver/wms"
23                         layer="urussanga:lotes"/>
24             <m:wmsLayer name="Edificações"
25                         url="http://localhost:8181/geoserver/wms"
26                         layer="urussanga:edificacoes"/>
27             <m:inputVectorLayer value="#{testeController.features}" />
28             <m:popup id="popup" styleClass="ol-popup"/>
29             <m:overviewMap />
30             <m:rotate />
31             <m:zoomSlider />
32         </m:map>
33     </h:body>
34 </html>
```

Figura 9. Página teste

É possível observar nas linha 3 a 7 a declaração dos *namespaces* necessários, destacando-se na linha 5 o *namespace* do presente *framework*. Em seguida os *scripts* e folhas de estilo da biblioteca OpenLayers são declarados nas linha 10 a 14. Na linha 18 é declarado o componente <m:map>, que servirá de *container* para o mapa. Em seguida vem a camada base, que utiliza o serviço *OpenStreetMap* para fornecer um mapa de fundo. Logo após vem as camadas WMS, que neste exemplo representam os lotes e edificações da cidade de Urussanga. Na linha 27 é declarada a camada de entrada de dados vetoriais. Finalmente, são declarados os componentes utilitários, como *popup*, *overviewMap*, e *zoomSlider*.

Este código irá gerar a página mostrada na figura 10.

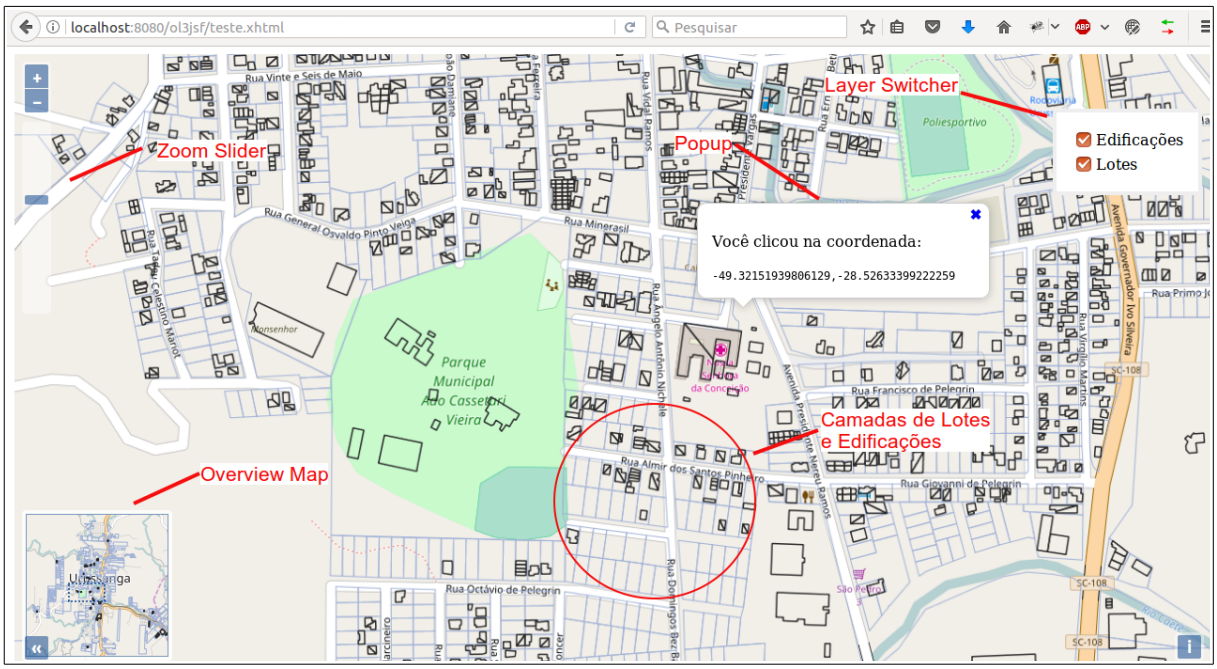


Figura 10. Mapa teste

4. Resultados

A pesquisa inicial permitiu determinar quais as principais funcionalidades necessárias em um SIG web. Deu-se início então ao estudo da biblioteca OpenLayers 3 e do processo de criação de componentes JSF, visando criar um framework que facilitasse o máximo possível o desenvolvimento de SIGs web com estas tecnologias.

