Universidade Federal
do Rio de Janeiro
Escola Politécnica

UM ESTUDO SOBRE PADRÕES E TECNOLOGIAS PARA O DESENVOLVIMENTO WEB – BACK-END

Vitor de Andrade

Projeto de Graduação apresentado ao Curso de Engenharia de Controle e Automação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro.

Orientador: Heraldo Luis Silveira de Almeida

Rio de Janeiro

Setembro de 2016

vitor.andrade@poli.ufrj.br

DRE: 110072408

UM ESTUDO SOBRE PADRÕES E TECNOLOGIAS PARA O DESENVOLVIMENTO WEB - BACK-END

Vitor de Andrade

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO DE ENGENHARIA DE CONTROLE E AUTOMAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO DE CONTROLE E AUTOMAÇÃO

Autores:

_____

Vitor de Andrade

Orientador:

_____

Heraldo Luis Silveira de Almeida, D.Sc.

Examinador:

_____

Flávio Luis de Mello, D.Sc.

Examinador:

_____

Aloysio de Castro Pinto Pedroza, Dr.

Rio de Janeiro – RJ, Brasil

Setembro de 2016

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

Escola Politécnica
Centro de Tecnologia, bloco H, sala H-217, Cidade Universitária
Rio de Janeiro – RJ CEP 21949-900

AGRADECIMENTOS

Gostaria de agradecer primeiramente a minha família. À minha mãe, Elaine, meu pai, José Roberto, e meus irmãos Lucas e Bruno, por todo o seu amor e por sempre me incentivarem e ajudarem nesta caminhada acadêmica.

Agradeço também a meu grande amigo Pedro Menezes, por ter me acompanhado em todos os momentos e nos esforços deste trabalho.

Finalmente, agradeço ao meu orientador, Heraldo, por todo o seu suporte no desenvolvimento deste trabalho.

RESUMO

Os sistemas voltados para a Web vêm se tornando mais populares a cada dia. Com o enorme crescimento do número de smartphones e a grande facilidade de acesso à Internet vivenciada nas últimas décadas, as pessoas passam cada vez mais conectadas. A cada dia, o mercado de trabalho pede mais profissionais capazes de entender, analisar e desenvolver sistemas web. Este texto contém um estudo sobre os conceitos e práticas mais importantes para o planejamento, desenvolvimento e sustentação da estrutura back-end de um sistema Web. Contém ainda um breve tutorial introdutório onde os conceitos apresentados serão demonstrados de forma prática.

Palavras-Chave: Redes, *Back End*, Desenvolvimento, *Software*, Web, Internet.

ABSTRACT

Web systems have become increasingly popular. Smartphones and Internet access have hugely grown in the last decades, allowing people to stay more connected every day. On the market, companies require qualified personnel, who are capable of understanding, analysing and developing web systems. This paper analyses the most important practices and concepts for the planning, development and sustaining of a web systems back-end. Besides, it presents a short tutorial where the presented topics can be implemented in a practical way.

Keywords: Networks, Back End, Development, Software, Web, Mobile, Internet

LISTA DE FIGURAS

Figura 1- Classe representando um retângulo 12

Figura 2 - Sistema utilizando a Inversão de Dependências 14

Figura 3 - Diagrama do TDD 18

Figura 4 - Inserção dos dados no banco 26

Figura 5 - Utilização da Factory para criação de um Logger 29

Figura 6 - Mensagem HTTP 31

Figura 7 - Interface REST em páginas Web (APIGEE) 41

Figura 8 - Interface REST em APIs 42

Figura 9 - Comparação de popularidade de busca entre os termos "xml api" e "json api" 49

