

UNIVERSIDADE COMUNITÁRIA DA REGIÃO DE CHAPECÓ

UNOCHAPECO

Área de Ciências Exatas e Ambientais

Ciência da Computação

William Klassmann

**MVC NO *BACK-END* E NO *FRONT-END*:
CRIANDO API'S *RESTFull* COM *ZEND FRAMEWORK 2* E
ANGULARJS**

Chapecó – SC, jun. de 2015

WILLIAM KLASSMANN

**MVC NO *BACK-END* E NO *FRONT-END*:
CRIANDO API'S *RESTFull* COM *ZEND FRAMEWORK2* E ANGULARJS**

Monografia (Trabalho de Conclusão de Curso)
apresentada ao curso de Ciência da Computação da
Unochapecó, Chapecó, SC, como parte dos requisitos
para a obtenção do grau de Bacharel em Ciência da
Computação.

Orientador: Cezar Júnior de Souza

Chapecó – SC, jun. de 2015

FOLHA DE APROVAÇÃO
TRABALHO DE CONCLUSÃO DE CURSO

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do título de Bacharel em Ciência da Computação, no Curso de Graduação em Ciência da Computação da Universidade Comunitária da região de Chapecó - UNOCHAPECÓ – Campus Chapecó

Apresentação à Comissão Examinadora integrada pelos professores:

Prof. Cezar Júnior Souza
Orientador

Prof. Marcos Antônio Moretto
Membro da Banca

Prof. Rodrigo Alceu Benedet
Membro da Banca

LISTAS DE FIGURAS

Figura 1: estrutura de fluxo de dados.	23
Figura 2: funcionamento do MVC.	24
Figura 3: utilização de variáveis no PHP.	29
Figura 4: utilização do gettype() no PHP.	29
Figura 5: inteiros nas bases decimal, hexadecimal e octal.	30
Figura 6: tipos de pontos flutuantes.	30
Figura 7: exemplos de Strings com aspas simples.	31
Figura 8: exemplo de String com a sintaxe Heredoc.	32
Figura 9: exemplo script em PHP com booleano.	33
Figura 10: exemplo de declaração de array.	34
Figura 11: exemplo de declaração de array com chaves e sem chaves.	34
Figura 12: exemplo de declaração de array na forma direta.	35
Figura 13: remoção de elemento de um array e de um array.	35
Figura 14: exemplo de objeto no PHP.	36
Figura 15: exemplo de utilização de recurso no PHP.	36
Figura 16: exemplo de utilização do comando get_resource_type() PHP.	37
Figura 17: exemplo de utilização do tipo null.	37
Figura 18: formato de um arquivo XML.	39
Figura 19: relacionamento entre os elementos.	41
Figura 20: elementos da mensagem SOAP.	43
Figura 21: ligação entre cliente e servidor.	47
Figura 22: diferenças entre um framework e uma Biblioteca de Classes OO.	51
Figura 23: declaração de namespace.	58
Figura 24: utilização de lambda function.	58
Figura 25: código com utilização de DI.	59
Figura 26: código com utilização de DI (complementação).	60
Figura 27: ciclo de vida do TDD.	62
Figura 28: estrutura de um Módulo do ZF2.	63
Figura 29: exemplos de comentários linha única (variante 1).	69
Figura 30: exemplos de comentários linha única (variante 2).	69

Figura 31: exemplos de comentários múltiplas linhas.	69
Figura 32: exemplo de comentário para marcação HTML e de <i>JavaScript</i>	69
Figura 33: formas de declaração de variáveis.	70
Figura 34: exemplo de dados literais.....	70
Figura 35: exemplo de hexadecimal no CSS.	71
Figura 36: exemplo de declarações usando literal inteiro.	71
Figura 37: literais decimais no JavaScript.	72
Figura 38: booleanos no JavaScript.	72
Figura 39: utilização de booleano.	72
Figura 40: strings no JavaScript.	73
Figura 41: formatação de Strings utilizando caracteres especiais no JavaScript.	73
Figura 42: resultado da utilização do \n.	73
Figura 43: declaração de array JavaScript.....	74
Figura 44: pegar o valor em uma posição do array.	74
Figura 45: sintaxe de um array misto.	74
Figura 46: pegar o valor em um array misto.	75
Figura 47: sintaxe para criação de um objeto.....	75
Figura 48: sintaxe para percorrer um objeto.	76
Figura 49: diagrama de interação entre o AngularJS com os componentes da arquitetura. ...	83
Figura 50: fluxo de protocolo de concessão de autorização do OAuth tradicional.....	88

LISTAS DE TABELAS

Tabela 1: caracteres de escape disponíveis no PHP.	32
Tabela 2: caracteres especiais JavaScript.	73
Tabela 3: palavras-chave do JavaScript.	76
Tabela 4: cronograma das atividades.	92

LISTAS DE ABREVIATURAS

ACL	<i>Access Control Lists</i>
AJAX	<i>Asynchronous JavaScript and XML</i>
API	<i>Application Programming Interface</i>
CGI	<i>Common Gateway Interface</i>
CLA	<i>Contributor License Agreement</i>
CRUD	<i>Create, Retrieve, Update and Delete</i>
CSS	<i>Cascading Style Sheets</i>
DI	<i>Dependency injection</i>
ECMA	<i>European Computer Manufactures Association</i>
HTML	<i>HyperText Markup Language</i>
HTTP	<i>HyperText Transfer Protocol</i>
IDE	<i>Integrated Development Environment</i>
IEEE	<i>Institute of Electric and Electronic Engineers</i>
I/O	Entrada e Saída
JSON	<i>JavaScript Object Notation</i>
MVC	<i>Model-View-Controller</i>
MVVM	<i>Model-View-ViewModel</i>
OO	Orientação a Objetos
OSI	<i>Open Source Initiative</i>
PHP	<i>Hipertext Preprocessor</i>

PHP/FI	<i>Personal Home Pages/Forms Interpreter</i>
REST	<i>Representation State Transfer</i>
RPC	<i>Remote Procedure Call</i>
SPA	<i>Single Page App</i>
SOAP	<i>Simple Object Access Protocol</i>
STS	<i>Secure Token Service</i>
TDD	<i>Test Driven Development</i>
UDDI	<i>Universal Description Discovery and Integration</i>
UI	<i>Interfaces de usuário</i>
URI	<i>Uniform Resource Identifier</i>
W3C	<i>World Wide Web Consortium</i>
WSDL	<i>Webservice Description Language</i>
XML	<i>Extensible Markup Language</i>
XP	<i>Extreme Programming</i>
ZF2	<i>Zend Framework 2</i>

SUMÁRIO

1. INTRODUÇÃO.....	11
1.1 Tema	12
1.2 Delimitação do Problema.....	12
1.3 Questões de Pesquisa	13
2. OBJETIVOS.....	14
2.1 Objetivo geral	14
2.2 Objetivos específicos	14
3. JUSTIFICATIVA	15
4. REVISÃO BIBLIOGRÁFICA	17
4.1 <i>Design Patterns</i>	17
4.1.1 Origens dos <i>Design Patterns</i>	17
4.1.2 Conceito de <i>Design Patterns</i>	18
4.1.3 Por que e quando usar <i>Design Patterns</i>	18
4.1.4 Tipos de <i>Design Patterns</i>	19
4.1.5 <i>Design Patterns</i> versus <i>Frameworks</i>	20
4.1.6 Considerações Finais sobre <i>Design Patterns</i>	20
4.2 Padrão MVC	22
4.2.1 Camada <i>Model</i>	22
4.2.2 Camada <i>View</i>	22
4.2.3 Camada <i>Controller</i>	23
4.2.4 Vantagens e desvantagens na utilização do padrão MVC	24
4.2.5 Considerações Finais sobre MVC.....	25
4.3 Linguagem de Programação PHP	26
4.3.1 História do PHP	26
4.3.1.1 PHP versão 3.....	27
4.3.1.2 PHP versão 4.....	27
4.3.1.3 PHP versão 5.....	28
4.3.2 Tipos de dados no PHP.....	28
4.3.2.1 Inteiros no PHP	30
4.3.2.2 Ponto flutuante no PHP	30

4.3.2.3	<i>Strings</i> no PHP.....	31
4.3.2.4	Booleano no PHP	33
4.3.2.5	<i>Arrays</i>	33
4.3.2.6	Objetos	36
4.3.2.7	Recursos	36
4.3.2.8	Nulo.....	37
4.3.3	Considerações finais sobre o PHP	38
4.4	<i>Web Service</i>	39
4.4.1	Tecnologia SOAP.....	41
4.4.2	Tecnologia REST	44
4.4.2.1	Restrições do REST.....	46
4.4.3	Tecnologia RESTfull.....	48
4.4.4	Considerações finais sobre <i>Web Service</i>	48
4.5	<i>Frameworks</i>	50
4.5.1	Diferenças entre <i>Framework</i> e uma Biblioteca de Classes OO	51
4.5.2	Diferenças entre <i>Frameworks</i> e <i>Design Patterns</i>	51
4.5.3	Características Básicas de <i>Frameworks</i>	52
4.5.4	Vantagens em utilizar <i>Frameworks</i>	52
4.5.5	Desvantagens em utilizar <i>Frameworks</i>	53
4.5.6	<i>Frameworks</i> caseiros	53
4.5.7	Considerações finais sobre <i>Framework</i>	54
4.6	<i>Zend Framework 2</i>	55
4.6.1	Características do <i>Zend Framework</i>	56
4.6.2	Conceitos do PHP utilizados no <i>Zend Framework</i>	57
4.6.2.1	<i>Namespace</i>	57
4.6.2.2	<i>Lambda function</i>	58
4.6.2.3	<i>Dependency injection</i>	59
4.6.2.4	Testes unitários	60
4.6.2.5	Eventos	63
4.6.2.6	Módulos.....	63
4.6.2.7	<i>Table gateway</i>	64
4.6.3	REST com ZF2.....	65
4.6.4	Considerações finais sobre o ZF2	66

4.7 JavaScript.....	67
4.7.1 Características do JavaScript.....	68
4.7.2 Declarações de variáveis no JavaScript.....	70
4.7.3 Literais do JavaScript	70
4.7.4 Considerações finais sobre o JavaScript	76
4.8 AngularJS.....	78
4.8.1 As características do AngularJS	79
4.8.2 Ferramentas necessárias para utilizar o AngularJS	81
4.8.3 Esqueleto da aplicação do AngularJS.....	81
4.8.4 A Anatomia de um aplicativo AngularJS	82
4.8.5 MVW Angular.....	83
4.8.6 Estruturando o código com MVC.....	84
4.8.7 Directives do AngularJS	85
4.8.8 Considerações finais do AngularJS.....	87
4.9 OAuth	88
4.9.1 Considerações finais sobre o OAuth	89
4.10 Considerações Finais	90
5 PROCEDIMENTOS METODOLÓGICOS.....	91
6 CRONOGRAMA	92
7 ORÇAMENTO.....	94
8 REFERÊNCIAS.....	95

1. INTRODUÇÃO

Buscando contextualizar este projeto, as aplicações *web* são divididas em duas partes: a parte *back-end* no *server-side* (lado servidor) e a parte *front-end* no *client-side* (lado do cliente), as quais utilizam tecnologias distintas e geralmente são organizadas de maneiras diferentes.

A parte *back-end* é onde fica a seção das regras de negócio da aplicação, é o gerenciamento da aplicação (LAMIM, 2014). Os programadores *back-end* se preocupam em desenvolver a parte de gestão dos dados, uma vez que esse sistema vai interagir com usuário, e dessa forma irá utilizar a *interface* criada por um programador *front-end* (LAMIM, 2014).

Então, se o *back-end* fica responsável pela gestão dos dados, cabe à parte *front-end* fazer a interação com o usuário. Dessa forma, programadores *front-end* cuidam de todo o segmento de interação com o usuário, utilização de CSS (*Cascading Style Sheets*), HTML (*HyperText Markup Language*), responsável pelo layout do sistema (LAMIM, 2014).

Neste projeto será criada uma API *RESTFull* com a linguagem de programação *back-end* PHP, em conjunto com o *Framework Zend Framework 2* (ZF2) e com a linguagem de programação *front-end JavaScript* utilizando o *Framework AngularJS*.

Segundo PLANSKY (2014), a aplicação REST (Transferência de Estado Representacional), é um design de arquitetura construído para servir aplicações em rede.

Ao construir uma aplicação, programadores planejam adquirir qualidade do código, ter ótimo desempenho, possuir uma estrutura padronizada para que desta forma a manutenção do *software* seja mais simples ou sem grandes complicações. Para garantir alguns desses benefícios e auxiliar no desenvolvimento, programadores utilizam *Frameworks*.

Os *Frameworks Zend Framework 2* (*back-end*) e *AngularJS* (*front-end*), utilizam o conceito do *design patterns MVC* (*Model-View-Controller*). Através desse projeto, será abordado como criar uma API *RESTFull* com ZF2 e AngularJS, mantendo organização e qualidade do código.

1.1 Tema

MVC no *Back-end* no *Front-end*: Criando API's *RESTFull* com *Zend Framework 2* e *AngularJS*.

1.2 Delimitação do Problema

Conforme Silva (2014), atualmente manter códigos *JavaScript* organizados é difícil, pois as aplicações *web* estão utilizando cada vez mais a linguagem *JavaScript* para deixar as páginas mais dinâmicas e com melhor aparência. Mas não somente com os códigos *front-end*, esse problema ocorre com linguagem *back-end*.

Essa falta de organização ocasiona sérios problemas, sendo que esses podem ocorrer tanto em pouco tempo quanto em um período maior de tempo após o desenvolvimento da aplicação. Os principais problemas são:

- dificuldade de manutenção de código;
- dificuldade de entendimento de código;
- falta de padronização.

A dificuldade de entendimento ocorre principalmente pela falta de padronização. A falta de padronização em uma aplicação é um grande problema, que se torna cada vez maior com o tamanho da aplicação.

Arquivos misturados, espalhados pelo projeto e códigos escritos de qualquer maneira, constituem grande exemplo da falta de padronização. Decorrente dessa ausência de padronização, a manutenção da aplicação se torna complicada e complexa. Com a utilização de um padrão, seria definida uma forma de organização da aplicação. Cada arquivo estaria em seu lugar, códigos escritos em arquivos adequados, por exemplo, arquivos com extensão *.js* seriam escritos somente com *JavaScript* e estariam em uma pasta separada, arquivos *CSS* em outra e para utilizar esses arquivos bastaria chamar dentro do arquivo que precisasse. Dessa forma, a organização da aplicação e do código ficaria melhor e mais entendível.

Os *designs patterns* são utilizados para resolver/diminuir esses problemas e a maioria dos *Frameworks* utilizam algum dos vários *designs patterns* existentes, o ideal seria utilizar um *Framework* para criação de uma aplicação, nesse contexto é que esse trabalho é apresentado.

1.3 Questões de Pesquisa

Quais as vantagens em utilizar o *design patterns* MVC?

Por que utilizar um *Framework JavaScript* MVC?

Como o *JavaScript* se comporta utilizando o *design pattern* MVC (*Model*, *View* e *Controller*)?

Como o AngularJS pode auxiliar no desenvolvimento *front-end*?

Como funciona a comunicação dos dados entre o MVC *back-end* (*Zend Framework 2*) e o MVC *front-end* (AngularJS)?

Como criar uma API *RESTFull* de maneira segura, utilizando o PHP em conjunto com o *Zend Framework 2*?

Como os *Frameworks* podem auxiliar no desenvolvimento tanto no *back-end* quanto no *front-end*?

2. OBJETIVOS

2.1 Objetivo geral

Criar uma API *RESTFull* para o Sistema de Gestão de Espaços da Unochapecó com *Zend Framework 2* e AngularJS utilizando o *design patterns* MVC tanto no *back-end* quanto no *front-end*.

2.2 Objetivos específicos

Criar uma API *RESTFull* com *Zend Framework 2* utilizando o componente *AbstractRestfulController* para manipular os dados do Sistema de Gestão de Espaços da Unochapecó.

Manipular os dados de uma API *RESTFull* com AngularJS que utiliza o *design pattern* MVC no *JavaScript*.

Utilizar técnicas de segurança OAuth para transição de dados entre as linguagens *back-end* e *front-end*.

Analisar e avaliar se a utilização do AngularJS auxilia na resolução de problemas de padronização, organização e manutenção do código *front-end*.

3. JUSTIFICATIVA

Para deixar os códigos de um determinado projeto de software organizados, a maioria dos programadores utilizam ferramentas, *Frameworks*, padrões de desenvolvimento, entre outros métodos.

Nas linguagens de desenvolvimento *server-side* (lado servidor), como o PHP, geralmente em grandes aplicações são utilizados *Frameworks*, por exemplo o *Zend Framework 2*. Segundo Chase (2006) existem vários *Frameworks*, como o *Zend Framework 2*, *Ruby on Rails*, *CakePHP* que utilizam alguns conceitos da programação, como os padrões:

- Orientação a objetos;
- Paradigma MVC (*Model-View-Controller*).

No *JavaScript* já é mais difícil ter uma organização de código, pois aplicações *web* estão utilizando cada vez mais *JavaScript* e muitas vezes não é enxergado como parte fundamental ou é ignorado pelos programadores (SILVA, 2014).

Para organização de código *JavaScript* existem *Frameworks*, mas também existem padrões de desenvolvimento que tornam possível a programação e a organização do código sem utilização de *Frameworks*.

Ter um código bem organizado e padronizado traz vantagens muito importantes. Algumas das vantagens são:

- manutenção do código;
- entendimento do código;
- facilidade de manipular o código.

Para as empresas e programadores autônomos que possuem algum *software* ativo, a manutenção às vezes pode causar transtornos, porque pode ser difícil encontrar algum erro ou acrescentar novos atributos ou regras na aplicação.

Já os programadores novos, que chegam nas empresas para trabalhar, encontram muitas dificuldades em se adaptar à forma de programar. Se o código estiver bem organizado e padronizado, essa dificuldade diminui consideravelmente, e o novo programador se adapta facilmente, o que é positivo para a empresa. Por esses motivos, ter um código organizado, tanto no *server-side* quanto no *client-side*, faz grande diferença.

Com essa dificuldade de organização, falta de padronização, manutenção e entendimento do código, o objetivo do trabalho é criar uma API *RESTFull* utilizando o design patterns MVC no *back-end* e no *front-end* utilizando os *Frameworks Zend Framework 2* e *AngularJS* respectivamente.

4. REVISÃO BIBLIOGRÁFICA

4.1 *Design Patterns*

No momento atual, desenvolver um projeto não é tarefa fácil. Essa dificuldade começa quando não é definido claramente um problema. Existe uma carência muito grande na documentação e nas soluções encontradas dos problemas (JUNQUEIRA; COSTA; LIRA, 1998).

O uso de *Design Patterns* (Padrão de projeto) vem para preencher essa lacuna, tornando-se um mecanismo para compartilhamento de conhecimentos entre desenvolvedores. Dessa forma, uma solução de um problema pode ser novamente aplicada por outro desenvolvedor para suprir sua necessidade e facilitar o trabalho (JUNQUEIRA; COSTA; LIRA, 1998). “O foco principal não é nem tanto tecnológico, mas sim o de criar uma cultura de catalogação de experiências e soluções para apoiar o desenvolvimento de software.” (JUNQUEIRA; COSTA; LIRA, 1998).

4.1.1 Origens dos *Design Patterns*

Os primeiros padrões de projetos surgiram por volta dos anos 80 através da engenharia civil, com um trabalho do engenheiro Christopher Alexander, que registrou suas experiências em resolver projetos de construções gerais, publicando os livros *Notes on the Synthesis of Form*, *The Timeless Way of Building* e *A Pattern Language*, descrevendo nesses livros todos os problemas que teve e como resolvê-los (HUMBERTO, 2010).

Na programação, os padrões de projetos surgiram por volta dos anos 70 na linguagem *Smalltalk*, quando Ward Cunningham e Kent Beck desenvolveram padrões para serem aplicados na *interface* do usuário. Nessa mesma época, Jim Coplien desenvolveu uma lista de padrões para o C++ e Erich Gamma, em sua tese de doutorado, reconheceu a importância de

acumular as estruturas de projetos que se repetiam com frequência no desenvolvimento do projeto (JUNQUEIRA; COSTA; LIRA, 1998).

4.1.2 Conceito de *Design Patterns*

Uma definição para *Design Patterns* pode ser: “Uma solução para um problema dentro de um contexto” (HUMBERTO, 2010).

Dessa forma, pode-se definir contexto como um conjunto de situações que se repetem, onde pode ser aplicado o *design patterns*, o problema um conjunto de objetivos e limitações que ocorrem dentro de um contexto e a solução é uma estrutura para ser aplicada na resolução de um determinado problema (JUNQUEIRA; COSTA; LIRA, 1998).

Em termos de orientação a objetos, *design patterns* identificam classes, instâncias, seus papéis, colaborações e a distribuição de responsabilidades. Seriam, então, descrições de classes e objetos que se comunicam, que são implementados a fim de solucionar um problema comum em um contexto específico (JUNQUEIRA; COSTA; LIRA, 1998).

Os padrões de projetos fazem mais do que identificar a solução de um problema, eles explicam porque a solução é necessária (JUNQUEIRA; COSTA; LIRA, 1998).

4.1.3 Por que e quando usar *Design Patterns*

No processo de desenvolvimento de *software* OO (Orientação a objetos), os *designs patterns* podem ser usados de diversas formas, que segundo Junqueira, Costa e Lira (1998) são elas:

- vocabulário para melhorar a comunicação entre os desenvolvedores;
- uma documentação mais completa;
- reduzem a complexidade do sistema através da definição de abstrações que estão acima das classes e instâncias;
- base de experiências reutilizáveis para construção de software;
- reduzem o tempo de aprendizado de uma biblioteca ou classe.

Os designs *patterns* devem ser utilizados em soluções que se repetem com variações, porque dessa forma o reuso não se faz necessário se for somente para um determinado problema (JUNQUEIRA; COSTA; LIRA, 1998).

Eles se aplicam adequadamente quando o desenvolvedor está mais interessado na existência de uma solução do problema do que no desenvolvimento da aplicação. Isto ocorre, na maior parte das vezes, quando o domínio do problema é o foco da questão (JUNQUEIRA; COSTA; LIRA, 1998).

Dessa forma, quando mais cedo utilizar os *designs patterns* em seu projeto, menor será o retrabalho em etapas mais avançadas (JUNQUEIRA; COSTA; LIRA, 1998).

4.1.4 Tipos de *Design Patterns*

Os autores do livro *Patterns of Software Architecture*, Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad e Michael Stal, definem três tipos de padrões, são eles:

- Padrão de Arquitetura;
- *Design pattern*;
- Idioma.

O padrão de arquitetura representa um esquema de organização estrutural do sistema. Se preocupa com o sistema inteiro, nos seus componentes macro e propriedades globais. Afetam a estrutura e a organização do sistema por inteiro (JUNQUEIRA; COSTA; LIRA, 1998).