O desenvolvimento do protótipo do *framework* foi completado de acordo com os objetivos estabelecidos no início do projeto. Os componentes desenvolvidos permitem criar mapas utilizando tags JSF, minimizando assim o uso de código Javascript. A partir dos atributos das *tags* é possível receber valores diretamente no componente, minimizando também a necessidade de funções auxiliares, e possibilitando, desta forma, um alto nível de integração entre *front-end* e *back-end*.

Foi constatado que a implementação utilizando componentes ocupou 14 linhas de código, enquanto a implementação JavaScript ocupou 52 linhas de código. Observou-se, portanto, uma redução de pelo menos 70% do número de linhas de código ao utilizar o presente *framework,* o que contribui também para melhorar a manutenibilidade do projeto.

O protótipo foi testado utilizando dados da base cartográfica do município de Urussanga, que disponibiliza estes dados de maneira pública. Durante a fase de testes foi verificado que o protótipo tem comportamento estável e não apresenta *bugs*. Deve-se ressaltar, porém, que ainda cabem melhorias ao código.

O código-fonte e exemplos de uso do protótipo estão disponíveis em https://github.com/elielwaltrick/ol3jsf.

Sendo assim, todos os objetivos propostos foram atingidos.

5. Considerações finais

A demanda por sistemas computacionais que possibilitam analisar e gerenciar a superfície terrestre vem crescendo com o passar do tempo. Tais sistemas, denominados Sistemas de Informações Geográficas, têm se tornado ferramenta de grande importância, tanto para profissionais da área quanto para cidadãos.

Contudo, percebe-se que a criação destes sistemas envolve um grau relevante de dificuldade, pois requer conhecimentos de múltiplas áreas e o uso de diversas tecnologias. Portanto, foi proposto neste trabalho a criação de um protótipo de um *framework* de componentes JSF que facilitasse a criação de mapas na web utilizando os recursos da biblioteca OpenLayers 3.

A partir do estudo bibliográfico foi possível estudar e compreender os conceitos e tecnologias necessárias para o desenvolvimento do trabalho, destacando-se como tema central do trabalho os componentes JSF e a biblioteca OpenLayers.

Durante a pesquisa foram encontradas algumas dificuldades, principalmente em relação a documentação das tecnologias envolvidas. Porém estas questões foram sendo resolvidas ao longo do desenvolvimento com o auxílio de comunidades *online* cujo conteúdo é direcionado a tais tecnologias. Outra dificuldade encontrada foi na criação do componente *Popup*. Durante a etapa de desenvolvimento teve-se a ideia de criar um atributo *value* para permitir que o texto da *popup* pudesse ser definido no *Managed Bean*. No entanto, observou-se que esta implementação seria complexa, e, devido ao tempo limitado, optamos por não criar este atributo.

Conclui-se, portanto, que foi possível atingir o objetivo geral e os objetivos específicos, resultando em um protótipo de um *framework* que permite, a partir de componentes JSF, o desenvolvimento de SIGs web de maneira simples, utilizando poucas linhas de código, e seguindo os padrões JSF.

Durante o desenvolvimento do trabalho foram observadas algumas possibilidades para trabalhos futuros, como a continuação do desenvolvimento do framework, adicionando novos componentes e funcionalidades, bem como atualização do código à medida que novas versões da biblioteca OpenLayers surgem. Outra sugestão seria fazer um comparativo entre as diversas bibliotecas para *web mapping* e analisar questões como desempenho, funcionalidades, acessibilidade, entre outras questões.

6. Referências

BALLATORE, Andrea et al. A comparison of open source geospatial technologies for web mapping. International Journal of Web Engineering and Technology, 2011. Disponível em: <http://dx.doi.org/10.1504/IJWET.2011.043440> Acesso em: 17 set. 2016.

BONHAM-CARTER, Graeme F. Geographic information systems for geoscientists: modelling with GIS. Tarrytown: Elsevier, 2014.

LV, Dekui et al. A WebGIS platform design and implementation based on Open Source GIS middleware. Geoinformatics 24th International Conference, 2016.