Figura 10 - Diagrama de classes representando um relacionamento entre tabelas 57

Figura 11 - Criação de uma solução e um projeto no Visual Studio 2015 61

Figura 12 - Criação de um projeto Web API 62

Figura 13 - Estrutura de pastas e projetos da solução 63

Figura 14 - Adicionando uma referência de projeto 68

Figura 15 - Escolhendo servidor no SQL Server Management Studio 78

Figura 16 - Conexão do Visual Studio com o banco de dados 80

Figura 17 - Obtendo a string de conexão 81

Figura 18 - Navegador em Localhost indicando a execução do sistema 82

Figura 19 - Interface do Postman para criação de uma Pessoa no sistema 83

Figura 20 - Interface do postman para listagem de todas as pessoas no sistema 84

LISTA DE TABELAS

Tabela 1 - Métodos HTTP em uma API 44

Tabela 2 - Métodos HTTP representando relações entre recursos em uma API 46

Tabela 3 - Valores indicando a relação entre os links para a paginação 51

Sumário

Capítulo 1 - Introdução 1
1.1 - Tema 1
1.2 - Finalidade/Justificativa 1
1.3 - Escopo/Delimitação 2
1.4 - Definições, Acrônimos e Abreviaturas 2
1.5 - Metodologia 3
Capítulo 2 – Desenvolvimento orientado a objetos 5
2.1 - Princípios básicos da orientação a objetos 5
2.2 - Sintomas de más-práticas no projeto de um sistema 7
2.3 - S.O.L.I.D 9
Capítulo 3 - Arquitetura 16
3.1 - Projeto - DDD, TDD, BDD 16
3.2 - Separação em camadas 21
Capítulo 4 – Padrões de Projeto 24
4.1 - Repositórios 25
4.2 - Unit Of Work 25
4.3 - Injeção de dependências 27
4.4 - Factory 28
Capítulo 5 – Conceitos Básicos em Redes de Computadores – O protocolo HTTP 30
5.1 - O protocolo HTTP 30
5.2 - Mensagens HTTP 31
5.3 - MIME Types ou Media Types 32
5.4 - Métodos (ou Verbos) HTTP 32
5.5 - Códigos de estado HTTP (*Status Codes*) 33
5.6 - Headers 35
Capítulo 6 – Desenvolvimento de sistemas Web – Back end 37
6.1 - SOA ou APIs? 37
6.2 - O que é uma api? 37
6.3 - Por que fazer uma API? 38
6.4 - REST 38
6.5 - Visão de negócios 53
Capítulo 7 – Conceitos Básicos de Banco de Dados 56
7.1 - Banco de dados relacional 56
7.2 - Banco de dados não relacionais 58

Capítulo 8 – Tutorial para a criação de um sistema 60
8.1 - Criação do Sistema 60
8.2 - Criação do banco de dados 77
8.3 - Testando a API 81
Capítulo 9 - Conclusões 85
Referências 87

Capítulo 1

Introdução

1.1 - Tema

Este texto consiste em um estudo comparativo das tecnologias e conceitos a respeito do desenvolvimento do *back-end* de uma aplicação Web. A fim de demonstrar suas implementações de forma prática, será apresentado ao final do texto um breve exemplo demonstrando a criação de um sistema simples.

1.2 - Finalidade/Justificativa

De acordo com o projeto internetlivestats.com (INTERNETLIVESTATS), atualmente já temos mais de 3 bilhões de usuários de Internet no mundo e mais de um bilhão de páginas na web. Vemos cada vez mais as empresas oferecendo seus serviços através da Internet, que traz grande comodidade e praticidade para as pessoas. Além disso, com o grande aumento no número de smartphones, as pessoas vêm passando cada vez mais tempo conectadas.

As empresas de tecnologia da informação vêm cobrindo cada vez uma parcela maior do mercado. Mesmo em momentos de crise, ocorre grande demanda por profissionais qualificados neste setor. Para isto, é importante que os profissionais entendam bem sobre todas as etapas do desenvolvimento de software e sejam capazes de projetar e desenvolver sistemas estáveis e escaláveis.

O desenvolvimento de sistemas web engloba conhecimentos de várias áreas. Dentre outros, é importante que o engenheiro seja capaz de compreender a estrutura e

arquitetura do sistema, o funcionamento de redes de computadores, o desenvolvimento de interfaces de usuário e a forma de armazenamento e persistência das informações - como por exemplo utilizando bancos de dados. Além disso, assim como no desenvolvimento voltado para outros fins, deve sempre seguir as boas práticas de forma que o sistema seja escalável e de fácil manutenção.

1.3 - Escopo/Delimitação

Este texto se destina a pessoas que já possuam algum conhecimento e experiência nas áreas de algoritmos, programação e desenvolvimento de software, porém não conhecem as práticas e padrões de mercado para o desenvolvimento de sistemas web, sejam elas estudantes, profissionais recém-formados, profissionais já estabelecidos em outras áreas de desenvolvimento que queiram se atualizar a respeito do desenvolvimento web ou empreendedores planejando iniciar um negócio nestas áreas.

A didática será sempre muito valorizada, de forma que o texto seja de fácil compreensão mesmo para pessoas sem um profundo conhecimento na área de computação. Porém, em alguns momentos, os conceitos serão aprofundados e conhecimentos prévios poderão facilitar a compreensão do texto.

Todos os conceitos apresentados serão agnósticos quanto à linguagem, e podem ser implementados nas aplicações independentemente da linguagem desejada. Ao final do texto, será exemplificada a implementação de um sistema utilizando a linguagem C# e o framework .Net.

1.4 - Definições, Acrônimos e Abreviaturas

- **URI**

De acordo com o RFC 2396, a URI consiste em uma string de caracteres usada para identificar um recurso físico ou abstrato. Existem dois tipos de URI: as URLs, que indicam o

local online onde um recurso se encontra, e a URN, que indica o nome do recurso, e é independente de seu local.