Os *designs patterns* descrevem situações que se repetem, soluções para resolver problemas em contextos específicos e não afetam o sistema por inteiro.

E o idioma é um padrão de baixo nível, direcionado a uma linguagem de programação específica (JUNQUEIRA; COSTA; LIRA, 1998).

Conforme Junqueira, Costa e Lira (1998), Erich Gamma propõe um modo de categorizar os *designs pattern*, definindo famílias de padrões, são elas:

- *Creational Patterns*;
- *Structural Patterns*;
- *Behavioral Patterns*.

A *Creational Patterns* (Padrão Criacionais), refere-se à criação de instâncias de objetos e classes (HUMBERTO, 2010).

A *Structural Patterns* (Padrão Estrutural), diz respeito à organização de classes e de objetos (HUMBERTO, 2010).

E a *Behavioral Patterns* (Padrão Comportamental), preocupa-se com a forma que as classes e objetos interagem e com a distribuição de responsabilidades (HUMBERTO, 2010).

4.1.5 Design Patterns versus Frameworks

“Um *Framework* consiste em uma arquitetura concreta reutilizável que provê estruturas genéricas para uma família de softwares.” (JUNQUEIRA; COSTA; LIRA, 1998). Pode ser adaptado para atender várias necessidades em um determinado domínio.

Uma diferença básica de *framework* para bibliotecas, é que o *framework* determina quais objetos e métodos devem ser usados quando da ocorrência de eventos (JUNQUEIRA; COSTA; LIRA, 1998).

Os *designs patterns* ao contrário dos *frameworks*, permitem o reuso de microarquiteturas que já possuem um vínculo, sem uma implementação concreta (JUNQUEIRA; COSTA; LIRA, 1998).

Em geral, os *frameworks* são mais especializados que os *designs patterns*, dessa forma os *designs patterns* podem ser usados em um número maior de aplicações. Normalmente *frameworks* maduros utilizam vários *designs patterns* (JUNQUEIRA; COSTA; LIRA, 1998).

4.1.6 Considerações Finais sobre Design Patterns

Os *designs patterns* foram criados para compartilhar o conhecimento, sendo este bem documentado entre os desenvolvedores. Dessa forma, quando se tem um problema recorrente, que já foi resolvido por outra pessoa, pode-se utilizar o conhecimento dela para resolver o próprio problema sem muita dificuldade. Os *designs patterns* são utilizados para resolver um problema específico, eles não pensam no desenvolvimento da aplicação em si e eles são incorporados dentro dos *frameworks*.

Dentro do desenvolvimento de *software*, os *designs patterns* podem ser utilizados em diversas ocasiões, fornecendo reutilização de código, reduzindo a complexidade e o tempo de aprendizado.

Após abordar sobre os *designs patterns*, no próximo tópico será abordado sobre o *design pattern* MVC, que é um padrão de projeto utilizado no desenvolvimento de software e é implementado por vários *frameworks* como o AngularJS, *Symfony*, *Zend Framework*, *Ruby on Rails*, entre outros, para auxiliar na organização, padronização, manutenção e desenvolvimento do código (FRAMEWORKS, 2014).

4.2 Padrão MVC

O padrão MVC (*Model-View-Controller*) é utilizado nas aplicações para facilitar a manutenção e organização de código, ou seja, ter tudo em seu lugar e cada camada com sua responsabilidade. O MVC cria uma estrutura quebrada em três camadas: *Model*, *View* e *Controller* (BASTOS, 2011).

4.2.1 Camada *Model*

A *model* é responsável pela manipulação das informações e é sempre recomendado que se utilize as *models* para realizar consultas, cálculos e todas as regras de negócio do sistema. O modelo (*model*) tem acesso a toda informação, sendo ela vinda de algum banco ou de dados ou de arquivos XML (BASTOS, 2011).

Também implementa a lógica do negócio da aplicação, sendo responsável por obter os dados e convertê-los em conceitos significativos para a aplicação. Simplificando, fica responsável por tudo que diz respeito a dados (CAKEPHP, 2015).

Na primeira impressão, a *model* pode ser vista como a primeira camada de interação com os bancos de dados, mas de uma forma geral, ela representa os conceitos que são implementados na aplicação (CAKEPHP, 2015).

4.2.2 Camada *View*

A *view* é responsável por tudo o que o usuário visualiza na tela, toda *interface* e todas as informações são exibidas pela *view* (BASTOS, 2011).

Contém as classes necessárias que fazem a interação com o usuário. Uma aplicação pode utilizar várias *interfaces*, podendo ser trocadas quando for preciso sem interferir na aplicação. Para a *view* conseguir fazer isso, ela captura todas solicitações que o usuário faz e as converte em mensagens para que o controlador possa entender. As exceções que são lançadas pelo

controlador são capturadas pela *view*, que fica responsável por criar ações para repassá-las ao usuário. Mas para a *view* fazer isso, ela precisa ter acesso a objetos controladores (JONATHAN, 2011).

A *view*, como visto anteriormente, exibe as representações dos dados, mas ela não tem uma ligação direta com o modelo (CAKEPHP, 2015).

Essa camada não está somente limitada à representação de dados no formato HTML ou de texto, pode ser usada para vários formatos diferentes, dependendo da necessidade, como vídeos, músicas, documentos, entre outros (CAKEPHP, 2015).

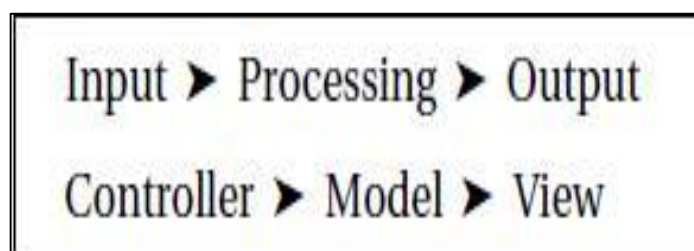
4.2.3 Camada *Controller*

O *controller* é responsável por controlar todo o fluxo de dados ou informações que passam pela aplicação. É no *controller* que se decide “se”, “o que”, “quando” e “onde” deve funcionar. O *controller* também é responsável por definir quais informações ou regras devem ser geradas ou acionadas e para onde devem ir as informações. É o *controller* que executa as regras de negócio (*model*) e manda as informações para a visão (*view*) (BASTOS, 2011).

Os controladores podem ser vistos como gerentes da aplicação, que tomam os cuidados necessários para que todos os recursos completem uma tarefa e sejam chamados corretamente. O *controller* aguarda alguma solicitação do usuário, verifica a validade da ação baseada na regra de autenticação e autorização ao determinado recurso, envia os dados para o modelo processar ou obtém os dados de um modelo e seleciona a forma correta de apresentação dos dados para o qual o usuário espera, para finalmente enviar os dados para a *view* (CAKEPHP, 2015).

Inicialmente o MVC foi construído com a ideia de mapear fluxo de dados. Dessa forma a entrada é o *controller*, o processamento é a *model* e a saída é a *view*, conforme a Figura 1.

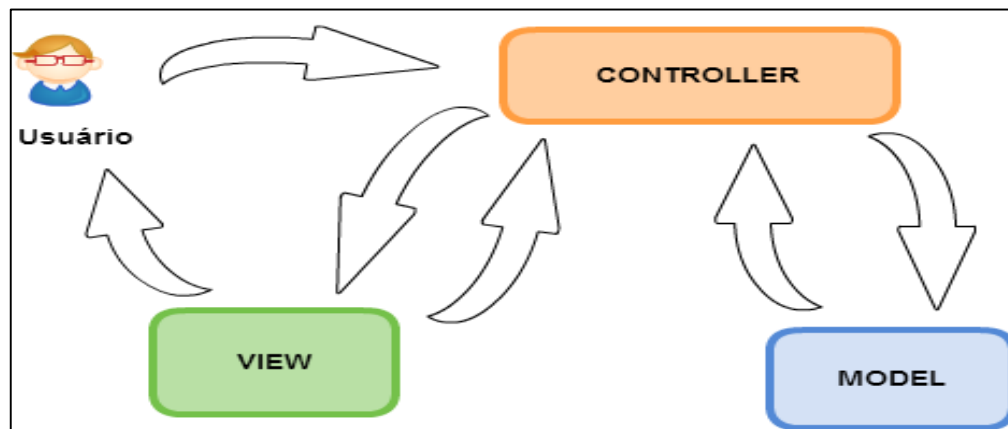
Figura 1: estrutura de fluxo de dados.



Fonte: Medeiros (2015).

Abaixo, na Figura 2, é apresentado como funciona o padrão MVC em uma aplicação.

Figura 2: funcionamento do MVC.



Fonte: elaborado pelo autor.

A título de exemplo, quando o usuário se comunica com a *interface* (*view*) para realizar um cadastro, o *controller* fica responsável por gerenciar as entradas ou ações e mapeia essas ações em comandos que são enviados para o modelo (*model*) e/ou para a visualização (*view*) para fazer alterações adequadas. O modelo fica responsável por fazer validações, filtros e a persistência dos dados. Por fim, a *view* fica responsável para apresentar as informações ao usuário. Tudo que ela faz é receber instruções ou dados do *controller* e exibir. Voltando para a *view*, o ciclo começa novamente (MEDEIROS).

4.2.4 Vantagens e desvantagens na utilização do padrão MVC

Existem algumas vantagens e desvantagens na utilização do padrão MVC para criação de aplicações, porém as vantagens são maiores.

Conforme Bastos (2011), as vantagens em se utilizar o MVC são:

- separação muito clara entre as camadas de visualização e de regras de negócio;
- manutenção do sistema ou da aplicação se torna muito mais fácil;
- reaproveitamento de código;
- as alterações na camada de visão não afetam as regras de negócio já implementadas na *model*;
- permite o desenvolvimento, teste e manutenção de forma separada entre as camadas;

- melhor organização do projeto;
- melhora o entendimento do projeto para novos desenvolvedores que não participaram na criação do mesmo;
- torna a aplicação escalável.

As desvantagens em se utilizar o MVC em um projeto são:

- em sistemas de baixa complexidade, o padrão MVC pode criar uma complexidade maior, ou seja, não é aconselhado utilizar em projetos pequenos;
- exige muita disciplina dos desenvolvedores em relação a separação em camadas;
- requer um tempo maior para modelar o sistema.

Como pode-se observar, existem muito mais vantagens em se utilizar o padrão MVC para o desenvolvimento de aplicações do que desvantagens. A utilização de um *framework* é ideal para quem quer um código bem organizado e de fácil entendimento; (BASTOS, 2011).

4.2.5 Considerações Finais sobre MVC

O padrão MVC foi desenvolvido para melhorar a organização e a qualidade do *software*, muito utilizado no desenvolvimento *web*. Esse padrão cria uma estrutura dividida em três camadas, *Model*, *View* e *Controller*, nos quais cada camada tem suas responsabilidades (BASTOS, 2011).

De forma resumida, a *model* fica responsável pela lógica do negócio, o *controller* fica responsável por controlar todo o fluxo de informações e a *view* é responsável por tudo o que o usuário visualiza na tela.

Esse padrão é ideal para quem quer ter qualidade de código, ter fácil manutenção do sistema, entendimento do código, entre vários outros benefícios como visto anteriormente. Para melhor organizar os códigos fontes de uma aplicação, foram criados vários *frameworks* utilizando o padrão MVC (BASTOS, 2011).

Em projetos desenvolvidos para web em PHP, o modelo MVC é muito utilizado para auxiliar na estruturação, organização e facilitar a manutenção do código (LOCKHART, 2015). Neste trabalho será utilizado a linguagem de programação web PHP, através desse contexto, o próximo tópico será tratado sobre a linguagem de programação PHP.

4.3 Linguagem de Programação PHP

O acrônimo PHP é a abreviação de *Hipertext Preprocessor* (pré-processador de hipertexto). É uma grandiosa e poderosa linguagem de programação *open source*, utilizada mundialmente, principalmente para o desenvolvimento *web*. Apesar disso, existe uma versão do PHP para ambiente *desktop*, chamado de PHP-GTK (SOARES, 2010, p. 28).

Uma das principais características do PHP é a sua capacidade de se misturar com o HTML. Dessa forma, é possível utilizar dentro do HTML códigos em PHP, tornando mais fácil a geração de páginas *web* dinâmicas (SOARES, 2010, p. 28). Para utilizar o PHP junto com o HTML, basta abrir o comando “<?php ” e fechar com “?>” e dentro desses dois comandos codificar o que for preciso.

4.3.1 História do PHP

O PHP foi criado em 1994, por Rasmus Lerdof e a primeira aparição do PHP foi um simples conjunto de binários *Common Gateway Interface* (CGI) escrito em linguagem de programação C. Originalmente ele foi usado pelo seu criador para um acompanhamento de visita para seu currículo *online*. Ele denominou o conjunto de *script* de "*Personal Home Page Tools*", mas foi chamado várias vezes de *PHP Tools*. Ao longo dos anos, novas funcionalidades foram elencadas (PHP, 2015).

Em 1995, Rasmus liberou o código fonte do PHP para o público, dessa forma os desenvolvedores usavam o código da forma que desejavam, com isso os usuários acabaram aperfeiçoando o código e arrumando *bugs* (erros) (PHP, 2015).

Em setembro do mesmo ano, Rasmus expandiu o PHP e por um período curto de tempo mudou seu nome para FI, abreviação de "*Forms Interpreter*". Essa nova implementação incluiu algumas funcionalidades básicas do PHP que ainda é utilizada atualmente. Um mês depois, Rasmus liberou uma reescrita completa do código, voltando para o nome de PHP, sendo nomeado de "*Personal Home Page Construction Kit*" e foi o primeiro lançamento considerado um avançado *script de interface* (PHP, 2015).

Segundo PHP (2015), a linguagem foi desenvolvida para, propositalmente, ser parecida com a linguagem C, para ser fácil a adoção dos programadores habituados em C, Perl e linguagens similares. Até aquele momento, o PHP era exclusivo para sistemas UNIX e sistemas compatível com POSIX e começava a ser explorada uma implementação para Windows NT (PHP, 2015).

4.3.1.1 PHP versão 3

O PHP 3 foi a primeira versão lançada semelhante ao PHP utilizado atualmente (2015). O PHP/FI estava ineficiente, não possuía recursos necessários para prover uma aplicação *eCommerce*¹ que os alunos Andi Gutmans e Zeev Suraski de Tel Aviv em Israel estavam desenvolvendo na Universidade e em 1997 começaram uma reescrita do interpretador.

Essa nova linguagem foi lançada com um novo nome, chamado de “PHP”, com significado de *Hypertext Preprocessor* (PHP, 2015).

Um dos pontos fortes dessa versão foram os recursos de extensibilidade. Também forneceu aos usuários uma *interface* para múltiplos bancos de dados, protocolos, API's. A facilidade de estender a linguagem atraiu dezenas de desenvolvedores. Isso foi a chave para o sucesso do PHP 3, além de outro ponto-chave que foi o suporte à programação Orientada a Objetos (PHP, 2015).

4.3.1.2 PHP versão 4

No ano de 1998, no inverno, depois do lançamento oficial do PHP 3, Andi Gutmans e Zeev Suraski, começaram a trabalhar em uma nova reescrita do PHP (PHP, 2015). Os objetivos dessa nova reescrita eram melhorar a performance das aplicações complexas e melhorar a modularização do código base do PHP (PHP, 2015).

O novo motor chamado de “*Zeend Engine*” (composto pelos primeiros nomes, Zeev e Andi), atingiu os objetivos da reescrita do PHP 3 e foi introduzido na metade de 1999. Essa

¹ *eCommerce* – *software* de comunicação e relacionamento com cliente.

nova versão PHP 3 é baseada nesse motor e foi lançada em maio de 2000. O PHP 4 introduziu suporte para maioria dos servidores *web*, sessões HTTP (*HyperText Transfer Protocol*) e maneiras seguras de manipular os dados de entrada feita pelos usuários (PHP, 2015).

4.3.1.3 PHP versão 5

Após longo desenvolvimento e vários pré-lançamentos, em julho de 2004 foi lançada a versão 5 do PHP. Foi principalmente impulsionado pelo *Zend Engine 2.0*, com novos recursos (PHP, 2015).

Atualmente, o time de desenvolvimento do PHP tem dezenas de desenvolvedores e também dezenas de outros trabalhando em algo relacionado ao PHP. É seguro imaginar que o PHP agora está instalado em dezenas ou centenas de milhões de domínios em todo o mundo (PHP, 2015).

4.3.2 Tipos de dados no PHP

Segundo Soares (2010, p. 46), no PHP existem oito tipos de dados que estão divididos em três grupos:

- Escalares;
- Compostos;
- Especiais.

O grupo dos Escalares é composto pelos seguintes tipos de dados:

- Inteiros (*int*);
- Ponto Flutuante (*float*, *double* e *real*);
- *String*;
- Booleanos.

No segundo grupo, o grupo dos Compostos, existem os seguintes tipos de dados:

- *Arrays*;
- Objetos.

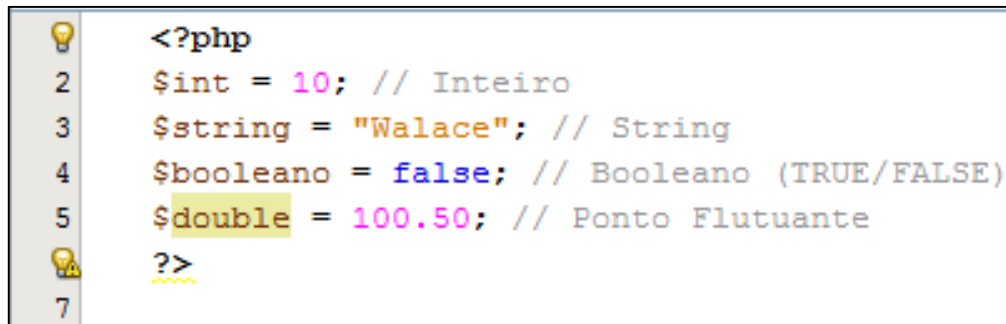
Por fim o grupo dos Especiais, composto pelos seguintes tipos de dados:

- Recursos;
- NULO (*null*).

O mais interessante da linguagem PHP é a não necessidade de declarar o tipo da variável antes de utilizá-la, pois o PHP faz isso automaticamente (SOARES, 2010, p. 46).

Por exemplo o código mostrado na Figura 3.

Figura 3: utilização de variáveis no PHP.



```

1  <?php
2  $int = 10; // Inteiro
3  $string = "Walace"; // String
4  $booleano = false; // Booleano (TRUE/FALSE)
5  $double = 100.50; // Ponto Flutuante
6  ?>
7

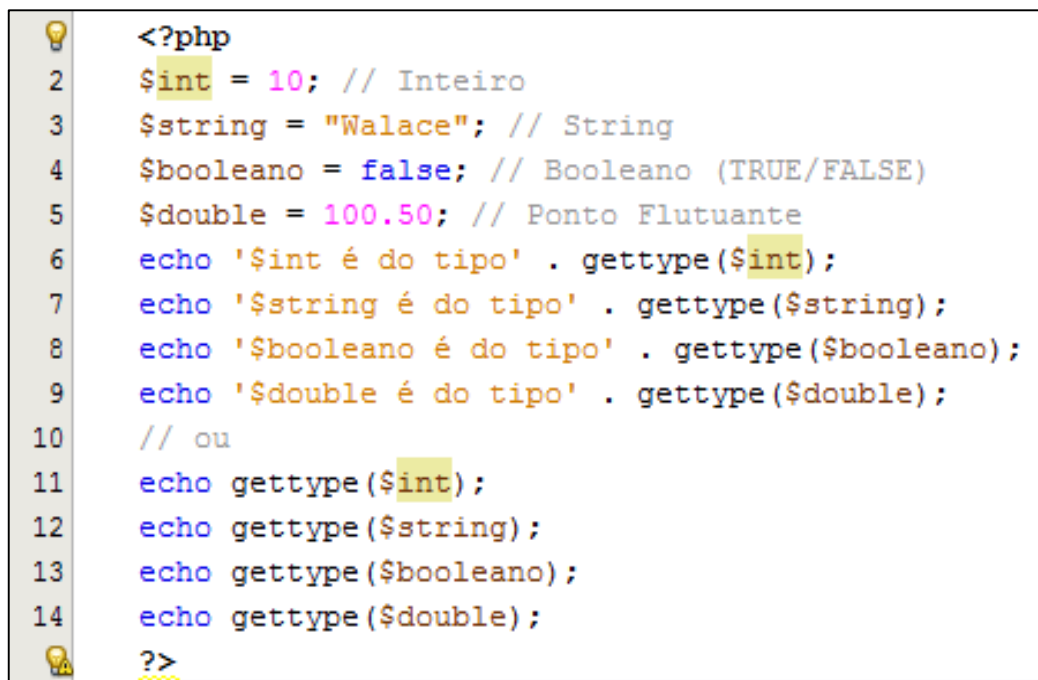
```

Fonte: Soares (2010, p. 46).

Para saber o tipo de uma variável, é utilizado o comando *gettype()*. Sua sintaxe é a seguinte: *string gettype (mixed \$var)* (PHP, 2015).

Na Figura 4 um exemplo da utilização do *gettype()*.

Figura 4: utilização do *gettype()* no PHP.



```

1  <?php
2  $int = 10; // Inteiro
3  $string = "Walace"; // String
4  $booleano = false; // Booleano (TRUE/FALSE)
5  $double = 100.50; // Ponto Flutuante
6  echo '$int é do tipo' . gettype($int);
7  echo '$string é do tipo' . gettype($string);
8  echo '$booleano é do tipo' . gettype($booleano);
9  echo '$double é do tipo' . gettype($double);
10 // ou
11 echo gettype($int);
12 echo gettype($string);
13 echo gettype($booleano);
14 echo gettype($double);
15 ?>

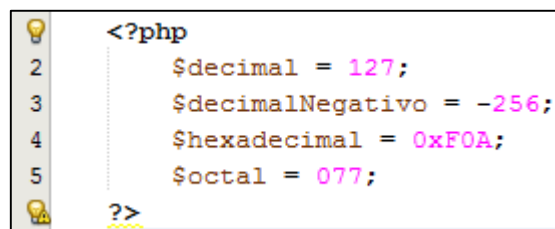
```

Fonte: Soares (2010, p. 46).

4.3.2.1 Inteiros no PHP

Um inteiro é um número sem decimais, podendo ser positivo ou negativo, além de poder ser representado na base decimal, hexadecimal ou octal. Para representar um número na base octal, ele deve iniciar com zero (0), já para representar um número na base hexadecimal, deve iniciar de 0x. Conforme a Figura 5, é observado um exemplo de inteiros em cada uma das bases (SOARES, 2010, p. 47).

Figura 5: inteiros nas bases decimal, hexadecimal e octal.



```
<?php
2     $decimal = 127;
3     $decimalNegativo = -256;
4     $hexadecimal = 0xF0A;
5     $octal = 077;
?>
```

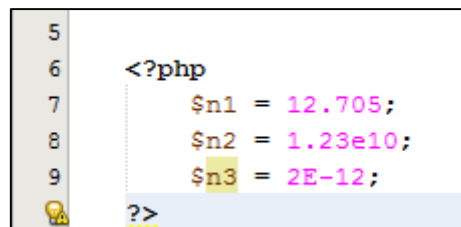
Fonte: Soares (2010, p. 47).