Exemplo de url: http://www.poli.ufrj.br/ (Url da Politécnica da UFRJ)

Exemplo de urn: urn:ietf:rfc:2648 (URN do RFC 2648)

- **Manutenibilidade**

De acordo com a ISO IEC 9126, a manutenibilidade é a "Capacidade do produto de software de ser modificado. As modificações podem incluir correções, melhorias ou adaptações do software devido a mudanças no ambiente e nos seus requisitos ou especificações funcionais".

- **Escalabilidade**

A escalabilidade é a capacidade de um sistema de acomodar o aumento do número de objetos ou elementos, do processamento de volumes cada vez maiores e de ser capaz de crescer de forma sustentável. (BONDI, A.B.)

- **RFC (Request for Comments)**

São documentos técnicos criados pela IETF, que definem e detalham as implementações de diversos padrões a respeito de redes e da Internet.

- **IETF (Internet Engineering Task Force)**

É uma comunidade de desenvolvedores, operadores e pesquisadores voltada para a evolução da arquitetura e das operações da Internet.

1.5 - Metodologia

O texto iniciará fazendo uma análise sobre diversos conceitos importantes para o desenvolvimento do back-end de um sistema web. Dentre os conceitos apresentados, estarão as boas práticas de desenvolvimento orientado a objetos, arquitetura do *back-end* de um sistema web, padrões de projeto, redes de computadores, APIs e banco de dados. Em seguida, será

apresentado um tutorial para o desenvolvimento de um sistema simples contendo alguns dos conceitos presentes nos capítulos anteriores.

Capítulo 2

Desenvolvimento orientado a objetos

Para se manter competitivo, um sistema deve sempre mudar. Novas funcionalidades devem ser implementadas, processos podem ser melhorados e *bugs* devem ser corrigidos. Conforme um sistema se torna mais popular e possui mais clientes, essas manutenções se tornam cada vez mais importantes. Entretanto, sistemas mal planejados e estruturados tornam-nas cada vez mais difíceis. Pequenas mudanças podem causar grandes problemas em diferentes partes do sistema. A experiência de engenheiros e desenvolvedores com o passar do tempo propiciou o surgimento de padrões de desenvolvimento que minimizam estes problemas e tornam o desenvolvimento e a manutenção dos sistemas mais simples e eficazes. Este capítulo contém uma análise dos princípios observados por estes profissionais, além de indicar características desejáveis para um sistema orientado a objetos.

2.1 - Princípios básicos da orientação a objetos

Existem 3 princípios básicos que maximizam a escalabilidade e as chances de sucesso de um sistema orientado a objetos. São eles:

Baixo acoplamento

O acoplamento é o grau em que uma classe, módulo ou qualquer outra parte do sistema depende de outras partes. É importante notar que o acoplamento, por si só, não é algo ruim. Uma classe que possua zero acoplamento não depende de absolutamente nenhuma outra parte do sistema, nem outras partes dependem dela. Essencialmente, isto faz dela uma classe inútil para o sistema. O processamento executado por ela não traria nenhuma funcionalidade e seria impossível entrar ou receber quaisquer dados a partir dela. Entretanto, é muito importante

manter os níveis de acoplamento de um sistema baixos. Reduzi-los melhora muito as possibilidades de reutilização de partes do sistema, além de facilitar as manutenções e os testes automatizados.

Alta coesão

A coesão indica o grau em que diversas partes do sistema trabalham em conjunto para alcançar os objetivos desejados. Um sistema possui alta coesão quando suas partes trabalhando em conjunto são capazes de criar algo muito maior do que cada uma seria capaz de criar isoladamente.

Podemos pensar na coesão a partir de vários níveis diferentes. Por exemplo:

1. Coesão de um método ou função - Indica o grau de interação entre as linhas de código de um método de forma a atingir o objetivo especificado pelo nome do método.
2. Coesão de uma classe - Indica o grau de interação entre os métodos e propriedades de uma classe de forma a atingir o objetivo especificado pelo nome da classe.
3. Coesão dos módulos - Indica o grau de interação entre classes de forma a criar um módulo com um objetivo claro e específico para um sistema.
4. Coesão das camadas - Indica o grau de interação entre os módulos de forma a criar uma camada com um objetivo claro e específico para o sistema.

Estes são só alguns exemplos de níveis que podem ser definidos. Dependendo da arquitetura utilizada para o sistema, é possível que existam outras definições e níveis de coesão para aquele sistema em específico.

Encapsulamento

Encapsulamento consiste em ocultar informações como propriedades, métodos ou até mesmo a lógica e processos que ocorrem internamente em um sistema. As interações entre as partes do sistema devem se dar a partir de interfaces públicas que indicam claramente seu propósito e forma de utilização. Quando um sistema é bem encapsulado, quaisquer alterações em detalhes internos de implementações podem ser feitas à vontade sem impactar outras partes do sistema que utilizam aquela dependência. Mais ainda, a utilização de quaisquer módulos ou partes do sistema pode ser feita sem o conhecimento da forma com que eles foram implementados.

2.2 - Sintomas de más-práticas no projeto de um sistema

Rigidez

Consiste na dificuldade de aplicar mudanças em um sistema. Um sistema rígido precisa de alterações em diversos módulos e classes para que sejam efetuados quaisquer pequenos ajustes. Isto torna as manutenções demoradas e complicadas, aumentando muito o receio dos donos do produto a respeito de ajustes no sistema.

Fragilidade

Indica qual a chance de que uma alteração cause falhas inesperadas em uma aplicação. Sistemas frágeis costumam apresentar muitas falhas em locais não relacionados a qualquer alteração feita, o que também aumenta a desconfiança e diminui sua credibilidade frente aos clientes e donos do produto.

Imobilidade

A imobilidade caracteriza a incapacidade de reutilização de trechos de códigos ou funcionalidades em um sistema. Ela é muitas vezes causada por um grande número de

dependências das classes envolvidas (alto acoplamento). Quando isso ocorre, o sistema pode apresentar comportamentos inesperados a partir da reutilização de software. Por receio destes comportamentos inesperados, muitas vezes a saída utilizada é reescrever o código para aquela funcionalidade específica, caracterizando a duplicação do código, que reduz a eficiência e clareza do sistema.

Viscosidade

Existem dois tipos de viscosidade: a do software e a do ambiente. Ambas têm relação com a dificuldade de aplicação de mudanças corretamente em um sistema.

Idealmente, o desenvolvimento de novas funcionalidades deve seguir os padrões de projeto e de arquitetura do sistema. Para isso, o código do sistema deve ser claro e os padrões devem facilitar novas implementações. Caso contrário, os desenvolvedores buscarão alternativas mais simples e rápidas de implementação. Quando as soluções alternativas são mais facilmente implementadas do que aquela seguindo os padrões do sistema, é dito que ele possui uma alta **viscosidade do software**.

Por outro lado, temos ainda o conceito de **viscosidade do ambiente**. Características como um alto tempo de compilação ou de execução dos testes automatizados podem induzir os desenvolvedores a evitar recompilações constantes ou execuções dos testes. Estas características se apresentam em sistemas com alta viscosidade do ambiente e aumentam os riscos de falhas em novas implementações.

Repetição desnecessária

Consiste em trechos do sistema onde a lógica é repetida. Pode acontecer tanto por apresentar trechos de código copiados e colados, quanto por uma lógica de negócios que tenha sido replicada em diferentes partes do sistema. É muito comum ocorrer em códigos mal-encapsulados e sistemas imóveis. Esta característica aumenta o número de pontos de manutenção quando alguma alteração deve ser feita, além de reduzir a clareza do código.

Opacidade

Representa a clareza do código. Um sistema opaco é aquele em que a compreensão do código é fortemente comprometida, prejudicando muito as manutenções e aumentando as chances de falhas causadas por má compreensão da lógica do sistema.

Complexidade desnecessária

O receio de falhas de projeto pode muitas vezes induzir o engenheiro a tornar o sistema excessivamente complexo sem uma real necessidade para isto. Esta prática acaba muitas vezes aumentando a opacidade e a rigidez do sistema, além de atrasar a entrega do sistema ou da funcionalidade.

2.3 - S.O.L.I.D

S.O.L.I.D é um acrônimo para "*Single responsibility, open-closed, Liskov substitution, interface segregation and dependency inversion*". Estes conceitos foram apresentados por Robert C. Martin (*a.k.a Uncle Bob*) como os primeiros 5 princípios da orientação a objetos. É importante notar que não são regras absolutas para o desenvolvimento. Em vez disso, são um guia para boas práticas que ajudam a evitar os problemas de projeto mencionados na seção anterior. Nesta seção será feita uma breve explicação sobre cada um destes princípios.

2.3.1 - O princípio da responsabilidade única (Single responsability principle).

"Uma entidade do sistema deve ter um, e apenas um, motivo para mudar".

Ou, apresentando de outra forma, pode existir apenas um requisito do sistema que, quando alterado, causará uma mudança em uma classe.

Quando uma classe possui várias responsabilidades, qualquer pequena alteração no sistema pode acabar causando problemas inesperados em outras partes do sistema.

Para exemplificar, vamos considerar uma classe de várias responsabilidades chamada *Authentication*, que possui as seguintes funcionalidades:

1. Receber as credenciais de login de um usuário
2. Verificar no banco de dados se as informações estão corretas.
3. Caso as credenciais estejam corretas, redirecionar o usuário para a página principal do sistema. Caso contrário, retornar uma mensagem de erro.