O tamanho do inteiro depende da plataforma em que o PHP está sendo executado. Mas em geral o tamanho é próximo a dois bilhões, equivalente a um número de 32 bits (SOARES, 2010, p. 47).

4.3.2.2 Ponto flutuante no PHP

O ponto flutuante, de precisão dupla ou um número real, pode ser definido de umas das seguintes formas, conforme a Figura 6 (SOARES, 2010, p. 49).

Figura 6: tipos de pontos flutuantes.



```
<?php
7     $n1 = 12.705;
8     $n2 = 1.23e10;
9     $n3 = 2E-12;
?>
```

Fonte: Soares (2010, p. 49).

Números inteiros muito grandes são considerados pontos flutuantes pelo PHP. O tamanho máximo do ponto flutuante também depende da plataforma em que está sendo executado, mas

o limite fica em torno de $1.8e308$ com uma precisão de 14 dígitos decimais, isso equivale ao padrão IEEE² (Instituto de Engenheiros Elétricos e Eletrônicos) de 64 bits (SOARES, 2010, p. 49).

4.3.2.3 *Strings* no PHP

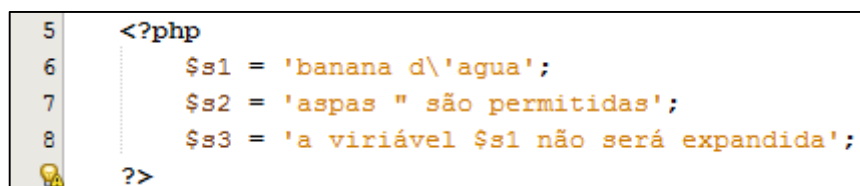
Segundo Soares (2010, p. 49), uma *string* é uma série de caracteres. Para o PHP, um caractere é exatamente um *byte*, ou seja, existem 256 caracteres diferentes no PHP.

Para definir uma *string*, deve-se utilizar aspas simples, aspas duplas, sintaxe *Heredoc* e sintaxe *Nowdoc* (PHP 5.3).

Para definir uma *string* com aspas simples, toda sequência de caracteres deve estar entre as aspas simples ('texto aqui').

Caso o texto contenha aspas simples, deve ser precedido pela barra invertida (\), caso contrário o PHP acusará um erro. Qualquer outro caractere que for enviado ao navegador, ficará sem tratamento. Outro exemplo de utilização de aspas simples é apresentado na Figura 7. As variáveis que estão entre as aspas simples, não serão expandidas (mostradas) no navegador (SOARES, 2010, p. 49).

Figura 7: exemplos de *Strings* com aspas simples.



```

5  <?php
6  $s1 = 'banana d\'agua';
7  $s2 = 'aspas " são permitidas';
8  $s3 = 'a variável $s1 não será expandida';
9  ?>

```

Fonte: Soares (2010, p. 50).

Para as aspas duplas usa-se o mesmo conceito das aspas simples, todo o texto deve estar entre as aspas duplas ("texto aqui") (SOARES, 2010, p. 50).

Conforme Soares (2010, p. 50), as vantagens básicas em se utilizar as aspas duplas são:

- maior quantidade de caracteres especiais disponíveis (caracteres escape);
- as variáveis são expandidas (mostradas).

A Tabela 1 mostra os caracteres de escape disponíveis.

² IEEE é a maior associação profissional do mundo dedicada ao avanço da inovação tecnológica e excelência para o benefício da humanidade (HUMANITY, 2015).

Tabela 1: caracteres de escape disponíveis no PHP.

Caracteres	Resultado
\n	<i>Linefeed</i> (LF)
\r	<i>Carriage return</i> (CR)
\t	<i>Tab</i> (HT)
\\	Barra invertida
\\$	\$
\“	Aspas duplas
\0xxx	Um número octal
\xXX	Um número hexadecimal

Fonte: Soares (2010, p. 50).

Outra forma de especificar uma *string* é a utilização da sintaxe *Heredoc*, que utiliza um caractere para delimitar a *string*. Deve-se utilizar <<< (três sinais de menor) e em seguida definir um delimitador, o qual deve estar presente no final da *string*, conforme a Figura 8 (SOARES, 2010, p. 51).

Figura 8: exemplo de *String* com a sintaxe *Heredoc*.

```

5      <?php
6          $bar = <<<EOL
7              Este é um exemplo da utilização do Formato
8              Heredoc no PHP para definir
9              Uma strig
10     EOL;
11     ?>

```

Fonte: Soares (2010, p. 51).

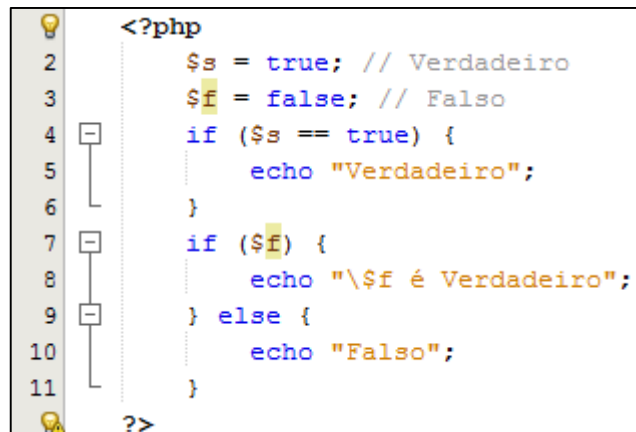
A sintaxe *Heredoc* segue as mesmas regras das strings com aspas duplas.

E por fim a sintaxe *Nowdoc*, que surgiu na versão 5.3 do PHP, utilizada para especificar novos tipos de *strings*, principalmente para textos longos. O padrão *Nowdoc* é muito semelhante ao do *Heredoc*, a diferença está que o delimitador do *Nowdoc* esteja entre aspas simples (SOARES, 2010, p. 52).

4.3.2.4 Booleano no PHP

O tipo booleano é muito simples, ele aceita apenas dois tipos de valores. Para o valor verdadeiro utilizasse *true*, para valor falso utilizasse *false*, conforme a Figura 9 (SOARES, 2010, p. 52).

Figura 9: exemplo script em PHP com *booleano*.



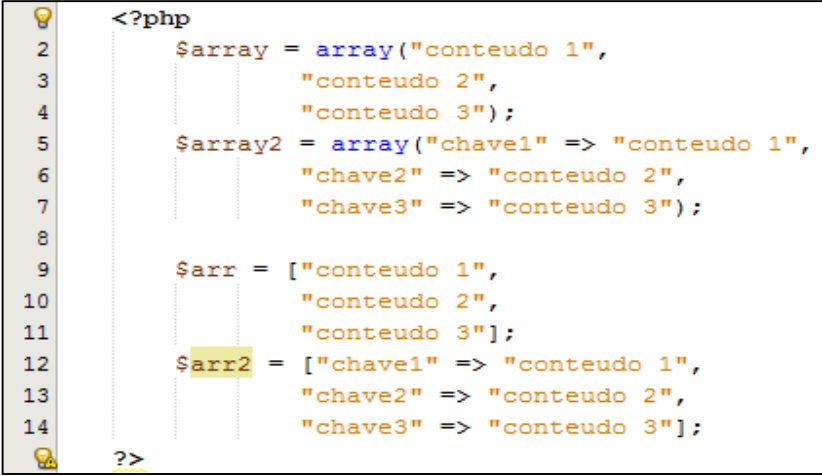
```
<?php
2      $s = true; // Verdadeiro
3      $f = false; // Falso
4      if ($s == true) {
5          echo "Verdadeiro";
6      }
7      if ($f) {
8          echo "\$f é Verdadeiro";
9      } else {
10         echo "Falso";
11     }
?>
```

Fonte: Soares (2010, p. 52).

4.3.2.5 Arrays

Segundo Soares (2010, p. 52): “No PHP *arrays* são mapas ordenados de chaves e valores, ou seja, é possível atribuir a um elemento do array uma chave e um valor”. Com isso, é possível representar listas, coleções, pilhas, filas, etc., além disso o *array* suporta um outro *array* dentro dele, podendo criar *arrays* multidimensionais. Dessa forma, é fácil representar uma estrutura em árvore no PHP através dos *arrays* (SOARES, 2010, p. 52).

No PHP existem duas formas para definir um *array*. A primeira é a forma simples, no qual utiliza-se o construtor *array()* ou através de colchetes (*[]*), que substitui o construtor *array()*. Na Figura 10 observa-se um exemplo de declaração de *arrays* (SOARES, 2010, p. 53).

Figura 10: exemplo de declaração de *array*.


```

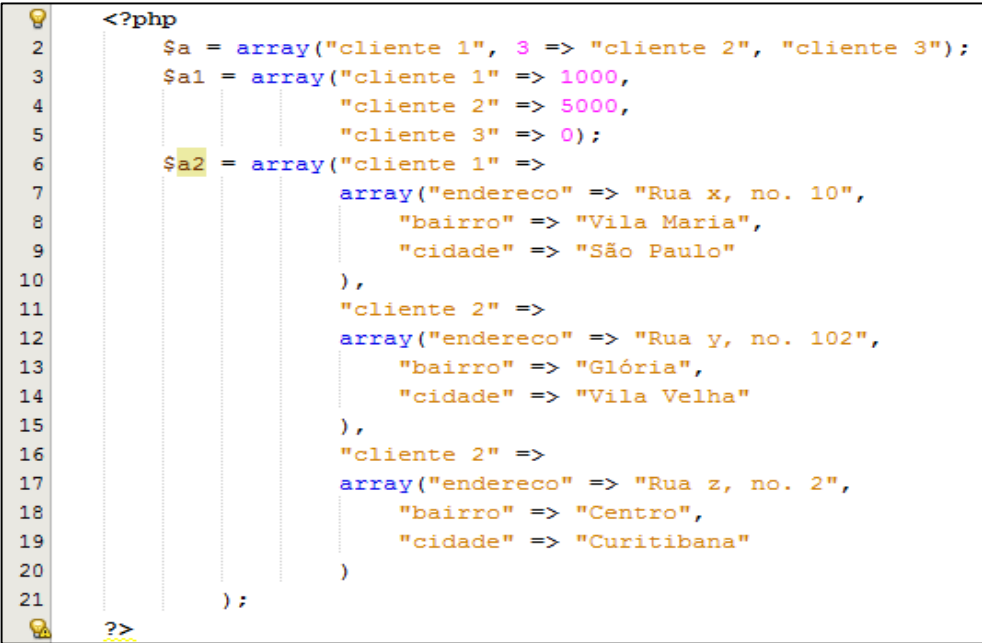
<?php
2   $array = array("conteudo 1",
3               "conteudo 2",
4               "conteudo 3");
5   $array2 = array("chave1" => "conteudo 1",
6                 "chave2" => "conteudo 2",
7                 "chave3" => "conteudo 3");
8
9   $arr = ["conteudo 1",
10         "conteudo 2",
11         "conteudo 3"];
12  $arr2 = ["chave1" => "conteudo 1",
13          "chave2" => "conteudo 2",
14          "chave3" => "conteudo 3"];
?>

```

Fonte: elaborado pelo autor.

Se não for definida uma chave, o PHP atribui automaticamente um número crescente ao elemento do *array*. Todo *array* criado sem nenhum valor, seu índice começa com -1, cada adição de valor no *array* é somado um valor ao índice caso não haja uma chave ao valor.

Arrays associativos é quando existe essa associação da chave com o valor (SOARES, 2010, p. 53, 54). A Figura 11 mostra um exemplo de *array* com definição de chaves e sem chaves.

Figura 11: exemplo de declaração de *array* com chaves e sem chaves.


```

<?php
2   $a = array("cliente 1", 3 => "cliente 2", "cliente 3");
3   $a1 = array("cliente 1" => 1000,
4              "cliente 2" => 5000,
5              "cliente 3" => 0);
6   $a2 = array("cliente 1" =>
7               array("endereco" => "Rua x, no. 10",
8                     "bairro" => "Vila Maria",
9                     "cidade" => "São Paulo"),
10              "cliente 2" =>
11              array("endereco" => "Rua y, no. 102",
12                    "bairro" => "Glória",
13                    "cidade" => "Vila Velha"),
14              "cliente 2" =>
15              array("endereco" => "Rua z, no. 2",
16                    "bairro" => "Centro",
17                    "cidade" => "Curitiba"));
21  );
?>

```

Fonte: Soares (2010, p. 53).

No exemplo da Figura 11, o *array* \$a tem os elementos 0 = “cliente 1”, 3 = “cliente 2” e 4 = “cliente 3”. Já o *array* \$a1 tem um *array* associativo e no *array* \$a2 um *array* multidimensional associativo (SOARES, 2010, p. 53).

A segunda forma é a direta, onde apenas é atribuído a um elemento do *array* um valor, conforme a Figura 12 (SOARES, 2010, p. 53).

Figura 12: exemplo de declaração de *array* na forma direta.

```
23 <?php
24     $array['chave'] = 'valor';
    ?>
```

Fonte: elaborado pelo autor.

Conforme a Figura 13, para remover um elemento do *array* ou o próprio *array*, utilizasse a função *unset()*.

Figura 13: remoção de elemento de um *array* e de um *array*.

```
<?php
$a = array("cliente 1", 3 => "cliente 2", "cliente 3");
$a1 = array("cliente 1" => 1000,
            "cliente 2" => 5000,
            "cliente 3" => 0);
$a2 = array("cliente 1" =>
            array("endereco" => "Rua x, no. 10",
                  "bairro" => "Vila Maria",
                  "cidade" => "São Paulo"
            ),
            "cliente 2" =>
            array("endereco" => "Rua y, no. 102",
                  "bairro" => "Glória",
                  "cidade" => "Vila Velha"
            ),
            "cliente 3" =>
            array("endereco" => "Rua z, no. 2",
                  "bairro" => "Centro",
                  "cidade" => "Curitiba"
            )
        );
unset($a[3]); // Remove o elemento 3 do array $a
unset($a1); // Remove o array $a1
?>
```

Fonte: Soares (2010, p. 54).

4.3.2.6 Objetos

Segundo Soares (2010, p. 55): “Para o PHP objeto é uma instância de uma classe, ou seja, é a vinculação de uma classe a uma variável. Isso é possível por meio da instrução *new*”. Na Figura 14 é apresentado um exemplo de objeto.

Figura 14: exemplo de objeto no PHP.

```

14 <?php
15     class nada
16     {
17         function imprime()
18         {
19             echo "Esta classe não faz nada";
20         }
21     }
22     $n = new nada();
23     $n->imprime();
    ?>

```

Fonte: Soares (2010, p. 52).

4.3.2.7 Recursos

Segundo Soares (2010, p. 55): “Os recursos são variáveis especiais no PHP, as quais referenciam recursos externos ao PHP”. Esses recursos são criados e utilizados por funções especiais no PHP, alguns exemplos de utilização de recursos são: manipulação de banco de dados, imagens, arquivos, etc. A Figura 15 mostra um exemplo de utilização de recurso no PHP.

Figura 15: exemplo de utilização de recurso no PHP.

```

<?php
$pdf = pdf_new();
pdf_open_file($pdf, "test.pdf");
pdf_begin_page($pdf, 595, 842);
$arial = pdf_findfont($pdf, "Arial", "host", 1); pdf_setfont($pdf, $arial, 10);
pdf_show_xy($pdf, "Existem mais coisas entre o céu ea terra, Horácio,", 50, 750);
pdf_show_xy($pdf, "do que sonha a nossa filosofia", 50, 730);
pdf_end_page($pdf);
pdf_close($pdf);
?>

```

Fonte: Php (2015).

Para ver o tipo de recurso que uma variável possui, devesse utilizar a função *get_resource_type()*, como mostra a Figura 16.

Figura 16: exemplo de utilização do comando *get_resource_type()* PHP.

```
<?php

$PDF = pdf_open_file($pdf, "test.pdf");
if (is_resource($PDF)) {
    echo get_resource_type($PDF);
}
?>
```

Fonte: Php (2015)

4.3.2.8 Nulo

Para representar uma variável sem nenhum valor, é utilizado o tipo nulo. Não pode confundir o nulo com o "" (aspas duplas - vazio). O tipo nulo aceita apenas o valor *null*, conforme a Figura 17 (SOARES, 2010, p. 56).

Figura 17: exemplo de utilização do tipo *null*.

```
<?php
1
2 $nulo = null;
3
?>
```

Fonte: Soares (2010, p. 56).

Segundo SOARES (2010, p. 56) o PHP vai considerar uma variável com o valor nulo nas seguintes situações:

- se a variável receber o valor *null*;
- se a variável não tiver sido criada ainda;
- se a função *unset()* tiver sido usada.

Para verificar se uma variável está com o valor *null* utilize a função *is_null()* (SOARES, 2010, p. 56).

4.3.3 Considerações finais sobre o PHP

A linguagem de programação PHP está entre as linguagens mais utilizadas do mundo e é adequada para o desenvolvimento de aplicações *web* (MÜLLER, 2015). O PHP se encontra na versão 5, uma versão estável e com vários recursos.

Nesse tópico foi estudado sobre a história do PHP, suas principais características e seus tipos de dados e a forma como manipulá-los.

O PHP é umas das linguagens de programação que pode ser utilizada para desenvolver *Web Services* para realizar a integração entre sistemas e comunicação entre aplicações. Através desse contexto, no próximo tópico será abordado mais detalhadamente sobre os *Web Services*.

4.4 Web Service

Com o avanço da *internet* e com evolução das redes de computadores, surgiram as aplicações distribuídas. Antigamente todo o processamento era centralizado em apenas um servidor. Com o advento dos *middlewares*³, esses processamentos foram distribuídos em vários servidores. Com essa necessidade de fazer a comunicação entre esses serviços surgiram os *Web Services* (LIMA, 2012, p. 8).

Web Service é uma solução/maneira utilizada para fazer integração de sistemas e na comunicação entre aplicações diferentes, transmitindo e enviando dados em formato XML⁴ (OFICINADANET, 2015).

Através dessa tecnologia é possível criar novas aplicações e fazê-la interagir com alguma outra aplicação existente, mesmo sendo de plataformas diferentes (OFICINADANET, 2015). A Figura 18 mostra como é o formato de um arquivo XML.

Figura 18: formato de um arquivo XML.

```
- <xml>
- <Inventario>
- <Computadoras>
+ <Marca nombre="HP">
- <Marca nombre="Acer">
  <Modelo>TYJGHFB/345</Modelo>
  <Almacen>34</Almacen>
  <Procesador>Pentium IV a 1.8 Ghz</Procesador>
  <Memoria>2 GB RAM</Memoria>
  <Disco>40 GB</Disco>
  </Marca>
+ <Marca nombre="DELL">
  </Computadoras>
</Inventario>
</xml>
```

Fonte: Oocities (2015)

Essa tecnologia pode trazer agilidade de processos para as empresas e também eficácia na comunicação entre as produções (OFICINADANET, 2015).

³ *Software* localizado entre o sistema operacional e a aplicação os *middlewares* são serviços que suportam o desenvolvimento e a execução de aplicações distribuídas (SANTOS, 2015).

⁴ XML - uma linguagem de marcação recomendada pela W3C para a criação de documentos com dados organizados hierarquicamente, tais como textos, banco de dados ou desenhos vetoriais (PEREIRA, 2009).

Os *Web Services* auxiliaram a *internet* a se tornar uma plataforma para aplicações ao invés de uma plataforma voltada a seres humanos (DAIPRAI, 2011, p.35 apud CERAMI, 2002).

Atualmente existem duas arquiteturas de *Web Service*, os que utilizam SOAP e XML e os que são baseados em REST, que além do XML utiliza o JSON⁵ para trocar informações (DAIPRAI, 2011, p.35 apud BEAN, 2009).

Há duas formas de utilizar um *Web Service*, para aplicativos de componentes reutilizáveis, como conversão de moeda, previsão de tempo e para conectar *softwares* de plataformas diferentes (W3SCHOOLS, 2015).

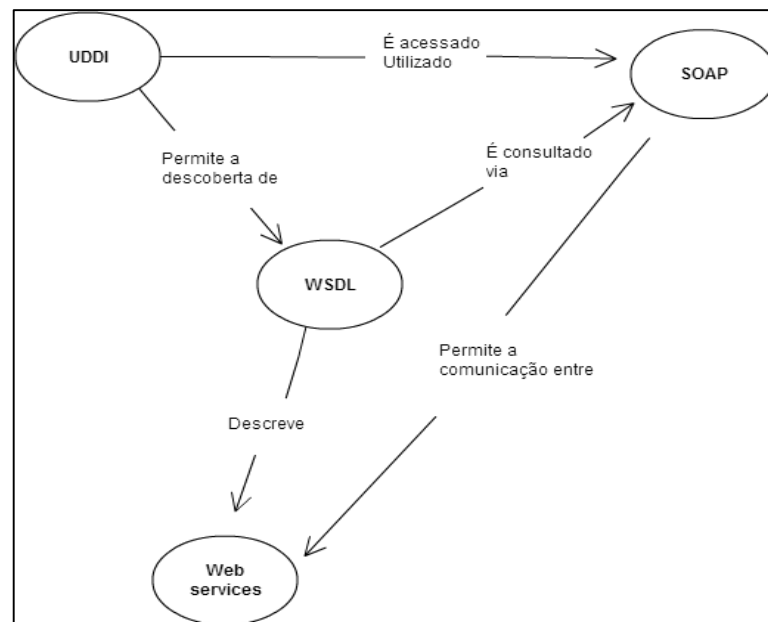
Sempre houve uma grande preocupação das empresas em transmitir seus dados pela *internet*. Mas com a vinda dos *Web Services*, as empresas podem enviar seus dados de forma simples, totalmente isolados da base dados e com segurança (OFICINADANET, 2015).

A comunicação entre as aplicações faz uso de quatro elementos que ficam responsáveis por fazer o encapsulamento da requisição e da resposta entre um servidor e outro (LIMA, 2012, p. 12). Os elementos são:

- XML (*Extensible Markup Language*);
- SOAP (*Simple Object Access Protocol*);
- WSDL (*Webservice Description Language*);
- UDDI (*Universal Description Discovery and Integration*);

A WSDL fornece a linguagem mais avançada para a descrição e definição de um *Web Service*. A UDDI fica responsável por serviços *Web Services* e responsável pela publicação. Toda vez que for feita a chamada de um *Web Service*, é necessário localizá-lo, descobrir sua *interface* e a semântica de sua chamada e também configurar o *software* para realizar a chamada ao *Web Service* (LIMA apud. UNIVERSAL, 2011). A Figura 19 mostra o relacionamento entre os elementos.

⁵ JSON é basicamente um formato leve de troca de informações/dados entre sistemas (GAMA, 2015).

Figura 19: relacionamento entre os elementos.

Fonte: elaborado pelo autor.