De imediato, podemos ver duas responsabilidades na classe. A primeira delas é visual - receber os dados do usuário e redirecioná-lo para uma página específica. A segunda é a respeito do banco de dados - deve buscar o usuário no banco e validar sua autenticação

Digamos que após o sistema ter sido lançado, foi encontrado um *bug* no HTML em que, para uma versão de browser específica, a página não está sendo carregada corretamente. Este é um erro puramente visual, e para corrigi-lo, deveria ser feita apenas uma alteração simples no código HTML da página. Entretanto, como todas as responsabilidades estão na mesma classe, não existe uma forma de se entregar o código corrigido sem recompilar e entregar também toda a parte de acesso aos dados e validação do usuário. As chances de que esta manutenção impactem uma funcionalidade totalmente independente aumentam consideravelmente.

Uma discussão de grande importância para este princípio é a respeito de como podemos *definir* uma responsabilidade. Afinal, existem infinitas formas de separar as funcionalidades de um sistema. No simples exemplo acima, podemos pensar por exemplo, se devemos separar as responsabilidades de obter o usuário do banco de dados daquela de comparar se as credenciais inseridas estão corretas. Não existe uma regra universal para esta pergunta. O que o desenvolvedor deve sempre ter em mente é que as responsabilidades devem ser definidas de acordo com quais mudanças realmente podem ser necessárias no sistema. Por exemplo: é um cenário possível que a forma de acesso ao banco de dados seja alterada? Podem surgir novos requisitos a respeito das informações obtidas a partir dali? Existe algum caso onde a validação das credenciais poderá ser alterada sem modificar a parte de acesso aos dados do banco?

Duas responsabilidades devem ser separadas apenas quando uma mudança independente em cada uma delas é realmente um cenário possível em um horizonte futuro do sistema. Separar responsabilidades apenas por separá-las, pode caracterizar uma *complexidade desnecessária*.

2.3.2 - Princípio do Aberto Fechado (Open-Closed principle)

"Uma entidade do sistema deve ser aberta para extensões, mas fechada para modificações".

Na prática, isto significa que quando uma classe está pronta, nós não devemos mais alterá-la para adicionar novas funcionalidades. Quaisquer alterações em termos de funcionalidade deve ser uma nova classe, que herda daquela que já existia. As alterações em uma classe existentes são permitidas apenas para consertar *bugs* encontrados nela, mas nunca para estender suas funcionalidades.

As linguagens de programação orientadas a objeto nos oferecem algumas ferramentas que auxiliam na implementação deste princípio, através da herança, classes abstratas e interfaces públicas.

2.3.3 - Princípio da substituição de Liskov (Liskov substitution principle)

"Subtipos devem ser substituíveis pelos seus tipos base".

O significado prático deste princípio é que qualquer objeto que herde de uma interface ou qualquer outra abstração deve implementar todas as funcionalidades de sua classe base de forma coerente, sem apresentar nenhum comportamento inesperado do ponto de vista do cliente.

O exemplo mais clássico de violação deste princípio consiste na implementação de classes para representar um retângulo e um quadrado. A classe *Rectangle* pode ser representada da seguinte forma:

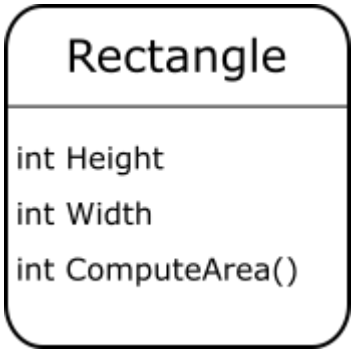

Figura 1- Classe representando um retângulo

Como sabemos que todo quadrado é um retângulo, podemos nos sentir tentados a definir a classe *Square* herdando da classe *Rectangle*. Porém imediatamente perceberíamos uma diferença entre eles. Na classe *Square*, nós deveríamos sobrescrever os *Setters* das propriedades *Height* e *Width*, de forma a garantir que a altura e a largura fossem sempre iguais.

O que aconteceria entretanto, se definíssemos um objeto quadrado da seguinte forma:

Rectangle square = new Square();

Sabemos que, pelo conceito de herança, essa implementação deveria fazer sentido, uma vez que todo quadrado é um retângulo. Porém com ela, o objeto *square* irá utilizar as funcionalidades da classe base - *Rectangle.* Isso desconsidera a restrição implementada na classe *Square* onde a altura e a largura do quadrado devem ser iguais. Dessa forma, pode ocorrer o seguinte problema em uma utilização destas classes:

square.Height = 2;

square.Width = 3;

square.ComputeArea();

O resultado da chamada ao método *ComputeArea* irá depender da forma que o objeto *square* foi definido. Se ele foi definido através da classe *Rectangle*, o resultado será 6. Entretanto, se foi definido pela classe *Square*, a definição da largura como 3 irá sobrescrever a

definição da altura como 2. Dessa forma, o resultado da chamada ao *ComputeArea* seria igual a 9. Esta discrepância caracteriza a violação do princípio da substituição de Liskov.

2.3.3 - O princípio da segregação de interfaces (Interface segregation principle)

"*Os clientes não devem ser forçados a depender de métodos que não usem*"

Na prática, o significado deste princípio é que não devemos generalizar demais as interfaces criadas, para evitar que os clientes sejam obrigados a implementar métodos que eles não utilizarão.

Assim como no princípio da responsabilidade única, não existe uma regra para definir como será feita a segregação das interfaces, ou até que ponto vale a pena separar seus métodos em diferentes interfaces. Esta discussão depende do projeto, dos requisitos e de uma análise das possibilidades para o futuro do sistema. O importante aqui é garantir que em nenhum momento, algum cliente ou alguma classe que implementa uma interface seja obrigado a escrever métodos que não utilizarão, causando muitas vezes que as classes contenham métodos que não fazem nada ou lançam exceções de método não implementado.

2.3.4 - O princípio da inversão de dependências

Este princípio possui duas partes:

A. "Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações"
B. "Abstrações não devem depender dos detalhes da implementação. Os detalhes devem depender das abstrações"

Os módulos de alto nível do sistema são aqueles que são responsáveis pelas regras de negócio. Por isso, é essencial que eles não sejam impactados por quaisquer alterações feitas

nos chamados módulos de baixo nível - interfaces de usuários, detalhes de implementações, camadas de dados, etc.

O diagrama a seguir mostra um exemplo de como deve ser estruturada uma aplicação onde o princípio da inversão de dependências é aplicado:

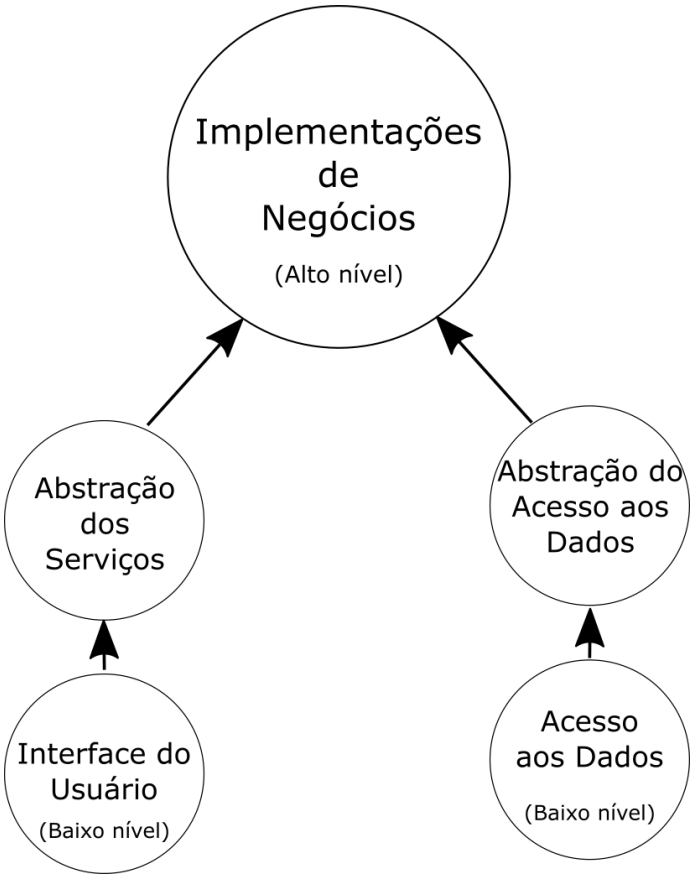

Figura 2 - Sistema utilizando a Inversão de Dependências

As setas demonstram dependências, de forma que a camada de interface de usuário depende da abstração dos serviços, que foi construída a partir da camada de implementação das regras de negócio do sistema. Analogamente, as implementações de acesso aos dados dependem de uma abstração do acesso aos dados que foi definida de acordo com a implementação das regras de negócios.

2.3.4 - Considerações finais sobre os princípios

É importante ter em mente que estes princípios não são regras rígidas que devem ser seguidas a todo tempo. Existem diversas outras variáveis que devem entrar em consideração, como por exemplo o tempo para a entrega e a complexidade do sistema sendo desenvolvido. Se o engenheiro desejar já iniciar o projeto tendo em mente que nunca irá desobedecer a nenhum destes princípios, muitas vezes o tempo de desenvolvimento até a primeira entrega pode se tornar muito maior. Em um sistema que não tende a evoluir e crescer demais, os benefícios trazidos pelos princípios S.O.L.I.D podem muitas vezes não compensar o aumento no tempo de desenvolvimento e da complexidade do sistema. Assim como em outras áreas da Engenharia, deve-se sempre considerar estes *trade-offs* entre os benefícios e os custos do desenvolvimento.