4.4.1 Tecnologia SOAP

Inicialmente o SOAP foi projetado para troca de mensagens entre computadores através da *internet* (LIMA, 2012, p. 17) e para chamar aplicações remotas ou para troca de mensagens (LIMA, 2012, p. 17 apud SIMPLE, 2011).

Um serviço *web* que é baseado em SOAP é qualquer serviço que esteja disponível na *internet* ou sobre uma rede privada, utilize o XML como envio de mensagens, não esteja preso a qualquer sistema operacional ou linguagem de programação, seja descrito com gramática XML comum e descoberto através de um mecanismo de busca simples (DAIPRAI, 2011, p.36 apud CERAMI, 2002).

As mensagens são transportadas pelo protocolo HTTP através de pacote e beneficia-se do HTTP para passar pelos *Firewalls* e roteadores facilitando a comunicação entre os *Web Services* (LIMA, 2012, p. 17).

Os *Web Services* oferecem dois tipos de modelos:

- RPC (*Remote Procedure Call*);
- *Document*;

O RPC permite a modelagem das chamadas dos métodos com parâmetros e receber valores de retorno. O SOAP, através do seu corpo (elemento *Body*), leva o nome do método e os parâmetros de entrada na chamada. O RPC envia um valor de retorno para o método chamado (LIMA, 2012, p. 17).

No *Document* o corpo da mensagem do SOAP tem um trecho de XML que é enviado ao serviço ao invés de métodos e parâmetros (LIMA, 2012, p. 17).

Os *Web Services* que são implementados em RPC necessitam da elaboração da *interface* de suas operações. No estilo *Document*, é necessário a criação de um *XML Schema* (esquemas) para definição das regras de validação e requer mais esforço na implementação (LIMA, 2012, p. 18).

Existem três agentes que são os principais para a arquitetura dos serviços *web* que são baseados em SOAP: o provedor do serviço, o consumidor do serviço e o registro do serviço (DAIPRAI, 2011, p.36 apud CERAMI, 2002).

Provedor de serviço: prove o serviço na *web*, é implementado em um servidor *web*.

Consumidor de serviço: é quem consome o serviço, o consumidor envia uma requisição para o serviço *web* para solicitar algum recurso que está provido pelo serviço.

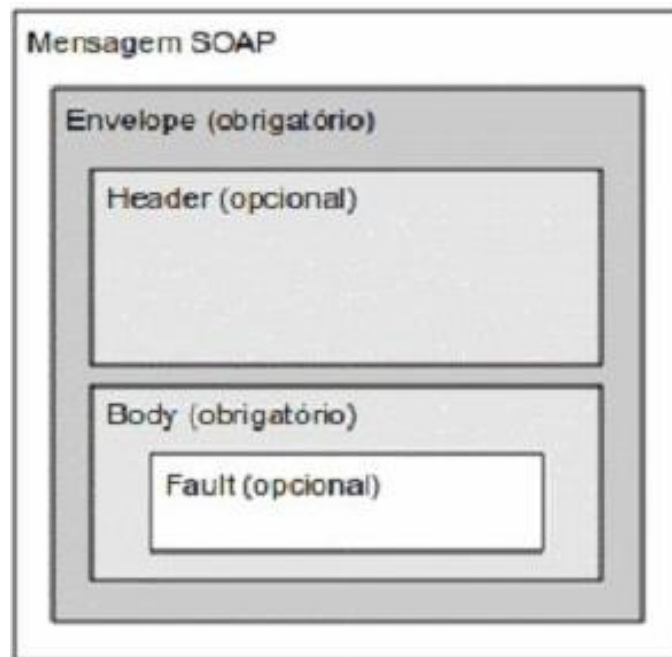
Registro de serviço: é um local onde são armazenadas informações sobre serviço *web*.

A W3C define o SOAP como um protocolo leve cujo seu objetivo consiste em prover uma infraestrutura que permita troca de informação em ambientes distribuídos, descentralizados (DAIPRAI, 2011, p.42 apud W3C, 2007).

O SOAP é um tipo de extensão do HTTP que suporta as mensagens do tipo XML. O SOAP envia e recebe mensagens XML através de uma requisição HTTP, ao invés de usar o HTTP para fazer as requisições. Para manipular de maneira correta as mensagens XML é necessário um servidor HTTP, como o *Apache* ou *Microsoft Internet Information Services* (DAIPRAI, 2011, p.35 apud NEWCOMBER, 2002).

Toda mensagem do SOAP possui um elemento *Envelope*, um elemento *Header*, um elemento *Body* e um elemento *Fault* (DAIPRAI, 2011, p.42 apud CERAMI, 2002), como pode ser visto na Figura 20.

Figura 20: elementos da mensagem SOAP.



Fonte: Daiprai (2011, p.35 apud CERAMI, 2002).

Segundo Daiprai (2011), cada um dos elementos possui suas regras. A regra de cada elemento é abordada da seguinte maneira:

- *Envelope*: é o elemento raiz da mensagem SOAP. Possui os elementos *Header* e *Body* como subelementos, sendo que o *Header* é opcional e o *Body* obrigatório. A versão do documento SOAP está indicada no elemento XML que se encontra dentro do *Envelope*;
- *Header*: contém informações adicionais sobre a mensagem. Todas as informações contidas nesse elemento podem ser modificadas no processo de troca de mensagens;
- *Body*: é o corpo da mensagem SOAP. São os dados contidos na mensagem que irão ser trocados;
- *Fault*: é um elemento utilizado pelo *Body* caso ocorra algum erro no processamento da mensagem pelo cliente ou pelo servidor.

Para que haja a troca de mensagem, é necessário dois agentes principais, o provedor e o consumidor.

4.4.2 Tecnologia REST

Outra maneira de comunicação entre aplicação é a utilização do REST que é um conjunto de princípios que definem como *Web standards*⁶ e como o HTTP devem ser usados.

O termo REST foi definido por Roy T. Fielding em sua tese de PhD. Roy foi uns dos principais desenvolvedores de vários protocolos que existem para a *web*. REST é um estilo de alto nível que pode ser implementado com diversas tecnologias, inclui conceitos de recursos e uma *interface* uniforme, ou seja, utilizando a ideia de que todos os recursos devem usar os mesmos métodos (TILKOV, 2008).

O HTTP foi baseado no estilo REST. O HTTP instancia a *interface* do REST, consistindo nos verbos HTTP. Utilizar o HTTP sem seguir os princípios do REST, é a mesma coisa que abusar do HTTP (TILKOV, 2008).

Esses princípios são:

- dê a todos os recursos um identificador;
- vincule as coisas;
- utilize métodos padronizados;
- recursos com múltiplas representações;
- comunique sem estado.

O estilo arquitetural do REST é baseado em cinco (5) conceitos, sendo eles: o recurso, a representação, o identificador uniforme, *interface* unificada e o escopo de execução. Também possui três princípios, são eles: princípio de endereçabilidade, princípio do estado não persistente e o princípio da conectividade (DAIPRAI, 2011, p.45 apud RICHARDSON; RUBY, 2007).

Os conceitos estão definidos nessa forma:

Recurso

Uma definição cabível é qualquer coisa que possa ser nomeada e endereçada, além de poder ser armazenada em um computador e representada como fluxo de *bits*, um documento, imagem ou um conceito abstrato (DAIPRAI, 2011, p.45 apud FLANDERS, 2008).

⁶ *Web standards* é um termo mais amplo que refere-se aos padrões web como um todo e não somente as linguagens de marcação e CSS (PEREIRA, 2006).

Representação

Os serviços manipulam as representações dos recursos, não manipulando diretamente os recursos (DAIPRAI, 2011, p.46 apud RICHARDSON; RUBY, 2007).

Identificador Uniforme

Cada recurso deve estar associado a um URI (*Uniform Resource Identifier*) que atua como nome e localizador do recurso na *web* (DAIPRAI, 2011, p.46 apud RICHARDSON; RUBY, 2007).

Interface unificada

Uma ação a ser executada é definida pelo método HTTP. Esse protocolo oferece quatro (4) métodos principais: *DELETE*, *GET*, *POST*, *PUT*.

Essas quatro ações são detalhadas da seguinte forma:

- HTTP GET – retorna a representação de um recurso;
- HTTP PUT e POST – cria um novo recurso;
- HTTP PUT – altera um recurso existente;
- HTTP DELETE – deleta/exclui um recurso existente.

Todas essas operações fazem analogia as operações de persistência de dados no desenvolvimento de *software*, conhecidos como CRUD (*Create, Retrieve, Update and Delete*).

Alguns comandos como por exemplo DELETE e PUT, não são suportados por alguns navegadores. Dessa forma, quando um usuário desejar usar algum desses recursos deve substituí-los pelos métodos GET e POST (DAIPRAI, 2011, p.47 apud RICHARDSON; RUBY, 2007).

Escopo de execução

No escopo de execução, cada URI contém não apenas o caminho do serviço, mas também qualquer parâmetro necessário para a identificação única de cada recurso (DAIPRAI, 2011, p.48 apud RICHARDSON; RUBY, 2007).

E os princípios estão definidos dessa maneira:

Endereçabilidade

Para um serviço *web* ser chamada de endereçável, é quando seu conjunto de dados é exposto como uma série de recurso, cada recurso com seu URI (*Uniform Resource Identifier*) (DAIPRAI, 2011, p.48 apud RICHARDSON; RUBY, 2007).

Estado não persistente

Existem dois tipos de estado: estado de recurso e de aplicação. O estado de aplicação permanece no cliente até que ele realize alguma operação que chame algum recurso. Já o estado de recurso permanece no servidor e somente é enviado ao cliente sob a forma de representação (DAIPRAI, 2011, p.48 apud FLANDERS, 2008).

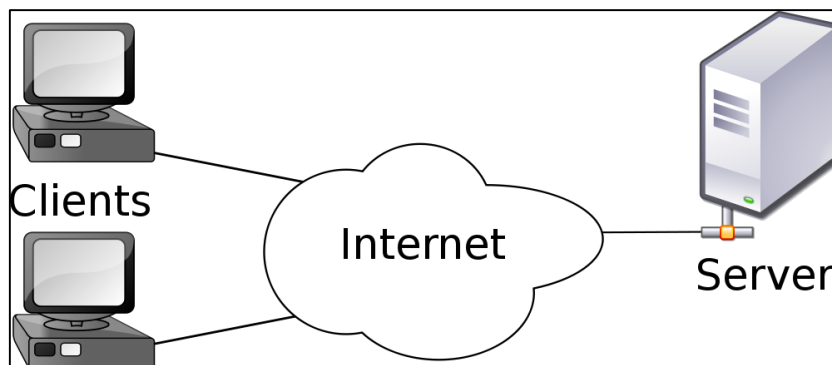
Conectividade

Conexão diz respeito a algo ligado a outro. Dessa forma, um recurso deve apontar para outros em sua representação (DAIPRAI, 2011, p.45 apud RICHARDSON; RUBY, 2007).

4.4.2.1 Restrições do REST

A primeira restrição para uma aplicação REST é a *cliente-server*. Nessa restrição são separadas as responsabilidades em dois ambientes diferentes, dessa forma o cliente não terá acesso ou não se preocupará com banco de dados, gerenciamento de *cache*, *log*, entre outros. Dessa mesma forma vale para o contrário, o servidor não se preocupa com a *interface*, experiência do usuário, entre outros. O servidor aguarda uma requisição do usuário, executa a ação e retorna uma resposta. A Figura 21 define como é a ligação entre o cliente e o servidor (PLANSKY, 2014).

Figura 21: ligação entre cliente e servidor.



Fonte: Plansky (2014).

A segunda restrição é *Stateless*. O *Stateless* define que uma aplicação pode mandar várias requisições para um servidor, mas cada uma dessas requisições deve ter as informações necessárias para que o servidor consiga processá-la da forma ideal. O servidor não deve guardar nada a respeito do cliente, todas as informações de estado deve permanecer no cliente (PLANSKY, 2014).

A terceira restrição é o *Cacheable*. Para evitar processamentos desnecessários, ocupando recursos do servidor, é preciso que as respostas de requisições iguais de vários clientes que acessam aos mesmo tempo um recurso, possam ser cacheadas, aumentando a performance da aplicação (PLANSKY, 2014).

Sempre que um cliente solicita um recurso para o servidor, ele processa e armazena essas informações em *cache*. Quando um outro cliente acessar o mesmo recurso, essas informações que estão em *cache* serão exibidas, não sendo necessário realizar outra requisição ao servidor. Para limpar os *caches* há duas maneiras, limpado sempre que houver mudança de estado do recurso ou em um determinado período de tempo (PLANSKY, 2014).

A quarta é a *Uniform Interface*, que é um contrato entre cliente e servidor (PLANSKY, 2014). Dentro do contrato existem regras, que são:

- Identificando o *resource* – cada recurso deve ter um identificador;
- Representação do *resource* – forma de como o recurso vai ser devolvido ao cliente;
- Resposta autoexplicativa – é necessária a passagem de meta informações na requisição e resposta, por exemplo código HTTP, *Host*, *Content-Type*, entre outros;
- *Hypermedia* – retorna as informações necessárias para que o usuário/cliente saiba navegar e para ter acesso a todos os recursos da aplicação.

A quinta é a *Layered System*. A *Layered System* tem como princípio que o cliente não deve chamar diretamente um servidor, deve-se passar antes para uma camada intermediadora, onde essa camada/*interface* fica responsável por fazer a comunicação com o servidor (PLANSKY, 2014).

E por último, sendo uma opção é a restrição *Code-On-Demand*. Essa condição permite que diferentes clientes possam se comportar de maneiras específicas utilizando os mesmos serviços (PLANSKY, 2014).

4.4.3 Tecnologia RESTfull

Para que uma aplicação seja considerada como RESTfull, ela deve seguir exatamente as regras que são definidas pela arquitetura REST e um nível de coesão e maturidade que são definidos na escala chamada Richardson Maturity Model, além de utilizar vários identificadores, verbos HTTP e diferentes formatos de retorno (PLANSKY, 2014).

Segundo Plansky (2014), esses níveis são:

- Nível 0 – ausência de qualquer regra, apenas utilizando o protocolo HTTP como transporte das operações no servidor;
- Nível 1 – aplicação de recursos;
- Nível 2 – implementação de verbos HTTP (GET, POST, PUT, DELETE) para diferentes tipos de operações;
- Nível 3 - a aplicação deve fornecer para o cliente toda a informação necessária para interagir com a aplicação.

4.4.4 Considerações finais sobre *Web Service*

Neste tópico foi visto que os *Web Services* são utilizados para realizar comunicação entre serviços e fazer integração de aplicações diferentes, dessa forma é possível que novas aplicações desenvolvidas possam interagir com outras já existentes e que os sistemas desenvolvidos em plataformas diferentes sejam compatíveis (OFICINADANET, 2015). Como

já abordado, existem duas formas de utilizar os *Web Services*: com componentes reutilizáveis e para fazer a conexão de *softwares* (W3SCHOOLS, 2015).

Vários *frameworks* oferecem componentes que auxiliam na utilização/desenvolvimento de *Web Services* de maneira simples. No próximo tópico será abordado o que são *frameworks*, para após apresentar o *Zend Framework 2*, que é desenvolvido pela linguagem de programação PHP e que oferece componentes para criação *Web Services*.

4.5 Frameworks

Um *framework* é o conjunto de códigos entre vários projetos, fornecendo funcionalidades genéricas. O *framework* pode atingir uma funcionalidade específica através da sua configuração durante a criação de uma aplicação (MÜLLER, 2008).

Segundo Sauv  (2015), para definir melhor o que   um *framework*, observa-se as seguintes ideias:

- Johnson-1: "Um *framework*   um conjunto de classes que incorpora um design abstrato de solu  es para uma fam lia de problemas relacionados";
- Johnson-2: " Um *framework*   um conjunto de objetos que colaboram para realizar um conjunto de responsabilidades para um dom nio do subsistema de aplica   o";
- Johnson-3: " Um *framework*   um conjunto de classes abstratas e a forma como os objetos dessas classes colaboram";
- Gama: " Um *framework*   um conjunto de classes que comp  em um projeto reutiliz vel para uma classe espec fica de software cooperando";
- Govoni: " Um *framework*   uma cole   o abstrata de classes, *interfaces* e padr  es dedicado a resolver uma classe de problemas atrav s de uma arquitetura flex vel e extens vel";
- Rogers: "A estrutura   uma biblioteca de classes que captura padr  es de intera   o entre objetos. A estrutura consiste de um conjunto de classes concretas e abstratas, explicitamente projetado para ser usado em conjunto".

Hoje existem v rios *frameworks* que podem auxiliar na programa   o, tanto para linguagens *back-end/server-side* quando para *front-end/client-side*, como por exemplo o *Zend Framework 2*, *AngularJS*, *Doctrine* (para persist ncia de dados), *Bootstrap* (para estilos CSS), *Hibernate* (persist ncia de dados) para o Java, entre outros.

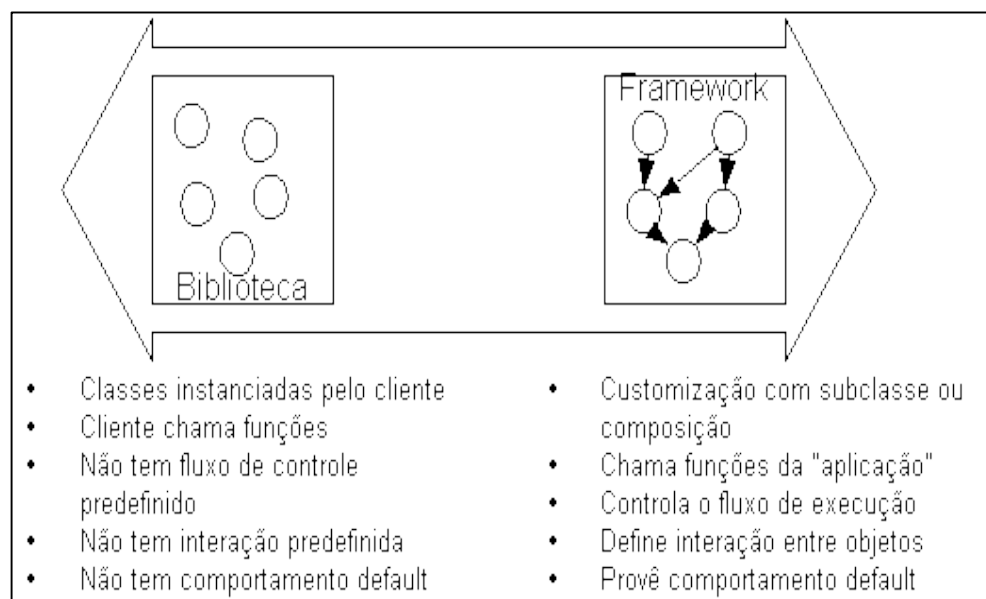
A utiliza   o de um *framework* por desenvolvedores se torna muito  til quando se quer reutilizar c digos, ou seja, utilizar um determinado componente mais que uma vez no projeto (M  LLER, 2008). Segundo M  ller (2008), um bom exemplo de reutiliza   o de c digo   um formul rio de *login*, pois o padr  o de *login* em todo lugar   o mesmo.

4.5.1 Diferenças entre *Framework* e uma Biblioteca de Classes OO

A principal diferença de um *framework* para uma Biblioteca de classes OO (Orientada a Objetos) é que na biblioteca de classes, cada classe é única, não depende das outras, conforme a Figura 22. Com a utilização de bibliotecas, as aplicações criam colaborações (SAUVÉ, 2015).

O *framework* já tem a comunicação entre objetos, dessa forma não precisa saber quando chamar cada método. O *framework* chama o código da aplicação que fica responsável por tratar cada particularidade (SAUVÉ, 2015).

Figura 22: diferenças entre um *framework* e uma Biblioteca de Classes OO.



Fonte: Sauvé (2015).

4.5.2 Diferenças entre *Frameworks* e *Design Patterns*

Segundo Sauvé (2015), as principais diferenças de um *Framework* para *design patterns* são:

- *framework* inclui códigos, os *designs patterns* só utilizam em exemplo de uso dos patterns;
- *framework* pode ser estudado a nível de código, executado e reusado;
- *frameworks* contém vários tipos de *designs patterns* que são bastante utilizados para documentar os *frameworks*, mas ao contrário nunca ocorre.

É evidente que ambos consistem em classes, *interfaces* e colaborações.

4.5.3 Características Básicas de *Frameworks*

Segundo Sauv  (2015), a principal caracter stica de um *framework*   ser reus vel, mas existem outras caracter sticas importantes que devem ser vistas, s o elas:

- deve ser bem documentado;
- deve ser f cil de utilizar;
- deve ser extens vel;
- deve ser de uso seguro;
- deve ser eficiente;
- deve ser completo.

Essas caracter sticas devem ser observadas na hora de utilizar um *framework* para auxiliar no desenvolvimento de uma aplica  o, pois sem essas garantias o desenvolvedor pode ter problemas para entender as funcionalidades, ter dificuldades de utilizar e n o ser eficiente da forma como se espera.

4.5.4 Vantagens em utilizar *Frameworks*

Com a utiliza  o de um *framework*, as vantagens e os benef cios s o claros quanto aos quesitos de redu  o de custo e redu  o do tempo de an lise do projeto e a disponibilidade de venda ou utiliza  o (SAUV , 2015).

Segundo Sauv  (2015), outras vantagens tamb m podem ser observadas, como:

- m xima reutiliza  o de c digos;
- desenvolvedores agregam valor ao *framework* com novas funcionalidades;
- estabilidade de c digo, dado pela grande utiliza  o em v rias aplica  es;
- melhor consist ncia e compatibilidade entre aplica  es;
- menos manuten  o.

Dentre essas vantagens, a utilização de um *framework* se torna muito importante quando se deseja ter qualidade, organização e fácil manutenção de código.

4.5.5 Desvantagens em utilizar *Frameworks*

As desvantagens em utilizar um *Framework* são poucas, tendo em vista todos os benefícios que ele traz. Algumas desvantagens segundo Sauv   (2015) s  o:

- construir um *framework*    complexo;
- benef  cios a longo prazo;
- precisa modificar o processo de desenvolvimento.

Hoje existem v  rios *frameworks* dispon  veis no mercado, dessa forma n  o h   a necessidade de construir um, depende da necessidade e a forma de utiliza  o de cada empresa.

4.5.6 *Frameworks* caseiros

Os *frameworks* caseiros s  o aqueles criados para atender os problemas de alguma empresa ou de uma pr  pria pessoa para resolver determinado problema (CRISTIAN, 2012).

Segundo Cristian (2012), se j   j   utilizado um *framework* caseiro, deve-se observar algumas quest  es importantes, como:

- se o *framework* possui documenta  o de todos os componentes e se todos os componentes s  o testados;
- se os desenvolvedores ter  o capacidade de desenvolver um projeto em tempo ideal;
- se s  o permanentemente feitas melhorias no *framework* acompanhando as inova  es da *internet*/mercado.