Capítulo 3

Arquitetura

3.1 - Projeto - DDD, TDD, BDD

3.1.1 - Domain Driven Design

O Projeto Orientado ao Domínio, ou Domain Driven Design (DDD), é um conceito apresentado por Eric Evans em seu livro de mesmo nome, lançado em 2003 (EVANS, E. 2003). Neste livro, o autor aborda práticas para um projeto disciplinado do software, alinhando a parte técnica e a não-técnica, de forma que o sistema seja feito sempre se considerando seu valor de negócios. Dizer que o projeto é focado no Domínio do software, significa dizer que este software deve automatizar um processo e atender completamente um negócio.

Outro conceito importante no projeto orientado ao Domínio é a **linguagem universal**. Isto significa que todos os agentes envolvidos no desenvolvimento de software devem estar sempre em contato para que todos entendam os mesmos conceitos e usem os mesmos nomes para todos os objetos de negócio. As nomenclaturas utilizadas devem ser reproduzidas, desde conversas entre os donos do produto e desenvolvedores, até os nomes das propriedades nos códigos. Esta é uma forma de tornar a comunicação a mais clara possível e garantir que todos os envolvidos entendem perfeitamente todos os conceitos necessários.

A **separação de camadas** é um conceito importante não só para o projeto orientado ao domínio, mas para as mais diversas estratégias e filosofias de projeto de software. Ela ajuda a reduzir o acoplamento do sistema e aumentar a coesão. No DDD, a importância desta separação também é levantada, e define um nome específico para a camada onde se encontram as principais regras de negócio do sistema - a camada de Domínio.

Para conceitualizar o Domínio do sistema, devemos introduzir a ideia de um **Modelo**, formado por Entidades, Objetos de Valor, *Factories*, Serviços e Repositórios.

As **Entidades** constituem tudo aquilo que possui valor ao domínio. No exemplo do aplicativo da agenda telefônica apresentado ao final deste trabalho, podemos considerar como entidades, por exemplo, as *Pessoas*. É importante que todas as entidades possuam uma identidade única para o sistema, que pode ser gerada automaticamente ou ter algum significado, como por exemplo um CPF do cliente.

Os **objetos de valor** representam objetos que também possuem valor para o sistema, porém são reconhecidos por seus atributos, sendo geralmente imutáveis. A principal diferença entre as entidades e os objetos de valor é que uma entidade pode mudar seus atributos sem alterar a sua identidade, porém os objetos de valor não.

Os **serviços** são ferramentas utilizadas para executar as tarefas relativas às entidades e aos objetos de valor. Eles possuem toda a parte do processamento das regras de negócio.

As **factories**, ou fábricas são utilizadas para a criação de objetos. Mais detalhes serão explicados na seção de padrões de projeto.

Os **repositórios** são utilizados para intermediar a camada de domínio e a lógica de acesso aos dados. Normalmente, são utilizados *frameworks* como o *NHibernate*, *Entity Framework* ou o *Dapper* para efetuar a conexão com um banco de dados relacional. É importante que esta camada de acesso aos dados não possua nenhuma regra de negócio, mas somente saiba como obter e gravar os dados no banco.

Por último, é importante notar que o DDD preza as *pequenas entregas*, de forma a garantir que os objetivos de negócio e o desenvolvimento estejam sempre alinhados. Para alcançar isso, uma das recomendações é a de utilizar metodologias ágeis de desenvolvimento, como o SCRUM ou o KANBAN. Estes métodos não são obrigatórios para se aplicar a filosofia

do projeto orientado ao domínio, porém, por prezarem pequenas entregas e *feedbacks* constantes, eles podem se alinhar muito bem ao DDD.