Segundo Luma (2013), existem alguns princ  pios de porque    desejado criar um *framework*, s  o eles:

- nenhum *framework* tem a funcionalidade que eu preciso;
- ter total controle sobre o c  digo;

- necessidade de algo mais simples;
- criar um *framework* por diversão e aprendizado.

Conforme Luma (2013), existem algumas questões que recomendam a não criação de *frameworks* caseiros, como:

- não ter tempo disponível para criar, pois para criar um *framework* requer muito tempo;
- para uma empresa ou uma pessoa que deseja criar um *software* é mais vantajoso utilizar um *framework* existente, pois o tempo para desenvolvimento será menor e com menos custo;
- maior produtividade por utilizar algo criado e com uma boa documentação.

Algumas observações devem ser levadas em conta antes de construir um *framework*, por exemplo, criá-lo é um trabalho complexo, requer tempo e equipe, a reutilização de código deve ser planejada e os benefícios são realizados em longo prazo (CRISTIAN, 2012).

4.5.7 Considerações finais sobre *Framework*

Neste tópico foi visto que Framework é um conjunto de códigos e funções que fornece funcionalidades genéricas para todos utilizarem, de modo à auxiliar no desenvolvimento de projetos e resolver problemas específicos.

Os *Frameworks* trazem grandes vantagens com sua utilização, mas deve-se observar a necessidade antes de optar por algum. Nesse contexto entram os *frameworks* caseiros, que muitos utilizam para resolver algum problema específico, mas há alguns cuidados a serem tomados para criação de *frameworks*. O ideal é utilizar um *framework* que tenha uma comunidade forte e que seja bem documentado para facilitar sua utilização e obter ganhos mais facilmente (CRISTIAN, 2012).

Dentre os vários *frameworks* para auxiliar no desenvolvimento de softwares, existem vários que podem ser utilizados no desenvolvimento *web*. No próximo capítulo será abordado o *framework* Zend Framework 2, que é utilizado juntamente com a linguagem de programação PHP, implementa os conceitos do MVC e possui componentes para a implementação de *Web Services*.

4.6 Zend Framework 2

O *Zend Framework 2* (ZF2), evolução do *Zend Framework 1*, é um *framework open source* e tem uma comunidade muito grande e com mais de 15 milhões de *downloads* (FRAMEWORK2, 2015).

O *Zend Framework* foi criado em 2005, nesse mesmo tempo *frameworks* como *Ruby on Rails*⁷ e *Spring*⁸ já estavam ganhando popularidade. Ele foi publicado pela primeira vez na *Zend Conference*. Os projetistas buscavam com o *Zend* combinar características e RAD (*Rapid Application Development*) (VASCONCELLOS et al., 2010).

O *Zend Framework 2* é um *framework* para ser utilizado em aplicações *web* e para serviços, utilizando a versão 5.3.x do PHP ou maior e utiliza 100% de código orientado a objetos. Além disso, oferece suporte a uma implementação poderosa e de alto desempenho com o *design patterns* MVC e ainda uma abstração de banco de dados de fácil utilização. Possui componentes para fornecer autenticação e autorização para os usuários, dando segurança a aplicação (FRAMEWORK2, 2015).

O principal patrocinador do ZF2 é a *Zend Technologies*, mas várias empresas contribuem para criar novos recursos. As empresas *Google*, *Microsoft* e *Strikelron* têm uma parceria com o *Zend* para fornecer *interfaces* e serviços *web* e outras tecnologias para colocar à disposição de usuários do ZF2 (FRAMEWORK2, 2015).

O ZF2 é implementado em PHP 5 e licenciado sob *Open Source Initiative* (OSI) *approved New BSD License* e todos os contribuintes com o ZF2 devem assinar um *Contributor License Agreement* (CLA), que é baseado no *Apache Software Foundation* 's CLA. Esse licenciamento e as políticas de contribuição foram criadas para anular questões de propriedade intelectual por usuários comerciais do *Zend Framework* (VASCONCELLOS et al., 2010).

Segundo Vasconcellos (et al., 2010), existem várias empresas que utilizam o *Zend Framework*, algumas dessas empresas que utilizam são:

- *Nokia*;
- *IBM*;
- *GNU/Linux Matters*;

⁷ *Ruby on Rails* – *Framework* de desenvolvimento web otimizado para a produtividade sustentável e a diversão do programador. *Ruby* é a linguagem de programação e o *Rails* é o *framework* (RAILS, 2015).

⁸ *Spring* – O *Framework Spring* fornece um modelo abrangente de programação e configuração de aplicativos corporativos baseados em Java modernos - em qualquer tipo de plataforma de implementação (SPRING, 2015).

- *Eurotransplant*;
- *Digital Sublimity*;
- *Magento*.

A patrocinadora corporativa do *Zend Framework* é a *Zend Technologies*, que é co-fundada pelos contribuintes do núcleo do PHP Andi Gutmans e Zeev Suraski e os parceiros são, IBM, *Google*, *Microsoft* e *StrikeIron*.

4.6.1 Características do *Zend Framework*

Foi visto anteriormente algumas características do *Zend Framework*. Conforme Vasconcellos (et al., 2010) existem outras que são importantes e que devem ser vistas, são elas:

- todos componentes são PHP 5 completamente Orientado a Objetos e tem conformidade com o *E Strict*;
- implementação MVC extensível;
- implementação flexível do *Table Gateway*, utilizado para acessar banco de dados relacional em um ambiente orientado a objetos;
- suporte para vários bancos de dados, por exemplo *MySQL*, *Oracle*, IBM DB2, *Microsoft SQL Server*, *PostgreSQL*, *SQLite* e *Informix Dynamic Server*;
- autenticação e autorização baseada em ACL (*Access Control Lists*);
- filtro de dados para segurança da aplicação e dos dados;
- gerenciamento de sessão;
- criação de formulário com PHP, arquivos de configuração ou por XML;
- componente nativo PHP para leitura, criação e atualização de documentos em PDF;
- serialização de estruturas de dados PHP para JSON (*JavaScript Object Notation*), para facilitar o desenvolvimento de AJAX⁹ (*Asynchronous JavaScript and XML*).

No PHP existem constantes pré-definidas e são definidas no núcleo do PHP. O *E Strict* é uma constante pré-definida que, segundo o site do PHP, manda notícias de erros em tempo de

⁹ AJAX - é a arte de trocar dados com um servidor, atualizando partes de uma página web - sem recarregar a página inteira (W3SCHOOLS, 2015).

execução e permite ao PHP sugerir mudanças no código, as quais irão assegurar melhor interoperabilidade e compatibilidade futura do código (PHP, 2015).

4.6.2 Conceitos do PHP utilizados no *Zend Framework*

Nesse tópico será visto os principais conceitos do PHP que o *Zend Framework 2* utiliza.

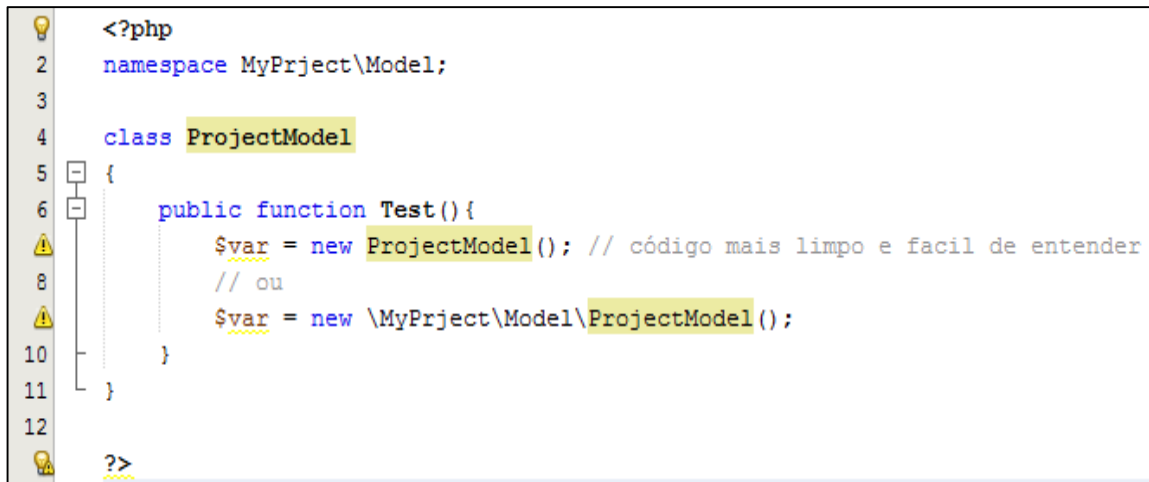
4.6.2.1 *Namespace*

Numa maneira mais ampla, *namespace* é uma forma de encapsular itens. Por exemplo: uma pasta/diretório dentro do sistema operacional serve para agrupar itens que são relacionados, esta pasta age como um nome geral para os itens que estão dentro da pasta (PHP, 2015).

Para o PHP (2015), os *namespace* servem para resolver dois problemas que os programadores encontram ao criar códigos reutilizáveis, esses problemas são:

- colisão de nomes criados no projeto, dessa forma é possível ter duas funções ou classes com mesmos nomes, mas cada arquivo deve ter *namespace* diferentes;
- capacidade de dar um apelido ao arquivo, dessa forma o código fica mais legível, por exemplo, não precisar colocar todo o nome do arquivo todas as vezes para chamar uma função ou classe.

Qualquer código PHP válido pode ser contido dentro de um *namespace*, mas os *namespace* só podem ser declarados em: classes, *interfaces*, funções, e constantes (PHP, 2015).

Figura 23: declaração de *namespace*.


```

1  <?php
2  namespace MyPrject\Model;
3
4  class ProjectModel
5  {
6      public function Test(){
7          $var = new ProjectModel(); // código mais limpo e facil de entender
8          // ou
9          $var = new \MyPrject\Model\ProjectModel();
10     }
11 }
12
13 ?>

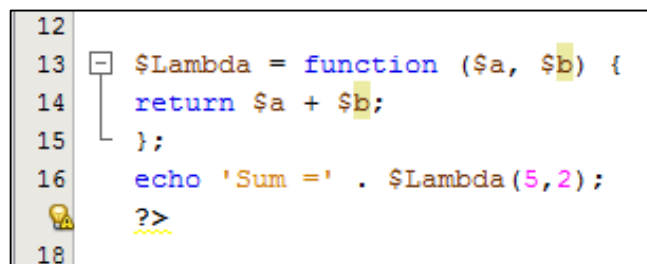
```

Fonte: elaborado pelo autor.

Conforme apresentado na Figura 23, para declarar um *namespace* basta utilizar a palavra-chave *namespace* antes do nome que se deseja utilizar. Devem ser declarados antes de qualquer código e só funciona com o PHP acima da versão 5.3.0.

4.6.2.2 *Lambda function*

Lambdas function são funções anônimas, ou seja, uma função sem nome que pode ser utilizada por uma variável e que pode ser passada como argumento para outras funções ou métodos (STAFF, 2014). Elas podem ser definidas em qualquer momento e normalmente estão associadas a uma variável. Dessa forma, uma função *lambda* só vai existir enquanto a variável existir, se a variável sair do escopo em que foi declarada acontece o mesmo com a função conforme mostra a Figura 24 (STAFF, 2014).

Figura 24: utilização de *lambda function*.


```

12
13 $Lambda = function ($a, $b) {
14     return $a + $b;
15 };
16 echo 'Sum =' . $Lambda(5,2);
17
18 ?>

```

Fonte: elaborado pelo autor.

As funções *lambda* são principalmente úteis para funções do PHP que aceitam uma função de retorno e podem ser utilizadas a partir da versão 5.3.0 do PHP (STAFF, 2014).

4.6.2.3 *Dependency injection*

Dependency injection (DI) e *DI containers* são diferentes, DI é um método utilizado para escrever um código melhor e o *DI container* é uma ferramenta para ajudar o DI (PHP-DI, 2015).

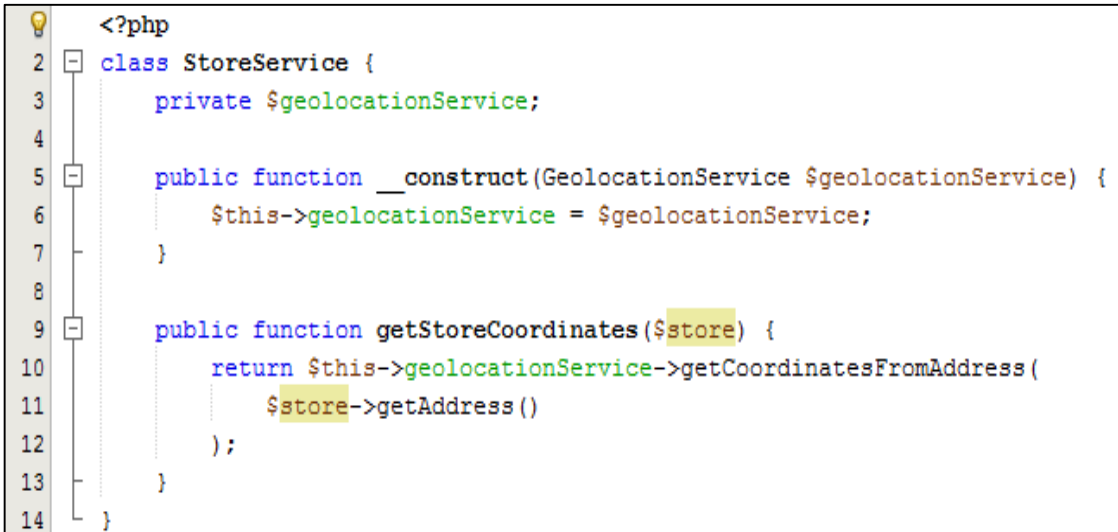
O *Dependency injection* permite padronizar e centralizar a maneira como os objetos são construídos em sua aplicação. Ao invés de criar um objeto dentro da classe, é injetado o objeto na classe através de um método construtor (FRAMEWORK, 2012).

Segundo PhalconPHP (2015) existem três modos de fazer a injeção de dependências:

- *Constructor Injection*: realizado através do método construtor;
- *Setter Injection*: utiliza o método *set* para injetar as dependências;
- *Properties Injection*: injetar dependências ou parâmetros diretamente em atributos públicos da classe.

A Figura 25 e 26 mostram um código com utilização de DI.

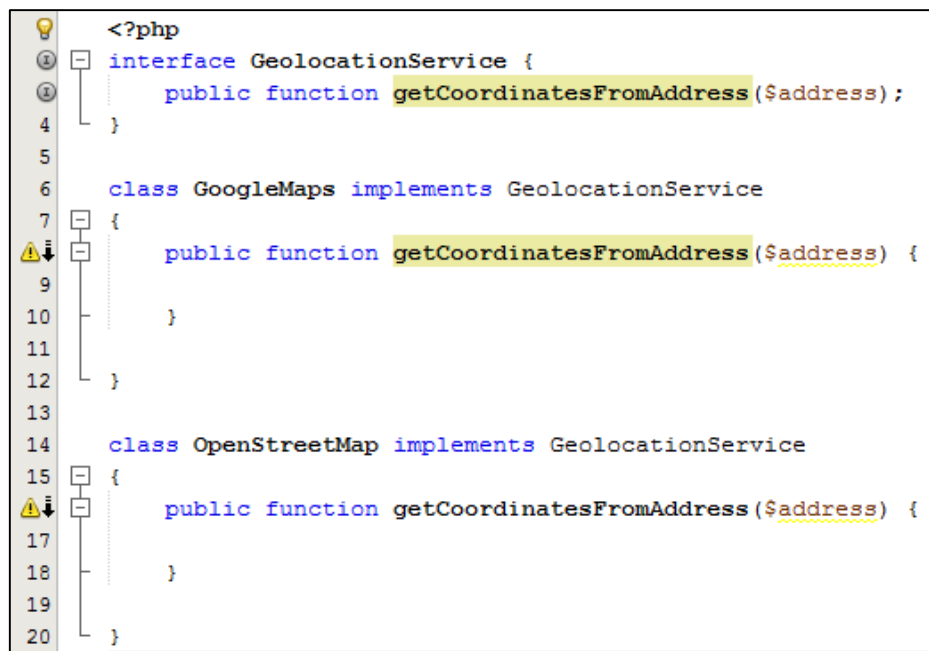
Figura 25: código com utilização de DI.



```
<?php
2 class StoreService {
3     private $geolocationService;
4
5     public function __construct(GeolocationService $geolocationService) {
6         $this->geolocationService = $geolocationService;
7     }
8
9     public function getStoreCoordinates($store) {
10         return $this->geolocationService->getCoordinatesFromAddress(
11             $store->getAddress()
12         );
13     }
14 }
```

Fonte: elaborado pelo autor.

Figura 26: código com utilização de DI (complementação).



```

<?php
interface GeolocationService {
    public function getCoordinatesFromAddress($address);
}

class GoogleMaps implements GeolocationService
{
    public function getCoordinatesFromAddress($address) {
    }
}

class OpenStreetMap implements GeolocationService
{
    public function getCoordinatesFromAddress($address) {
    }
}

```

Fonte: elaborado pelo autor.

Na Figura 26 é criada uma *interface* que fica responsável por criar a função *getCoordinatesFromAddress*. Também são criadas duas classes, *GoogleMaps* e *OpenStreetMap* que são implementadas pela *interface* e que contém a função da *interface*. Na Figura 25 é criada uma classe chamada *StoreService*, onde ela fica responsável por criar o método construtor da *interface*. Quando é chamado o método construtor *geolocationService*, ele entra primeiramente na *interface* e através dela é feita a chamada da função *getCoordinatesFromAddress* das classes que são implementadas pela *interface*, no caso as classes *GoogleMaps* e *OpenStreetMap* (PHP-DI, 2015).

4.6.2.4 Testes unitários

No ZF2 é utilizado o *PHPUnit* para a criação dos testes de unidades para garanti que a aplicação esteja funcionando como deve funcionar. Para compreender como funciona os testes unitários, será abordado detalhadamente este assunto neste tópico.

Com o crescimento e grande demanda de *softwares*, houve a necessidade de garantir sua melhor qualidade. Segundo Alexandre Bartié: “Qualidade de software é um processo sistemático que focaliza todas as etapas e artefatos produzidos, com o objetivo de garantir a conformidade de processos e produtos, prevenindo e eliminando defeitos” (CARDOSO, 2013).

Os testes unitários, conhecidos também por TDD (*Test Driven Development*), tem por objetivo fazer testes de ponta a ponta e pensar nos testes antes da implementação, garantindo dessa forma a qualidade no *software* (CARDOSO, 2013).

TDD é a parte central da *Extreme Programming* (XP), é também utilizado em outras metodologias e podendo ser usado livremente (CARDOSO, 2013).

Com a utilização de TDD, primeiro são criados os testes e depois é feita a implementação do sistema. Os testes são utilizados para dar melhor entendimento do projeto, pois como visto, os testes vem antes da implementação, dando uma ideia do projeto (ROCHA, 2015).

Cada componente do sistema é testado separadamente. Esses componentes podem ser classes de objetos, funções, entre outros. Mas como um sistema é um conjunto de entidades integradas, é importante testar a integração dessas entidades (ROCHA, 2015).

Segundo Alencar (2015), existem vários motivos para utilizar testes unitários em um projeto, alguns são:

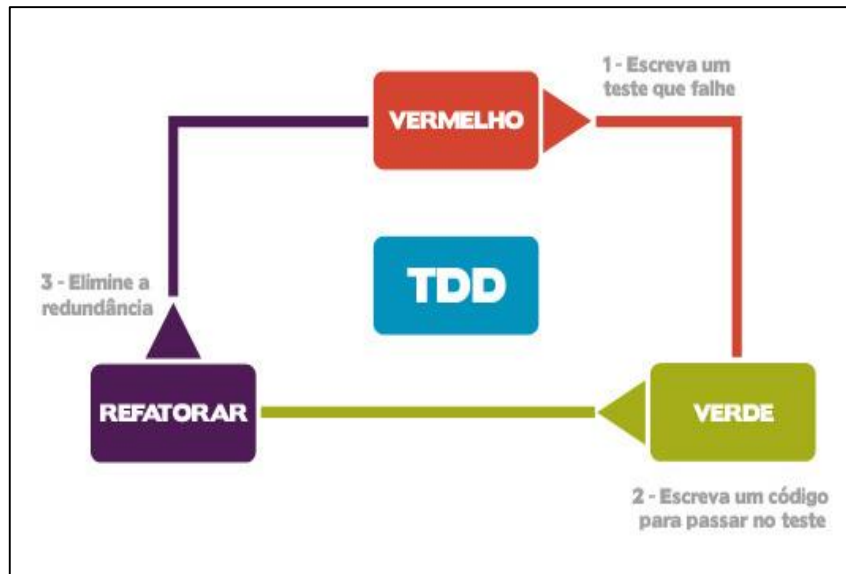
- *feedback* rápido sobre novas funcionalidades e sobre funcionalidades existentes;
- código mais limpo;
- segurança na correção de *bugs*;
- maior produtividade;
- código da aplicação mais flexível.

Segundo Rocha (2015), os benefícios em utilizar o TDD são:

- processo mais confiável;
- reduz custos;
- regras de negócio foram entendidas;
- evita retrabalho de equipe.

O ciclo do TDD é da seguinte forma: é criado um teste que irá falhar -> cria a codificação para passar no teste -> refatorar o código, como mostra a Figura 27 (ROCHA, 2015).

Figura 27: ciclo de vida do TDD.



Fonte: Rocha (2015).

Conforme Rocha (2015), os testes devem seguir o padrão chamado F.I.R.S.T:

- F (*Fast*) – os testes devem ser rápidos, pois testam apenas uma unidade;
- I (*Isolated*) – testes são isolados, testam as unidades individualmente sem a integração;
- R (*Repeatable*) – repetição dos testes;
- S (*Self-verifying*) – verificar se passou ou se deu falha na execução do teste;
- T (*Timely*) – teste deve ser oportuno, realizando um teste por unidade.

Para começar a utilizar os testes no desenvolvimento, pode ser mais fácil se for utilizando uma biblioteca *XUnit's*. Existem várias ferramentas que auxiliam e ajudam nos testes, são elas:

- *JUnit*: ferramenta de testes para Java, disponível para os mais diversos IDE'S como o Eclipse, Netbeans, etc.;
- *PHPUnit*: *framework* para testes unitários no PHP, possível integrar aos IDE'S;
- *PyUnit*: *framework* para testes unitários para a linguagem Python;
- *NUnit*: *framework* para testes na plataforma .NET;
- *SimpleTest*: outra ferramenta para testes no PHP.

Depois disso, é preciso iniciar a fase de desenvolvimento, criando os primeiros testes (ROCHA, 2015).

Com o ZF2 os testes são feitos através do *PHPUnit*, mas é necessário criar uma diretório para salvar os códigos de testes, depois é só criar os códigos de testes e executá-los.

4.6.2.5 Eventos

Eventos são extensões para agendar eficientemente I/O (entrada e saída), tempo e eventos usando o melhor mecanismo de notificação de I/O disponível para a plataforma desejada (PHP, 2015).

Os requisitos necessários para usar eventos são:

- Biblioteca *libevent*;
- Biblioteca *OpenSSL*, para recursos *OpenSSL*.

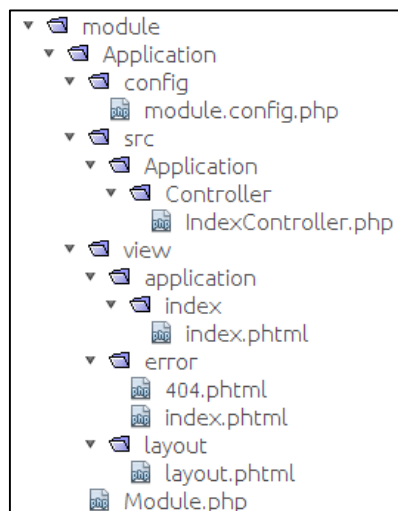
Um evento nunca vai ser acionado, a não ser que ele seja acionado através do método *Event::add()*. Um evento adicionado fica no estado pendente até que o evento ocorra, transformando-o em ativo. O método *Event::del()* deleta um evento (PHP, 2015).

4.6.2.6 Módulos

O ZF2 utiliza módulos para organizar códigos específicos da aplicação e é utilizado para fornecer controle de níveis da aplicação (*FRAMEWORK*, 2015).

No exemplo abaixo, quando é criado um novo modulo, por exemplo, módulo *Application*, ele fica da seguinte forma como mostra a Figura 28:

Figura 28: estrutura de um Módulo do ZF2.



Fonte: elaborado pelo autor.

Observando a Figura 28 acima, o modulo *Application* tem vários diretórios separados para adicionar cada tipo de arquivo em seu devido lugar. Os arquivos PHP que contém classes estão dentro do diretório *src/Application*. O diretório *view* contém um sub-módulo que contém *scripts* para mostrar os dados na página *web*. As classes dentro de um determinado módulo terão o *namespace* do nome do módulo, que é o nome do diretório do módulo. No mesmo nível da pasta *Controller* é possível criar a pasta *Form*, é onde são criados os formulários da aplicação e também a pasta *Model*, na qual são criadas as classes de objetos para o banco de dados, a pasta *Controller* é onde ficam os controladores da aplicação e a pasta *config* é onde são feitas as configurações da aplicação (FRAMEWORK, 2015).

4.6.2.7 *Table gateway*

O objeto *table gateway* (*Zend\Db\TableGateway*) fornece um objeto que representa uma tabela em um banco de dados e os métodos dos objetos espelham as operações mais comuns em uma tabela de banco de dados (FRAMEWORK, 2015).

Segundo *Framework* (2015), alguns recursos internos do *Zend\Db*:

- *GlobalAdapterFeature* – capacidade de utilizar um adaptador global ou estático sem a necessidade de injetá-lo em um *TableGateway*;
- *MasterSlaveFeature* – capacidade de utilizar um adaptador mestre para *insert()*, *update()* e *delete()*;
- *MetadataFeature* – capacidade de preencher *TableGateway* com informações da coluna de um objeto de metadados;
- *EventFeature* - capacidade de utilizar um objeto *TableGateway* com *Zend \EventManager*;
- *RowGatewayFeature* – habilidade de usar o *select()* para retornar um objeto.

Existem duas formas de implementar a interface do *Table Gateway*: *AbstractTableGateway* e *TableGateway*. O *AbstractTableGateway* fornece funções básicas para *select()*, *insert()*, *update()*, *delete()*. O *Table gateway* simplesmente adiciona um construtor ao *AbstractTableGateway* (FRAMEWORK, 2015).

4.6.3 REST com ZF2

O *AbstractRestfulController* é um controlador abstrato que o ZF2 fornece (*Zend \ Mvc \ Controlador \ AbstractRestfulController*). Esse controlador fornece implementação nativa para RESTfull, onde mapeia os métodos de solicitação do HTTP para o *controller* (*FRAMEWORK*, 2015).

Segundo o site do *Zend Framework*, o *AbstractRestfulController* utiliza a seguinte matriz:

- GET: mapeado pelo comando *get()*, é utilizado para pegar os dados, passando por parâmetro algum “id” (identificador), ou para ler dados passando algum parâmetro;
- POST: mapeado pelo comando *create()*. Esse método espera um argumento *\$data*. Utilizado para criar uma nova entidade e a resposta deve ser um *status* HTTP 201;
- PUT: mapeado pelo comando *update()*. Esse método é utilizado para atualizar uma entidade e se for bem sucedida retornará um *status* 200 ou 202;
- DELETE: mapeado pelo comando *delete()*. Esse método é utilizado para excluir uma entidade passando por parâmetro um “id” e se for bem sucedido deve retornar um *status* 200 ou 204.

O *AbstractRestfulController* implementa as seguintes interfaces:

- *Zend\Stdlib\DispatchableInterface*;
- *Zend\Mvc\InjectApplicationEventInterface*;
- *Zend\ServiceManager\ServiceLocatorAwareInterface*;
- *Zend\EventManager\EventManagerAwareInterface*.

Além disso, com esse controlador é possível mapear os métodos “*action*”. Esses métodos serão seguidos do sufixo “*Action*”, por exemplo, *indexAction()*. Isso permite executar ações como fornecer formulários utilizados para enviar diversos métodos RESTfull, ou para adicionar métodos RPC para uma API *RESTFull* (*FRAMEWORK*, 2015).

4.6.4 Considerações finais sobre o ZF2

O *Zend Framework* é empregado junto com a linguagem de programação PHP e bastante utilizado para o desenvolvimento de aplicações *web*. Fornece estrutura para o padrão MVC, dessa forma é possível ter uma aplicação bem organizada, padronizada, facilitando assim a manutenção do código. Possui também funções que facilitam a integração com banco de dados, geração de arquivos PDF, Excel, entre outros formatos (VASCONCELLOS et al., 2010).

O ZF2 incorpora vários conceitos do PHP dentro de sua estrutura, já que o mesmo foi implementado em PHP. Alguns desses conceitos são necessários para utilizar com o PHP, como o *namespace*.

Através do componente *AbstractRestfulController*, é possível utilizar os conceitos de *Web Service*, realizando a comunicação dos dados utilizando os métodos do protocolo HTTP.

As linguagens de programação *front-end* também têm à disposição *frameworks* para auxiliar no desenvolvimento. Antes de abordar algum *framework cliente-side*, é preciso saber sobre a linguagem de programação *front-end* que será utilizada no projeto, portanto no próximo tópico será abordado sobre a linguagem de programação *JavaScript*.

4.7 *JavaScript*

O *JavaScript* é uma linguagem de programação que foi desenvolvida originalmente por Brendan Eich da *Netscape* com o nome de Mocha, posteriormente foi mudado para *LiveScript* e atualmente *JavaScript*. *LiveScript* foi lançada pela primeira vez na versão beta do navegador NetScape 2.0, em setembro de 1995 (STENZEL, 2013).

A mudança de nome para *JavaScript* se deu na época em que a NetScape adicionou suporte à tecnologia Java em seu navegador. Com essa escolha final, causou confusão dando impressão de que a linguagem foi baseada em Java, mas na verdade essa escolha foi caracterizada por muitos como uma estratégia de *marketing* da *NetScape* para aproveitar a fama do recém-lançado Java (STENZEL, 2013).

“*JavaScript* foi a primeira linguagem a ser tratada como uma extensão da HTML e que trouxe capacidade de processamento *client-side*” (CARVALHO, 2001, p. 2).

No começo muitos desacreditaram na linguagem, pois ela tinha como principal alvo o público leigo, mas com o passar dos tempos o *JavaScript* tem se transformado na linguagem de programação mais popular da *web* (STENZEL, 2013).

Com a chegada do *AJAX*, o *JavaScript* teve sua popularidade aumentada, recebendo mais atenção dos profissionais. Com esse grande aumento, houve grande proliferação de bibliotecas e *frameworks* e o aumento no uso do *JavaScript* fora do ambiente de navegadores bem como o uso de plataformas de *JavaScript server-side* (STENZEL, 2013).

As principais características do *JavaScript* são: ter tipagem dinâmica, funções internas, dentre outras que será abordado no próximo tópico.

Segundo Carvalho (2001, p. 2), algumas possibilidades que o *JavaScript* traz, que outras linguagens como o HTML e CSS trazem, não ultrapassam as do *JavaScript*:

- efetuar contas matemáticas;
- gerar páginas com aparências que podem ser definidas em tempo de execução;
- tratar determinados eventos, podendo interagir com o usuário através desse tratamento;
- manipular janelas dos navegadores;
- modificar propriedades da página dinamicamente.

JavaScript é a linguagem mais utilizada pelos desenvolvedores da *Web* e sua entrada no mercado foi levada em consideração pela ECMA (*European Computer Manufactures Association*), uma entidade normativa europeia (CARVALHO, 2001, p. 2).

4.7.1 Características do *JavaScript*

O *JavaScript* possui programação imperativa, que é por sequência de ordens, chamada de comandos ou instruções (ALBUQUERQUE, 2010) e a estruturada, que é um método para combinar as estruturas de controle formadas por blocos de maneira organizada e que traduza o problema a ser resolvido (SATO, 2009).

Possui tipagem dinâmica, ou seja, não tem declaração de tipo de dados. É baseada em objetos, que no *JavaScript* são *arrays* associativos. *JavaScript* inclui uma função chamada *eval*, que consegue executar em tempo de execução comandos da linguagem. O *JavaScript* tem funções ‘internas’ ou ‘aninhadas’ que são funções definidas dentro de outras funções. *JavaScript* permite que funções aninhadas sejam criadas com o escopo estático, no momento de sua definição e possui o operador “()” para invocá-las em outro momento, ou seja, se no código-fonte uma estrutura está aninhada em outra, a de dentro pode acessar variáveis na de fora (PRADO, 2014).

Talvez o principal aspecto seja que o *JavaScript* é *case-sensitive*, ou seja, existe diferença entre letras maiúsculas e minúsculas. Dessa forma, “document” é diferente de “Document”, que para o HTML isso não ocorre (CARVALHO, 2001, p. 16).

Outra característica que foi herdado do JAVA, é a utilização de ponto e vírgula no final de cada comando, por exemplo: *var numero = 1234;*. Apesar de ter herdado, não é uma regra da linguagem, mas sua utilização pode evitar problemas e erros na execução de *scripts* (CARVALHO, 2001, p. 16).

Segundo Silva (2010, p. 43), comentários também estão à disposição dos desenvolvedores. Existem três tipos de comentários: comentários de uma linha (variante 1), comentário em linha única (variante 2) e os comentários de múltiplas linhas.

As Figuras 29, 30 e 31 abaixo, mostram exemplos de utilização de comentários.

Figura 29: exemplos de comentários linha única (variante 1).

```

1 <script type="text/javascript">
2     //caixas de dialogo JavaScript
3     alert("Olá Mundo"); // caixa de alerta
4     confirm("Você tem certeza?"); // caixa de confirmação
5     prompt("Entre seu CPF"); // caixa de dados
6 </script>

```

Fonte: Silva (2010, p. 43).

Figura 30: exemplos de comentários linha única (variante 2).

```

8 <script type="text/javascript">
9     <!-- caixas de dialogo JavaScript
10    alert("Olá Mundo"); <!-- caixa de alerta
11    confirm("Você tem certeza?"); <!-- caixa de confirmação
12    prompt("Entre seu CPF"); <!-- caixa de dados
13 </script>

```

Fonte: Silva (2010, p. 43).

Figura 31: exemplos de comentários múltiplas linhas.

```

15 <script type="text/javascript">
16     /* caixas de dialogo JavaScript
17     caixa de alerta
18     caixa de confirmação
19     caixa de dados */
20     alert("Olá Mundo");
21     confirm("Você tem certeza?");
22     prompt("Entre seu CPF");
23 </script>

```

Fonte: Silva (2010, p. 43).

Use-se duas barras “//” para iniciar um comentário em linha única ou é colocado “/* */” para múltiplas linhas (SILVA, 2010, p. 43).

A variante 2, para comentários de uma linha, usa uma sintaxe semelhante a do HTML, com a diferença de que não é necessário seu fechamento, como mostra a Figura 32.

Figura 32: exemplo de comentário para marcação HTML e de *JavaScript*.

```

25 <!-- Aqui comentário para a marcação HTML-->
26 <!-- Aqui comentário para a marcação JavaScript

```

Fonte: Silva (2010, p. 43).

É utilizado a variante 1 para os comentários de uma linha (SILVA, 2010, p. 43).

4.7.2 Declarações de variáveis no *JavaScript*

Cada uma das instruções de um *script* constitui uma declaração independente. Para separar uma declaração de outra existe duas formas. Conforme as Figura 33, é possível separar por ponto e vírgula e em linhas diferentes (SILVA, 2010, p. 44).

Figura 33: formas de declaração de variáveis.

```
//declarações na mesma linha
a = 5; b = 8; alert("Alô Mundo");

// declarações em linhas separadas
a = 5
b = 8
alert("Alô Mundo");
```

Fonte: Silva (2010, p. 44).

Conforme abordado anteriormente, o ponto e vírgula não é obrigatório, mas é recomendada a utilização para não haver erros. Para o caso de declaração na mesma linha, a utilização de ponto e vírgula se faz necessária.

4.7.3 Literais do *JavaScript*

Segundo Silva (2010, p. 45), a palavra literal para o *JavaScript* quer dizer qualquer dado que não seja variável, ou seja, que tenha seu valor fixo.

No exemplo da Figura 34, o valor cinquenta (50) e Alô Mundo, são literais.

Figura 34: exemplo de dados literais.

```
3 a = 50;
4 mensagem = "Alô Mundo";
5
```

Fonte: Silva (2010, p. 45).

Existem seis tipos de dados literais no *JavaScript*, são eles:

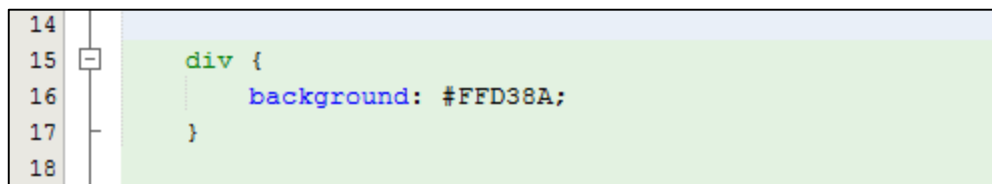
- Inteiros;

- Decimais;
- Booleanos;
- *Strings*;
- *Arrays*;
- Objetos.

Para a sintaxe do *JavaScript*, os inteiros são números em base decimal (base dez), hexadecimais (base dezesseis) ou octal (base oito).

A base dez é a mais conhecida, usada na matemática no conjunto \mathbb{Z} que os alunos aprendem no ensino médio, por exemplo: $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$, ou seja, conjunto dos números inteiros positivos e negativos incluindo o número zero (SILVA, 2010 p. 45). A base hexadecimal é muito utilizada no CSS, nas declarações de cores, conforme mostra a Figura 35. Seus símbolos são de zero (0) a nove (9) juntamente com as seis primeiras letras do alfabeto, de A à F.

Figura 35: exemplo de hexadecimal no CSS.



```

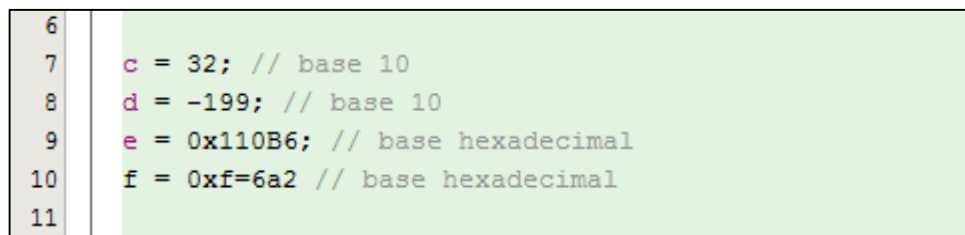
14
15
16     div {
17         background: #FFD38A;
18     }

```

Fonte: Silva (2010, p. 46).

Para utilização de hexadecimal nos arquivos *JavaScript* é necessário utilizar antes o 0x (zero xis), conforme mostra a Figura 36.

Figura 36: exemplo de declarações usando literal inteiro.



```

6
7     c = 32; // base 10
8     d = -199; // base 10
9     e = 0x110B6; // base hexadecimal
10    f = 0xf=6a2 // base hexadecimal
11

```

Fonte: Silva (2010, p. 46).

Nota-se que para a sintaxe na base hexadecimal, não é necessária a utilização de letras somente maiúsculas ou minúsculas (SILVA, 2010, p. 46).

Os literais decimais são os números constituídos por um inteiro e um número fracionário, separados por um ponto (.), por exemplo a Figura 37.

Figura 37: literais decimais no *JavaScript*.

```

13
14  a = 3.1416;
15  b = -76.89;
16  c = .33333;

```

Fonte: Silva (2010, p. 46).

O *JavaScript* suporta notação científica, para escrever tanto literais inteiros como fracionários. As notações científicas são maneiras de representar números compostos por muitos algarismos e consiste em acrescentar a letra E ou e a um número para indicar expoentes de 10 (SILVA, 2010, p. 46).

Os booleanos são representados no *JavaScript* pelas palavras-chave *true* e *false*, definindo condições verdadeiras ou falsas, respectivamente (SILVA, 2010, p. 497). A figura 38 mostra como o *JavaScript* trata os booleanos.

Figura 38: booleanos no *JavaScript*.

```

19
20  a = true;
21  b = false;
22

```

Fonte: Silva (2010, p. 47).

Os booleanos são muito utilizados para controle e para testar a validade de uma determinada condição. Dessa forma, permite ao *script* tomar uma decisão baseando na condição de verdadeiro ou falso retornada pela condição feita pelo desenvolvedor (SILVA, 2010, p. 43).

Observa-se abaixo na Figura 39, o uso de booleano.

Figura 39: utilização de *booleano*.

```

a = 10;
se a < 10 faça isto se não faça aquilo;

```

Fonte: Silva (2010, p. 47).

Transformando essa lógica em sintaxe *JavaScript*, ela traz uma condição que retornara *true* caso seja menor que 10, e *false* caso contrário.

Strings são sequências de caracteres, já para o *JavaScript* são um conjunto de zero ou mais caracteres entre aspas duplas ("") ou simples (' ') (SILVA, 2010, p. 47).

Na Figura 40 abaixo, pode-se observar como o *JavaScript* trata as *Strings*.

Figura 40: *strings* no *JavaScript*.

```

30
31 nome = "Maurício Samy Silva"; // string com aspas duplas
32 outroNome = "Pedro Nascimento Júnior"; // string com aspas simples
33

```

Fonte: Silva (2010, p. 47).

Pode-se também utilizar caracteres especiais para formatar uma *string*, como mostra a Figura 41.

Figura 41: formatação de *Strings* utilizando caracteres especiais no *JavaScript*.

```

35
36 mensagem = "Obrigado pela visita.\n Volte em breve.";
37

```

Fonte: Silva (2010, p. 47).

Ao inserir o caractere especial `\n` na *string*, ele pula (quebra) uma linha onde estiver localizado, dando o resultado conforme mostra a Figura 42.

Figura 42: resultado da utilização do `\n`.

```

42
43 Obrigado pela visita.
44 Volte em breve.
45

```

Fonte: Silva (2010, p. 47).

Na Tabela 2 abaixo, observe alguns caracteres especiais da linguagem e a sintaxe geral para caracteres Latin-1 e Unicode (bibliotecas de caracteres).

Tabela 2: caracteres especiais *JavaScript*.

Caractere	Descrição – Caractere Unicode hexadecimal
<code>\b</code>	<i>Backspace</i> - <code>\u0008</code>
<code>\f</code>	<i>Form feed</i> - <code>\u000C</code>
<code>\n</code>	Nova linha - <code>\u000A</code>
<code>\r</code>	Retorno do carro - <code>\u000D</code>
<code>\v</code>	Tabulação vertical - <code>\u000B</code>
<code>\t</code>	Tabulação horizontal - <code>\u0009</code>
<code>\'</code>	Apostofo ou aspas simples - <code>\u0027</code>
<code>\"</code>	Aspas duplas - <code>\u0022</code>

\\	Barra invertida - \u005C
\XXX	Caractere Latin-1 expresso por dígitos octais de 1 a 377
\xXX	Caractere Latin-1 expresso por dois dígitos hexadecimais de 00 a FF
\uXXXX	Caractere Unicode expresso por quatro dígitos hexadecimais

Fonte: Silva (2010, p. 48).

Nas três últimas linhas consta a sintaxe geral para representação de caracteres em *string* em hexadecimal e octal. O sistema octal está em desuso, então deve ser evitado (SILVA, 2010, p. 47).

Os *arrays* são conjuntos de zero ou mais valores, que são separados por vírgula e estão dentro de colchetes ([]). Todo *array*, sem definir sua posição começa com índice zero, depois um, assim sucessivamente (SILVA, 2010, p. 51). Abaixo na Figura 43, é mostrado como declarar um *array* no *JavaScript*.

Figura 43: declaração de *array JavaScript*.

```
frutas = ["laranja", "pera", "goiaba", "morango"];
```

Fonte: Silva (2010, p. 51).

Abaixo, na Figura 44, é mostrado um exemplo de como pegar um valor do *array*, que consiste em pegar o nome do *array*, seguido de um índice entre colchetes (SILVA, 2010, p. 51).

Figura 44: pegar o valor em uma posição do *array*.

```
frutas[0] // contém o valor laranja
frutas[3] // contém o valor morango
```

Fonte: Silva (2010, p. 51).

Além disso, um *array* pode conter vários tipos de dados, expressões, e outros *arrays*, como mostra a Figura 45 abaixo.

Figura 45: sintaxe de um *array* misto.

```
a = 4;
b = 12;
ArrayMisto = ["laranja", 34, "casa", true, [1,2,3], a + b];
```

Fonte: Silva (2010, p. 51).

A Figura 46 mostra como fazer a extração dos dados do *array*.

Figura 46: pegar o valor em um *array* misto.

```
ArrayMisto[0] //Contém o valor laranja
ArrayMisto[1] //Contém o valor 34
ArrayMisto[3] //Contém o valor true
ArrayMisto[4][1] //Contém o valor 1
ArrayMisto[5] //Contém o valor 16
```

Fonte: Silva (2010, p. 51).

E finalmente os objetos, que para o *JavaScript* são conjuntos de pares nome/valor separador por vírgula e envolvidos por chaves (`{ }`). A relação de nome/valor pode ser da forma, objeto e seu valor, um método do objeto e seu valor e mesmo outro objeto. Para criar um objeto, utiliza-se seguinte sintaxe, conforme Figura 47 (SILVA, 2010, p. 52).

Figura 47: sintaxe para criação de um objeto.

```
23 Carro =
24 {
25     marca: "Renault",
26     modelo: "Logan",
27     ipva: "valor('rb15')",
28     dimensoes :
29     {
30         c: "4.250",
31         l: "1.735",
32         h: "1.525"
33     }
34 };
```

Fonte: Silva (2010, p. 52).

O objeto que foi criado chamado de Carro, possui três propriedades, a marca, o modelo e a dimensão. A propriedade dimensão é um outro objeto, que existem mais três propriedades, que são *c*, *l* e *h*, que definem o comprimento, a largura e a altura do carro, e também tem um método com nome de “ipva”, que é uma função chamada *valor()*, que aceita um argumento de cálculo *rb15*, com intenção de calcular o IPVA do carro (SILVA, 2010, p. 52).

Para percorrer um objeto, é utilizado a seguinte sintaxe: *propriedade.valor*, conforme mostra a Figura 48 a seguir.

Figura 48: sintaxe para percorrer um objeto.

```

41 <script type="text/javascript">
42
43 Carro =
44 {
45     marca: "Renault",
46     modelo: "Logan",
47     ipva: "valor('rb15')",
48     dimensoes :
49     {
50         c: "4.250",
51         l: "1.735",
52         h: "1.525"
53     }
54 };
55 marca = Carro.marca;
56 modelo = Carro.modelo;
57 comprimento = Carro.dimensoes.c;
58 largura = Carro.dimensoes.l;
59 altura = Carro.dimensoes.h;
60
61 </script>

```

Fonte: Silva (2010, p. 52).

As palavras reservadas do *JavaScript*, segundo SILVA (2010, p. 56), são conforme mostra a Tabela 3.

Tabela 3: palavras-chave do *JavaScript*.

<i>break</i>	<i>else</i>	<i>new</i>	<i>var</i>
<i>case</i>	<i>finally</i>	<i>return</i>	<i>void</i>
<i>catch</i>	<i>for</i>	<i>switch</i>	<i>while</i>
<i>continue</i>	<i>function</i>	<i>this</i>	<i>with</i>
<i>default</i>	<i>if</i>	<i>throw</i>	<i>delete</i>
<i>in</i>	<i>try</i>	<i>do</i>	<i>instanceof</i>
<i>typeof</i>			

Fonte: Silva (2010, p. 56).

4.7.4 Considerações finais sobre o *JavaScript*

O *JavaScript* é uma linguagem de programação utilizada para o desenvolvimento *front-end* da aplicação. Possui algumas características iguais a do PHP, como por exemplo a declaração de variáveis, que não é preciso declara o seu tipo para utilizá-la (PRADO, 2014). O

JavaScript é baseado em funções, que maioria das vezes são chamadas através de eventos ou entre outras funções.

Para trabalhar junto com o *JavaScript* existem vários *frameworks*. Alguns desses *frameworks front-end* aplicam os conceitos de MVC, melhorando a organização do código, estruturando e facilitando a manutenção. O próximo capítulo tratará sobre o AngularJS, que é um *framework* para a linguagem *JavaScript* que possui o *design pattern* MVC.

4.8 AngularJS

AngularJS é um *framework open source* (código aberto) para a linguagem *JavaScript*, no qual se pode criar incríveis aplicativos *web* baseado em AJAX. Geralmente, a complexidade envolvida na construção de aplicativos *AJAX* de grande porte é muito grande. O AngularJS visa minimizar essa complexidade oferecendo um grande ambiente para o desenvolvimento, bem como os meios para testar seus aplicativos (PANDA, 2014, p. xvii).

O AngularJS foi desenvolvido por Miško Hevery e Adam Abrons, em 2009, para promover alta produtividade e experiência na programação *web* (BRANAS, 2014, pg. 7).

O AngularJS implementa os conceitos do *design patterns* MVC, que é uns dos seus recursos mais poderosos. Outro recurso interessante é que ele amplia o vocabulário HTML para torna-lo adequado para a construção de uma aplicação *web*, criando-os de forma declarativa, deixando seu código limpo e muito legível (PANDA, 2014, p. xvii).

AngularJS é um *framework* MVW (*Model-View-Whatever*), onde *Whatever* significa “o que quer que funciona para você”. A razão é que pode ser usado o *framework* AngularJS tanto como *Model-View-Controller* (MVC) e *Model-View-View-Model* (MVVM). Mas o que é importante para os desenvolvedores é poder construir aplicações *web* sólidas com grande estrutura e design com o mínimo esforço (PANDA, 2014, p. 1).

O desempenho do AngularJS é bom com interação com outras bibliotecas. Pode-se utilizar o jQuery e o AngularJS juntos para criar uma ótima aplicação *web* (PANDA, 2014, p. xvii).

AngularJS favorece muito o *Test Driven Development* e possui grande apoio tanto para unidade e teste de ponta-a-ponta (PANDA, 2014, p. xvii).

Todos sabem que o HTML é ótimo para a criação de documentos estáticos, mas para o desenvolvimento de uma aplicação *web* dinâmica, HTML *standalone* (independente) não é realmente a melhor ferramenta. Este é o lugar onde AngularJS entra. AngularJS é o que HTML teria sido se ele foi projetado para a construção de aplicações *web* (PANDA, 2014, p. 1).

AngularJS é uma estrutura em que pode se estender o HTML, ensinando-lhe novas sintaxes, tornando aplicável para o desenvolvimento de grandes aplicações *web*. Com AngularJS pode-se introduzir novos elementos HTML e atributos personalizados que carregam um significado especial, dependendo da sua necessidade.

Por exemplo, pode-se criar um novo elemento HTML `<datepicker />` e adicionar um *widget* para selecionar uma data ou um elemento `<drop-zone />`, que pode fazer ações de arrastar e soltar (PANDA, 2014, p. 1).

4.8.1 As características do AngularJS

Neste tópico serão abordadas as características mais interessantes que o AngularJS possui, que segundo Panda (2014, p. 2, 3, 4 e 5) são:

- Dados mágicos de duas vias de ligação: é provavelmente o recurso mais legal e o mais útil que o AngularJS possui, ou seja, de modo simplificado a ligação de dados é a sincronização automática de dados entre o ponto de vista (HTML) e o modelo (variáveis *JavaScript*). Com o AngularJS pode-se criar modelos e ligar diferentes componentes com modelos específicos. Dessa forma, sempre que o valor do modelo muda, a visão é atualizada automaticamente, e sempre que o valor de todas as mudanças dos componentes da visão, por exemplo, o valor de entrada de texto, o modelo vinculado também será atualizado. Assim, pode-se realizar operações no modelo, mudar seus valores, e AngularJS garante que a visão será atualizada para refletir as alterações;
- Estrutura código *front-end*: sem uma estrutura para o desenvolvimento sua vida só vai ficar mais difícil quando o aplicativo torna-se complexo. Também não deixa margem para um bom teste. Mas se utilizar o AngularJS, poderá construir aplicações sólidas, bem estruturadas, e totalmente testáveis. Dessa forma, a manutenção do seu código será mais simples e deixa a vida mais fácil;
- Suporte Encaminhamento: os aplicativos de única página ou *Single Page Apps* (SPAs) estão em toda parte hoje em dia. Não é bom redirecionar o usuário para uma nova página toda vez que ele clicar em algo. Ao contrário disso, é ideal carregar o conteúdo de forma assíncrona na mesma página e apenas mudar a URL no navegador. Com a utilização do AngularJS, poderão ser criados SPAs muito facilmente e com esforço mínimo. AngularJS irá então carregar a visualização adequada na página principal quando uma URL específica é requisitada;

- *Templating* bem feito com HTML: o AngularJS usa HTML simples antigo como uma linguagem de modelagem, dessa forma o fluxo de trabalho torna-se muito mais simples. Os *designers* podem criar *interfaces* de usuário (UIs) de forma habitual, e os desenvolvedores podem usar sintaxe de vinculação declarativa para amarrar diferentes componentes de *interface* com os modelos de dados muito facilmente;
- Experiência de usuário aprimorada com a validação do formulário: os formulários são a parte mais importante para um CRUD (*Create, Read, Update, Delete*). O AngularJS possuiu formas de incorporar validações em tempo real, validadores customizados, formatadores e muito mais. Ele também oferece várias classes CSS que indicam a forma como os controles de formulário estão, mais importante ainda se são válidos ou inválidos;
- Ensinar as diretivas (instruções) com nova sintaxe HTML: A diretiva em AngularJS é o que engana HTML em fazer as coisas novas que não são suportados nativamente. É feito através da introdução de novos elementos/atributos e ensinando a nova sintaxe para HTML. Com as diretivas você pode criar um novo elemento chamado `<data-picker />` e usá-lo em HTML. Esse conceito de diretivas é exclusivo do AngularJS e é muito bom para desenvolvimento de UI;
- Incorporável, testável, e injetável: com o AngularJS pode-se incorporar um aplicativo AngularJS dentro de outro, além de poder utilizar outras tecnologias junto com ele. AngularJS favorece TDD e também oferece suporte para o *Dependency Injection*, dessa forma o aplicativo pode ser mais testável e sustentável;
- Fornecido pelo Google e uma comunidade ativa: sempre que for adotar ou utilizar uma nova tecnologia, é ideal procurar um bom apoio nas comunidades. AngularJS é mantido pelos desenvolvedores do Google. Como o AngularJS é *open source*, pode-se baixar o código fonte e adicionar novas funções, dessa forma deixando o *framework* com novas soluções para diversos problemas. Possuiu uma ótima documentação, facilitando muito seu uso. O AngularJS continua melhorando muito em cada versão, com isso o número de desenvolvedores que utilizam o *framework* também está aumentando.

4.8.2 Ferramentas necessárias para utilizar o AngularJS

Conforme Panda (2014, p. 8), para escrever bons *softwares*, é preciso ter as ferramentas certas. Para o AngularJS existem várias ferramentas e editores disponíveis. Algumas opções são:

- *Jetbrains WebStorm2*: IDE comercial da Jetbrains, oferece um bom suporte para AngularJS;
- *Sublime Text 2 & 3*: um bom editor com suporte a AngularJS;
- *NetBeans 5*: um editor de código aberto que também é uma boa opção para o desenvolvimento AngularJS.

Para depurar aplicativos com o AngularJS foi lançado um *plug-in* do navegador Chrome chamado *Batarang*. Aborda quaisquer gargalos de desempenho que pode haver. É uma ferramenta muito boa e é recomendável sua instalação (PANDA, 2014, p. 9).

4.8.3 Esqueleto da aplicação do AngularJS

No *Zend Framework 2*, todos os arquivos estão separados na estrutura para melhor organização utilizando um “esqueleto” de aplicação. Da mesma forma o Angular divide seus arquivos, através de um “esqueleto” proporcionando uma estrutura fixa, oferece um ambiente pré-configurado de teste, entre outras funcionalidades. Esse esqueleto pode ser baixado através do link do *Seed* (Semente/Esqueleto) Angular¹⁰ (PANDA, 2014, p. 9).

Dentro desse esqueleto há uma pasta chamada “js” que contém todos os arquivos *JavaScript* do projeto. Existem cinco arquivos .js que representam módulos diferentes. Por exemplo, sempre que você criar um controlador novo, ele deve ir para a pasta *controllers.js* (PANDA, 2014, p. 10).

Os outros componentes do diretório app são:

- *directives.js*;
- *filters.js*;

¹⁰ *Seed Angular* - <https://github.com/angular/angularseed/tree/69c9416c8407fd5806aab3c63916dfcf0522ecbc>

- *services.js*.

Ao abrir o esqueleto pode-se observar várias pastas. Por exemplo, todos os arquivos de CSS irão na pasta “css”, todas as imagens na pasta “img”, na pasta “*partials*” é onde são colocados os arquivos HTML para a estrutura da página e “lib/angular” é a biblioteca do AngularJS para recursos, animações entre outros (PANDA, 2014, p. 10).

4.8.4 A Anatomia de um aplicativo AngularJS

Segundo Panda (2014, p. 10 e 11), antes de começar a construir uma aplicação com o AngularJS, é preciso saber alguns componentes. Alguns componentes importantes que se deve saber são:

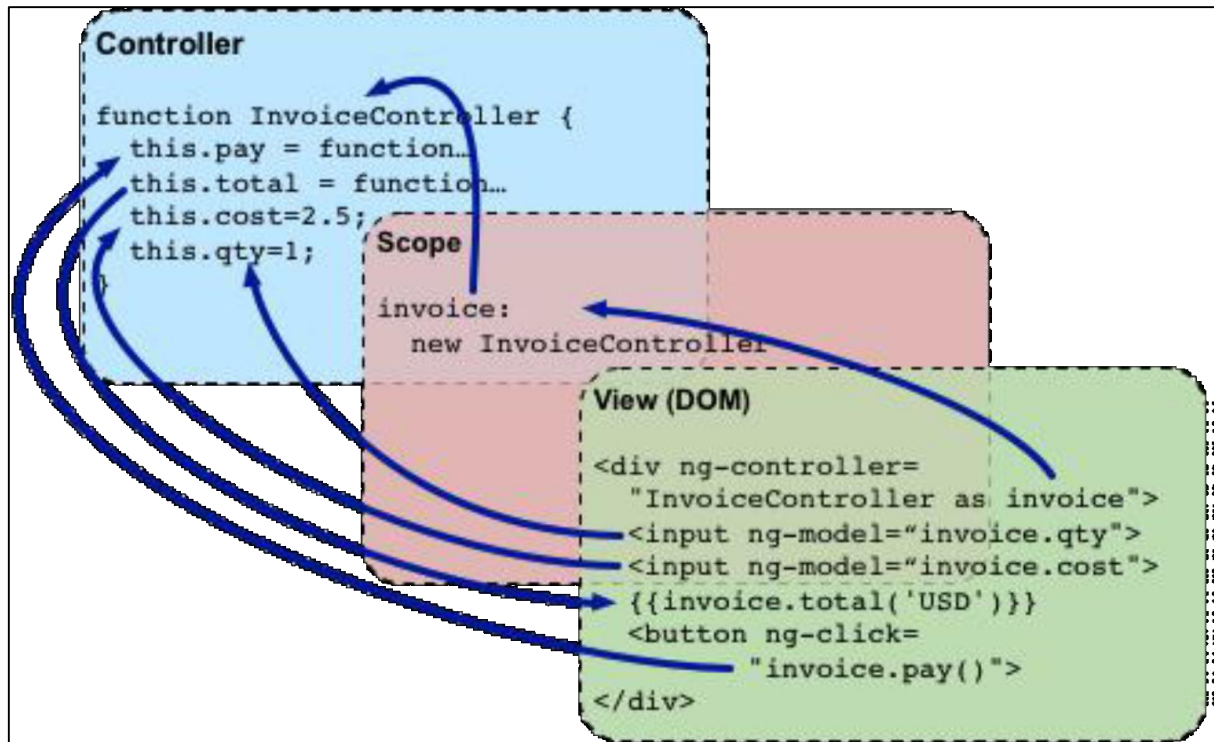
- *Model*: são os dados que serão mostrados para os usuários;
- *View*: é tudo o que o usuário vê quando carrega a página;
- *Controller*: é a lógica do negócio que conduz a aplicação;
- *Scope*: um contexto que contém os modelos de dados e funções. Um controlador normalmente define esses modelos e funções no escopo;
- *Directives*: ensina novas sintaxes HTML. Estende o HTML com elementos personalizados e atributos;
- *Expressions*: são representadas por `{{ }}` no HTML. São usadas para acessar modelos e funções do *scope*;
- *Template*: é o HTML com marcações na forma de diretivas (`<drop-zone/>`) e de expressões `{{ }}`.

Quando criar uma pequena aplicação, para ver os resultados é preciso de um servidor *web*. Para executar o servidor que vem junto com o esqueleto do Angular é preciso utilizar o Node.js¹¹ (PANDA, 2014, p. 11).

A Figura 49 apresenta a interação entre os AngularJS e os componentes da arquitetura MVC:

¹¹ Node.js – É um *framework* dirigido para eventos assíncronos, foi projetado para construir aplicações de rede escaláveis (NODEJS, 2015).

Figura 49: diagrama de interação entre o AngularJS com os componentes da arquitetura.



Fonte: Official documentation (www.angularjs.org)

4.8.5 MVW Angular

Códigos *front-end* são tão importantes quando os *back-end*. Para deixar a aplicação sustentável e bem estruturada é preciso seguir um padrão. Com o AngularJS pode-se utilizar tanto o padrão MVC quando o MVVM. Esta foi a razão para ele ser declarado como um *framework* MVW (*Model-View-Whatever*). Será apresentado agora cada um dos padrões (PANDA, 2014, p. 13).

MVC

É utilizado nas aplicações para dividir as funcionalidades em camadas, essas camadas são conhecidas como *Model*, *View* e *Controller* e cada uma delas com suas características básicas como visto no tópico 4.2, promovendo um código organizado. Essa divisão é realizada para facilitar o entendimento de um determinado problema.

MVVM

Model-View-ViewModel (MVVM) é um *design patterns* para construir *interfaces* declarativas de usuário. No MVVM, a *ViewModel* expõe os dados de negócios do seu aplicativo para a *View* de forma a *View* pode compreendê-los. Por exemplo, ao criar um editor de notas, a *ViewModel* deve manter uma lista de notas e expor métodos para adicionar, editar e apagar (PANDA, 2014, p. 13).

4.8.6 Estruturando o código com MVC

Para o MVC, tudo tem seu lugar. O principal benefício é que toda a lógica do negócio não está ligada na *view*. Outro benefício é a manutenção rápida e fácil e pode-se estruturar o código sem módulos (PANDA, 2014, p. 14).

Para o Angular os módulos são classificados dessa maneira:

- *Controller*: o controlador manipula entradas e chama o código que realiza as regras de negócio. Essa lógica dentro do Angular é executada dentro do serviço e injetada no *controller*. Pode-se mudar toda a *interface* do usuário a partir de uma visão da *Web* para exibição móvel e a lógica de negócios será o mesmo porque o controlador é completamente separado da visão; (PANDA, 2014, p. 14).
- *View*: no AngularJS a *view* lê os dados do escopo que já foi definido pelo controlador e os exibe; (PANDA, 2014, p. 14).
- *Model*: o modelo representa os dados de negócio que conduz a *interface*. A *interface* do usuário é uma projeção dos dados do modelo em qualquer momento através da *view*; (PANDA, 2014, p. 14).

Com essa estrutura, o código *JavaScript* fica mais entendível e mais limpo, facilitando a manutenção.

4.8.7 Directives do AngularJS

Por padrão, um *framework* traz um conjunto básico de *directives*, de como interagir com *arrays*, executar comportamentos quando um elemento é clicado e vários outros (BRANAS, 2014, pg. 19).

Segundo o Branas (2014, pg. 19 até pg. 31), o AngularJS tem vários tipos de *directives*, que são:

- A *directive ngApp*: é a primeira *directive* que é preciso entender porque define a raiz de uma aplicação AngularJS. Aplicado a um dos elementos, em geral HTML ou órgão, esta *directive* é usada para iniciar o *Framework*. No entanto, é recomendável fornecer um nome do módulo, definindo a entrada ponto do aplicativo no qual outros componentes, como controladores, serviços, filtros e diretivas podem ser vinculados;
- A *directive ngController*: pode-se atribuir qualquer controlador para *view* usando o *ngController*. Depois de utilizar a *directive*, a *view* e *controller* podem começar a compartilhar o mesmo escopo e estão prontos para trabalhar em conjunto;
- A *directive ngBind*: a *directive ngBind* é geralmente aplicada a um elemento de extensão e substitui o conteúdo do elemento com os resultados da expressão fornecida;
- A *directive ngBindHtml*: às vezes, pode ser necessário ligar uma sequência de HTML puro. Neste caso, a *directive ngBindHtml* pode ser usada da mesma maneira como *ngBind*, no entanto, a única diferença é que o conteúdo não fuja do escopo. Para utilizar esta *directive*, é necessário utilizar a dependência *angular-sanitize.js*. Ele traz a *directive ngBindHtml* e protege o aplicativo contra ataques comuns;
- A *directive ngRepeat*: a *directive ngRepeat* é realmente útil para iteração sobre matrizes e objetos. Pode ser usado com qualquer tipo de elemento, como as linhas de uma tabela, os elementos de uma lista, e até mesmo as opções de selecionar dados em um determinado campo;
- A *directive ngModel*: a *directive ngModel* atribui o elemento a uma propriedade no escopo, ligando, assim, a *view* com o *model*;

- A *directive ngClick*: a *directive ngClick* é um dos tipos mais úteis de *directive* do *framework*, pois permite vincular qualquer comportamento personalizado para o evento *click* do elemento;
- A *directive ngDisable*: pode desativar elementos com base no valor booleano de uma expressão;
- A *directive ngClass*: a *directive ngClass* é usada toda vez que é necessário aplicar dinamicamente uma classe para um elemento, fornecendo o nome da classe em uma expressão de *data-binding* (vinculação de dados);
- A *directive ngOptions*: a *directive ngRepeat* pode ser usado para criar as opções de um elemento *select*, no entanto, há uma *directive* muito mais recomenda que deve ser usado para este propósito, a *directive ngOptions*. Através de uma expressão, é necessário indicar o escopo da propriedade do qual a *directive* irá interagir, o nome da variável temporária que irá armazenar o conteúdo de iteração de cada loop, e a propriedade da variável que deve ser exibido;
- A *directive ngStyle*: a *directive ngStyle* é usada para suprir a demanda de configuração estilo dinâmico que segue o mesmo conceito usado com a *directive ngClass*, no entanto, com essa *directive* pode-se usar diretamente as propriedades de estilo e seus valores;
- A *directive ngShow and ngHide*: a *directive ngShow* altera a visibilidade de um elemento com base na sua propriedade *display*. Dependendo da implementação que se deseja fazer, é possível usar o *ngHide*, que tem o mesmo conceito da *ngShow*;
- A *directive ngIf*: a *directive ngIf* pode ser usada da mesma maneira como a *directive ngShow*, no entanto, enquanto a *directive ngShow* apenas lida com a visibilidade do elemento, o *ngIf directive* impede a representação de um elemento no *template*;
- A *directive ngInclude*: a AngularJS fornece uma maneira de incluir outros fragmentos de HTML externo nas páginas. A *directive ngInclude* permite a fragmentação e a reutilização do *layout* na aplicação;
- *oauth-ng*: uma *directive* para utilização do OAuth (MODULES, 2015);
- *restangular*: é um serviço do AngularJS que simplifica pedidos comuns de *GET*, *POST*, *DELETE* e *UPDATE* com um mínimo de código no lado cliente. É um

ajuste perfeito para qualquer aplicação *web* que consome dados de uma API *RESTFull* (DIRECTIV, 2015).

Essas *directives* são utilizadas para facilitar o desenvolvimento de uma aplicação com o AngularJS.

4.8.8 Considerações finais do AngularJS

O AngularJS, como visto, é um *framework* para a linguagem de programação *JavaScript*, o mesmo trabalha com os conceitos do *design pattern* MVC.

O AngularJS também é conhecido como modelo *Model-View-Whatever*, já que o mesmo pode ser utilizado com o padrão *Model-View-Controller* e também com o padrão *Model-View-ViewModel* (PANDA, 2014, p. 13).

O AngularJS será utilizado no projeto para manipular os dados da API *RESTFull* que será criada com o *Zend Framework 2*. Ao manipular os dados por *Web Service*, o *framework* poderá organizar os dados retornados e realizar a exibição dos mesmos no *design patterns* MVC no *cliente-side*.

A transição de dados entre *server-side* e *Client-side* deve ocorrer de forma segura e confiável. Para realizar a transação de dados o usuário deve solicitar ou enviar algum recurso, esses recursos devem ser controlados através de permissões, onde cada usuário terá permissões diferente de outros. Para garantir essa segurança e oferecer permissões, existe o OAuth, um *Framework*/protocolo que será abordado no próximo tópico.

4.9 OAuth

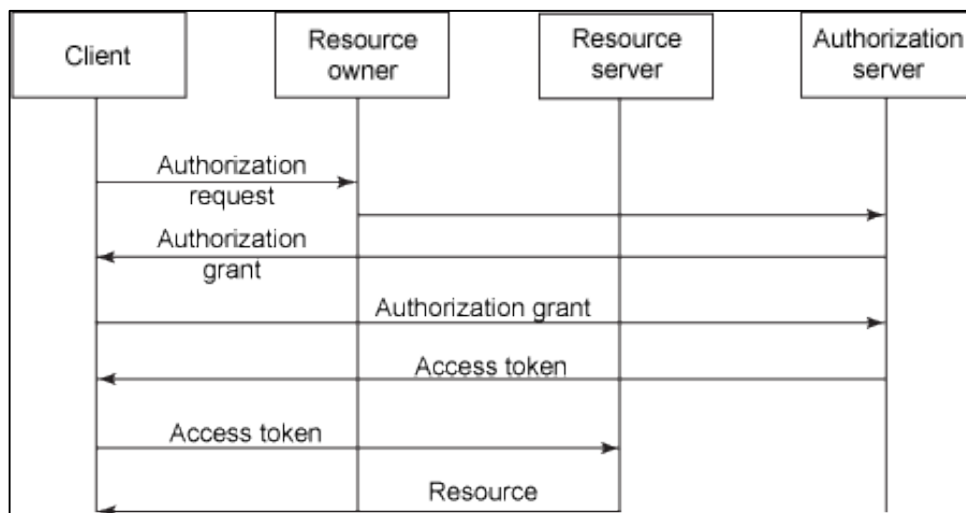
O OAuth é um *framework* poderoso e flexível que pode ser usado na proteção de API *RESTFull*. Com o OAuth é possível realizar interação entre componentes de forma segura, fornecendo um serviço de autorização, emitindo *tokens* de acesso específicos para terceiros (DOLPHINE, 2014).

Segundo Rasmussen (2012), existem alguns fluxos básicos para a obtenção de acesso à algum recurso, são eles:

- O proprietário do recurso: é uma entidade, capaz de conceder acesso a recursos protegidos;
- Servidor de recurso: é um processo, tipicamente um servidor HTTP, que é capaz de aceitar e responder a solicitação de recurso e resolver *tokens* de acesso;
- Serviço de autorização: é um processo, tipicamente um *Secure Token Service* (STS), capaz de autenticar um proprietário do recurso e emitir *tokens* de acesso;
- Cliente: realiza uma solicitação para acesso de um recurso.

A Figura 50 mostra um fluxo de autorização.

Figura 50: fluxo de protocolo de concessão de autorização do OAuth tradicional.



Fonte: Rasmussen (2012).

Nesse fluxo o cliente solicita acesso a um recurso para o proprietário do recurso, o qual responde com uma concessão de autorização que define privilégios específicos de acesso. A concessão autorizada pelo proprietário é enviada ao servidor de autorização, que responde com

um *token* de acesso que vai ser utilizado para acessar o recurso através do servidor de recurso (RASMUSSEN, 2012).

Conforme Rasmussen (2012), o OAuth define dois tipos gerais de cliente, são eles:

- Confidencial: são os clientes capazes de manter confidencialidade de suas credenciais;
- Pública: são os clientes que não são capazes de manter confidencialidade de suas credenciais.

Os *tokens* fornecem o direito de acesso, como foi visto anteriormente, mas também fornece restrições. Segundo Rasmussen (2012), os *tokens* podem ter formatos diferentes, como:

- *Token* de acesso: consiste em dados opacos. Para a transmissão do token é necessário utilizar protocolos seguros, como o HTTPS;
- *Token* MAC: usa uma chave MAC para assinar as seções de solicitação para verificar a integridade;
- *Tokens* atualizados: que podem ser emitidos durante a geração do token de acesso. Podem ser utilizados pelo cliente para obter tokens de acesso adicionais quando os tokens de acesso existentes tiverem expirado ou para ter permissões mais limitadas.

Esses tokens normalmente são uma cadeia de caracteres que pode assumir o formato de um cursor para recuperar o token de acesso real ou para fornecer informações autocontidas (RASMUSSEN, 2012).

4.9.1 Considerações finais sobre o OAuth

O OAuth é um *framework* utilizado para dar permissões de acesso a determinados recursos. O acesso ao recurso é solicitado ao proprietário que responde sobre os privilégios do acesso ao recurso. Essa resposta é enviada para o servidor de autorização que devolve como resposta um token para acessar o recurso através do servidor do recurso (RASMUSSEN, 2012).

Antes de aplicar esses conceitos é preciso verificar um problema que eles possam ser utilizados. Para esse projeto, os conceitos vão ser aplicados na distribuição de dados do Sistema de Gestão de Espaço da Unochapecó.

4.10 Considerações Finais

Através da pesquisa realizada para a elaboração da Revisão Bibliográfica, pode-se concluir que é possível criar uma aplicação que utilize o *design pattern* MVC tanto no *back-end*, através da linguagem PHP juntamente com o ZF2, quando no *front-end*, através da linguagem *JavaScript* juntamente com o AngularJS.

Como levantado na revisão, tanto o *Zend Framework 2* e quanto o AngularJS possuem componentes próprios para criação e manipulação dos dados de um *Web Service*, além disso, o AngularJS possui uma diretiva para implementar a técnica de segurança OAuth para garantir a proteção dos dados nas transições.

Para a execução do projeto será desenvolvido um *Web Service* RESTFull em um cenário real, onde existe um sistema que possui dados que necessitam ser distribuídos para outros sistemas.

A Unochapecó possuiu o Sistema de Gestão de Espaço que foi desenvolvido com o *Zend Framework 1* em 2011 pelo Centro de Residência em Software para controlar os espaços da instituição. Os dados que estão contidos nele necessitam ser distribuídos para outros sistemas, como o Sistema de Alocação de Espaços que precisa utilizar os dados de todos os espaços da instituição que estão contidos no Sistema de Gestão de Espaço. Através da revisão bibliográfica, e deste cenário, o projeto será executado com a finalidade de atingir os objetivos propostos.

5 PROCEDIMENTOS METODOLÓGICOS

Para alcançar os objetivos, esse projeto se caracteriza como pesquisa exploratória, pois envolve levantamento bibliográfico e também será necessário seguir os seguintes passos:

- Preparar o ambiente de trabalho, instalando e configurando o banco de dados e a IDE Netbeans, instalar a linguagem PHP na versão superior da 5, que contém um servidor *web* embutido;
- Preparar o ambiente de desenvolvimento instalando e configurando o *Zend Framework* 2 e o AngularJS;
- Modelar a aplicação;
- Implementar a aplicação e aplicar os conceitos do OAuth;
- Analisar se houve melhoria na organização do código, e manutenção.

Esse projeto é de natureza aplicada, pois o objetivo foi gerar conhecimentos para desenvolver uma aplicação prática dirigida à solução de um problema específico, nesse caso a distribuição de dados do Sistema de Gestão de Espaços da Unochapecó.

A abordagem do problema é da forma qualitativa, pois não requer o uso de métodos e técnicas estatísticas, como percentagem, média, moda, mediana, desvio-padrão. O ambiente é a fonte direta para coleta de dados e também é descritiva, porque o processo e seu significado são os focos principais de abordagem.

O procedimento técnico utilizado no projeto é a pesquisa bibliográfica, porque o projeto foi criado através de materiais já publicados, constituído principalmente de livros, artigos de periódicos e atualmente com material disponibilizado na *Internet*.

6 CRONOGRAMA

Cronograma das atividades para realização do Monografia I e Monografia II, conforme Tabela 4.

Tabela 4: cronograma das atividades.

Atividades	Mar.	Abr.	Mai.	Jun.	Jul.	Ago.	Set.	Out.	Nov.	Dez.
Levantamento do problema	X									
Estudo do tema	X									
Desenvolvimento dos Objetivos	X	X								
Questões de pesquisa		X								
Justificativa		X	X							
Levantamento e realização da revisão bibliográfica		X	X							
Referências	X	X	X							
Procedimentos metodológicos			X							
Cronograma		X	X							
Orçamento			X							
Entrega				X						
Defesa				X						
Correções				X	X					
Resumo e Abstract					X					
Modelagem do sistema						X				
Configuração do ambiente de trabalho						X				
Desenvolvimento e Aplicação dos conceitos						X	X	X	X	X

Conclusão									X	X
Entrega monografia										X
Defesa										X

Fonte: elaborado pelo autor.

7 ORÇAMENTO

Para realização desse projeto não houve gastos com qualquer tipo de bem material ou não material.

8 REFERÊNCIAS

ALBUQUERQUE, Toni. **Paradigmas de programação**, 2010. Disponível em: < <http://www.profissionaisti.com.br/2010/10/paradigmas-de-programacao/> >. Data de acesso: 18 mar. 2015.

ALENCAR, Roberto Luiz Sena de. **Test-Driven Development (TDD)**. Disponível em: < <http://www.devmedia.com.br/test-driven-development-tdd/10025#ixzz3Wx2f37sI> >. Acesso em: 12 abr. 2015.

BASTOS, Daniel Flores. **O que é Model-view-controller (MVC)?** 2011. Disponível em: < http://www.oficinadanet.com.br/artigo/desenvolvimento/o_que_e_model-view-controller_mvc >. Data de acesso: 23 mar. 2015.

BRANAS, Rodrigo. **AngularJS Essentials: Design and construct reusable, maintainable, and modular web applications with AngularJS**. Birmingham: Packt Publishing Ltd, 2014. 180 p.

CAKEPHP. **Visão Geral do CakePHP: Entendendo o Model-View-Controller**. Disponível em: < <http://book.cakephp.org/2.0/pt/cakephp-overview/understanding-model-view-controller.html#a-camada-controller> >. Acesso em: 14 maio 2015.

CARDOSO, André. **TDD, por que usar?** 2013. Disponível em: < <http://tableless.com.br/tdd-por-que-usar/> >. Acesso em: 05 maio 2015.

CARVALHO, Alan. **JavaScript**. Rio de Janeiro: Book Express Ltda., 2001. 228 p.

CHASE, Nicholas. **Entendendo a Zend Framework, Parte 1: O Básico**, 2006. Disponível em: < <http://www.ibm.com/developerworks/br/library/os-php-Zend1/> >. Data de acesso: 24 mar. 2015.

DAIPRAI, Willian Bonho. **Análise Comparativa Entre Os Serviços Web Baseados Em Rest E Soap Utilizando A Plataforma Microsoft .Net**. 2011. 92 f. TCC (Graduação) - Curso de

Sistemas de Informação, Universidade Comunitária da Região de Chapecó - Unochapecó, Chapecó - Sc, 2011. Disponível em: <<http://www5.unochapeco.edu.br/pergamum/biblioteca/php/imagens/000089/000089F8.pdf>>. Acesso em: 02 maio 2015.

DIRECTIV. **Restangular**. Disponível em: <<http://www.directiv.es/restangular/readme>>. Acesso em: 28 maio 2015.

DOLPHINE, Tiago. **OAuth2 - uma abordagem para segurança de aplicações e APIs REST. 2014**. Disponível em: <<http://www.infoq.com/br/presentations/oauth2-uma-bordagem-para-seguranca>>. Acesso em: 11 maio 2015.

FRAMEWORK, Master *Zend*. **Zend Framework 2 Conceitos Básicos - Dependency Injection**. 2012. Disponível em: <<http://www.masterZendFramework.com/articles-2/Zend-Framework-2-core-concepts-understanding-dependency-injection>>. Acesso em: 11 maio 2015.

FRAMEWORKS, Php Mvc. **MVC Frameworks Written in PHP**. 2014. Disponível em: <http://www.phpwact.org/php/mvc_frameworks>. Acesso em: 28 maio 2015.

FRAMEWORK, *Zend*. **Zend \ Db: Zend \ Db \ TableGateway**. Disponível em: <<http://Framework.Zend.com/manual/current/en/modules/Zend.db.table-gateway.html>>. Acesso em: 05 maio 2015.

FRAMEWORK, *Zf - Zend*. **Available Controllers**. Disponível em: <<http://Framework.Zend.com/manual/current/en/modules/Zend.mvc.controllers.html>>. Acesso em: 06 maio 2015.

FRAMEWORK, *Zf - Zend*. **Módulos**. Disponível em: <<http://Framework.Zend.com/manual/current/en/user-guide/modules.html>>. Acesso em: 06 maio 2015.

FRAMEWORK2, *Zend*. **About**. Disponível em: <<http://Framework.Zend.com/about/>>. Acesso em: 28 abr. 2015.

GAMA, Alexandre. **Introdução: JSON**. Disponível em: < <http://www.devmedia.com.br/introducao-json/23166> >. Acesso em: 21 maio 2015.

HUMANITY, Ieee Advancing Technology For. **IEEE Mission & Vision**. Disponível em: <https://www.ieee.org/about/vision_mission.html?WT.mc_id=lp_ab_mav>. Acesso em: 28 maio 2015.

HUMBERTO, Caio. **Design Patterns – Introdução – Parte 1**, 2010. Disponível em: < <http://www.devmedia.com.br/design-patterns-introducao-parte-1/16780> >. Data de acesso: 26 mar. 2015.

JONATHAN, Miguel. **Introdução ao padrão de projeto MVC – Model-View-Controller**. 2011. Disponível em: < <http://www.dcc.ufrj.br/~comp2/TextosJava/MVCIntroducao.pdf> >. Acesso em: 14 maio 2015.

JUNQUEIRA, Alvaro R. B., COSTA, André Fernandes, LIRA, Édson Carlos. **Design Patterns: Conceitos e Aplicações**, 1998. Disponível em: < http://www.dcc.ufrj.br/~schneide/PSI_981/gp_6/design_patterns.html >. Data de acesso: 26 mar. 2015.

LAMIM, Jonathan. **Afinal, o que é Frontend e o que é Backend?** 2014. Disponível em: < <http://www.oficinadanet.com.br/post/13541-afinal-o-que-e-frontend-e-o-que-e-backend-> >. Acesso em: 15 abr. 2015.

LIMA, Jean Carlos Rosário. **WEB SERVICES (SOAP X REST)**. 2012. 41 f. TCC (Graduação) - Curso de Tecnólogo em Processamento de Dados, Faculdade de Tecnologia de São Paulo, São Paulo, 2012. Disponível em: < <http://www.fatecsp.br/dti/tcc/tcc00056.pdf> >. Acesso em: 29 abr. 2015.

LOCKHART, Josh. **PHP: Do Jeito: Design Patterns (Padrões de Projeto)**. Disponível em: < <http://br.phptherightway.com/pages/Design-Patterns.html> >. Acesso em: 21 maio 2015.

MEDEIROS, Higor. **Introdução ao Padrão MVC**. Disponível em: < <http://www.devmedia.com.br/introducao-ao-padrao-mvc/29308> >. Data de acesso: 19 mar. 2015.

MODULES, Angular. **Oauth-ng**: AngularJS directive for OAuth 2.0. Disponível em: <<http://ngmodules.org/modules/oauth-ng>>. Acesso em: 28 maio 2015.

MÜLLER, Leonardo. **JavaScript é a linguagem mais popular do mundo; Swift da Maçã cresce rápido**. 2015. Disponível em: < <http://www.tecmundo.com.br/programacao/72739-JavaScript-linguagem-popular-mundo-swift-maca-cresce-rapido.htm> >. Acesso em: 07 maio 2015.

MÜLLER, Nicolas. **Framework, o que é e para que serve?** 2008. Disponível em: < http://www.oficinadanet.com.br/artigo/1294/Framework_o_que_e_e_para_que_serve >. Acesso em: 27 abr. 2015.

NODEJS. **About Node.js**. Disponível em: <<https://nodejs.org/about/>>. Acesso em: 28 maio 2015.

OFICINADANET. **O que é Web Service?** Disponível em: < http://www.oficinadanet.com.br/artigo/447/o_que_e_web_service >. Acesso em: 29 abr. 2015.

OOCITIES. **Ejemplos de XML**. Disponível em: <http://www.oocities.org/es/guia_xml/ejemplos_de_xml.htm>. Acesso em: 28 maio 2015.

PANDA, Sandeep. **AngularJs: Novice to Ninja**. Estados Unidos da América: Sitepoint, 2014. 305 p.

PEREIRA, Ana Paula. **O que é XML?** 2009. Disponível em: < <http://www.tecmundo.com.br/programacao/1762-o-que-e-xml-.htm> >. Acesso em: 21 maio 2015.

PEREIRA, Henrique Costa. **Tableless vs Web Standards**. 2006. Disponível em: < <http://revolucao.etc.br/archives/tableless-vs-web-standards/> >. Acesso em: 21 maio 2015.

PHALCONPHP. **Dependency Injection/Service Location**. Disponível em: < <http://docs.phalconphp.com/en/latest/reference/di.html> >. Acesso em: 11 maio 2015.

PHP. **A classe Event**. Disponível em: < <http://php.net/manual/en/class.event.php> >. Acesso em: 04 maio 2015.

PHP. **Constantes pré-definidas**. Disponível em: < http://php.net/manual/pt_BR/errorfunc.constants.php >. Acesso em: 06 maio 2015.

PHP. **Documentation: PDF_new**. Disponível em: <http://php.net/manual/pt_BR/function.pdf-new.php>. Acesso em: 26 abr. 2015.

PHP. **Gettype**. Disponível em: < http://php.net/manual/pt_BR/function.gettype.php >. Acesso em: 21 abr. 2015.

PHP. **História do PHP**. Disponível em: < <http://php.net/history.php> >. Acesso em: 23 abr. 2015.

PHP. **Namespaces overview**. Disponível em: < http://php.net/manual/pt_BR/language.namespaces.rationale.php >. Acesso em: 04 maio 2015.

PHP-DI. **Understanding Dependency Injection**. Disponível em: < <http://php-di.org/doc/understanding-di.html> >. Acesso em: 04 maio 2015.

PLANSKY, Ricardo. **Definição, restrições e benefícios do modelo de arquitetura REST**, 2014. Disponível em: < <http://imasters.com.br/desenvolvimento/definicao-restricoes-e-beneficios-modelo-de-arquitetura-rest/> >. Data de acesso: 16 mar. 2015.

PLANSKY, Ricardo. **Definição, restrições e benefícios do modelo de arquitetura REST**. 2014. Disponível em: < <http://imasters.com.br/desenvolvimento/definicao-restricoes-e-beneficios-modelo-de-arquitetura-rest/> >. Acesso em: 30 abr. 2015.

PRADO, Romilson. **História do JavaScript**, 2014. Disponível em: < <http://www.romilsonprado.com.br/historia-do-JavaScript/> >. Data de acesso: 18 mar. 2015.

RAILS. **Desenvolvimento web sem dor**. Disponível em: <<http://www.rubyonrails.com.br/>>. Acesso em: 28 maio 2015.

RASMUSSEN, John. **Usando OAuth no IBM WebSphere DataPower Appliances, parte 1: Apresentando o Suporte OAuth 2.0 no Firmware do DataPower Revisão 5.0**. 2012. Disponível em: < http://www.ibm.com/developerworks/br/websphere/library/techarticles/1208_rasmussen/1208_rasmussen.html >. Acesso em: 11 maio 2015.

ROCHA, Fabio Gomes. **TDD: fundamentos do desenvolvimento orientado a testes**. Disponível em: < <http://www.devmedia.com.br/tdd-fundamentos-do-desenvolvimento-orientado-a-testes/28151> >. Acesso em: 05 maio 2015.

SANTOS, Welton L.. **Web Service – O que é Middleware?** 2015. Disponível em: < <https://cienciacomputacao.com.br/me-salva/web-service-o-que-e-middleware/> >. Acesso em: 14 maio 2015.

SATO, Fernando. **Programação estruturada**, 2009. Disponível em: < <http://www.fisica.ufjf.br/~sjfsato/fiscomp1/node23.html> >. Data de acesso: 18 mar. 2015.

SAUVÉ, Jacques Philippe. **Frameworks: O que é um Framework?** Disponível em: < <http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/frame/oque.htm> >. Acesso em: 27 abr. 2015.

SILVA, Maurício Samy. **JavaScript: Guia do programador**. São Paulo: Novatec Editora Ltda., 2010. 597 p.

SILVA, Paulo. **Seja organizado, sempre!** 2014. Disponível em: <<https://medium.com/@Neocite/seja-organizado-sempre-5464845d328e>>. Data de acesso: 24 mar. 2015.

SILVA, Viviane Torres da. **Frameworks.** 2010. Disponível em: <<http://www2.ic.uff.br/~viviane.silva/2010.1/es1/utl/aula13.pdf>>. Acesso em: 12 abr. 2015.

SOARES, Wallace. **PHP5: Conceitos, Programação e Integração com banco de dados.** 6. ed. São Paulo: Érica Ltda, 2010. 528 p.

SOURCEFORGE. **Documentação do PostgreSQL 8.0.0:** Uma breve história do PostgreSQL. Disponível em: <<http://pgdocptbr.sourceforge.net/pg80/history.html>>. Acesso em: 15 maio 2015.

SPRING. **Spring Framework.** Disponível em: <<http://projects.spring.io/spring-framework/>>. Acesso em: 28 maio 2015.

STAFF, Php Builder. **Lambdas in PHP.** 2014. Disponível em: <<http://www.phpbuilder.com/articles/application-architecture/optimization/lambdas-in-php.html>>. Acesso em: 04 maio 2015.

STENZEL, Carlos. **JavaScript um pouco de história,** 2013. Disponível em: <<http://www.carlosstenzel.com/TI/JavaScript-um-pouco-de-historia/>>. Data de acesso: 18 mar. 2015.

TILKOV, Stefan. **Uma rápida Introdução ao REST.** 2008. Disponível em: <<http://www.infoq.com/br/articles/rest-introduction>>. Acesso em: 30 abr. 2015.

VASCONCELLOS, Antônio et al. **ZEND FRAMEWORK - PHP.** 2010. Disponível em: <<http://www.inf.ufsc.br/~frank/INE5612/Seminario2010.1/Zend.pdf>>. Acesso em: 02 maio 2015.

W3SCHOOLS. **AJAX Tutorial.** Disponível em: <<http://www.w3schools.com/ajax/>>. Acesso em: 28 maio 2015.

W3SCHOOLS. **SOAP Introduction.** Disponível em: <http://www.w3schools.com/webservices/ws_soap_intro.asp>. Acesso em: 29 abr. 2015.