3.1.2 - Test Driven Development

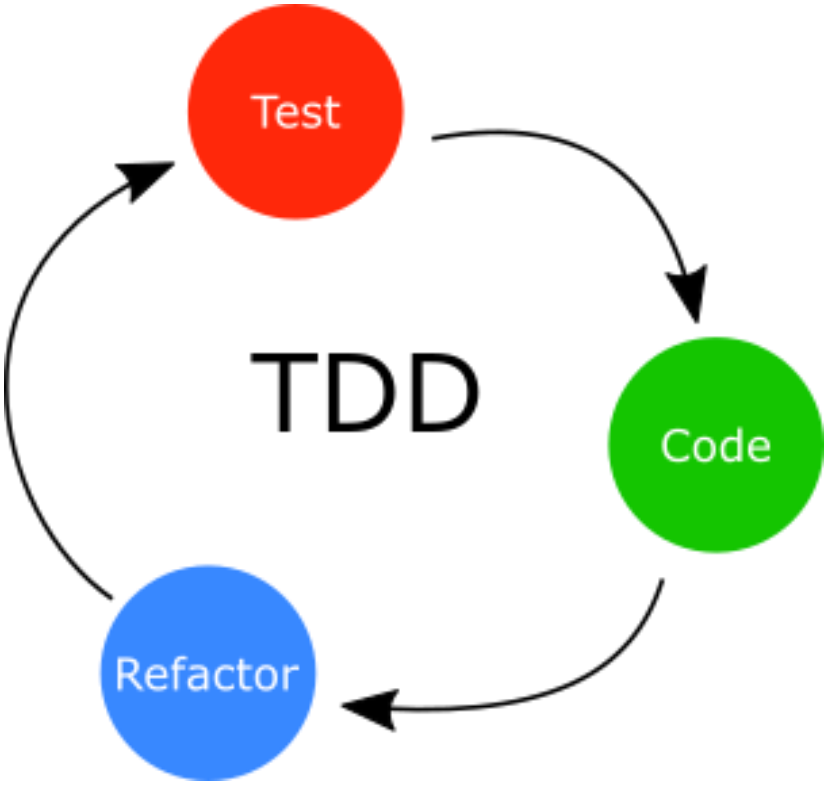

Figura 3 - Diagrama do TDD

O Desenvolvimento Orientado a Testes (*Test Driven Development* ou TDD) (BECK, K., 2002), muito apoiado pelos defensores do *Extreme Programming* (XP) (BECK, K., 2004), preza pela cobertura de 100% do código por testes automatizados. A ideia é que o desenvolvimento de software deve sempre seguir o seguinte ciclo:

1. Escrever um teste para um comportamento esperado do sistema, mesmo antes de ter escrito o código para executar este comportamento.

2. Executar o teste e verificar a sua falha.
3. Codificar a parte do sistema necessária para que o teste escrito seja executado com sucesso
4. Executar novamente o teste, e verificar o sucesso
5. Refatorar o código e o teste para que a funcionalidade e o teste reflitam perfeitamente o comportamento esperado e se adequem às boas práticas de desenvolvimento. Nesta etapa, é importante que tanto o código recém-escrito quanto o código já existente do sistema sejam refatorados de forma a se manter a estrutura e boas práticas de desenvolvimento no sistema.
6. Seguir para o próximo caso de uso do sistema e executar todos esses passos novamente.

3.1.3 - Behaviour Driven Development

O Desenvolvimento Orientado ao Comportamento (Behaviour Driven Development ou BDD) foi proposto por Dan North em 2003 como uma resposta ao TDD (NORTH, D., 2006). Ele veio para tornar os requisitos do sistema mais claros e melhorar a comunicação entre os desenvolvedores e os donos do produto. Para isso, defende algumas ideias semelhantes ao DDD, como a utilização da **linguagem universal** e o desenvolvimento de fora para dentro (ou *outside-in development*) para envolver todas as pessoas - técnicas e não-técnicas - no projeto. Além disso, esta filosofia dá grande importância à criação dos testes e aos *feedbacks* contínuos.

Os **testes automatizados** utilizando dublês de teste (como *mocks* ou *stubs*) também ajudam a proporcionar *feedbacks* contínuos, o que permite ajustes e adequações a mudanças de requisitos de forma ágil. Além disso, escrever os nomes dos testes como sentenças completas descrevendo claramente suas intenções e iniciando com a palavra *should* é altamente recomendado. Isto torna mais claro o comportamento esperado do teste para um desenvolvedor, além de permitir que *frameworks* auxiliem na criação da documentação. O framework

agiledox, por exemplo, gera um texto de documentação a partir dos nomes e das verificações presentes nos testes, de forma simples e inteligente, para sistemas em Java.

As funcionalidades desejadas para o sistema devem ser escritas em duas partes. A primeira indica quem solicitou, qual a funcionalidade e o valor de negócio, da seguinte forma:

Funcionalidade: [Nome da funcionalidade]

Para [Valor de negócio]

Eu, como [Papel da pessoa que solicitou a funcionalidade]

Desejo poder realizar [Funcionalidade]

A segunda etapa consiste em destacar os cenários em que a funcionalidade será validada, indicando o estado inicial do sistema, a ação a ser realizada e o comportamento esperado após a sua execução:

Cenário: [Nome do cenário]

Dado que [Um contexto inicial do sistema]

Quando [Um evento que irá acontecer]

Então [Comportamento esperado após a ação]

Para descrever os cenários, é possível que o contexto inicial ou o comportamento esperado precisem de uma descrição mais detalhada do que apenas uma sentença. Neste caso, é possível conectar várias sentenças com conectores lógicos, por exemplo:

Cenário: Cadastro de um cardápio

Dado que: Um usuário está logado

E: O usuário é um estabelecimento

Quando: O usuário clicar no botão “Adicionar cardápio”

Então: Ele deverá ser redirecionado para uma tela de criação de cardápio.

3.1.4 - Como decidir

É importante que todos estes conceitos apresentados não sejam regras ou padrões rígidos a respeito do desenvolvimento do software. Ao iniciar o desenvolvimento, não existe uma regra universal de qual destas filosofias seguir. O importante é que o desenvolvedor conheça bem todos os conceitos apresentados, para que possa definir qual é a filosofia que se adequa melhor ao seu sistema e seu ambiente de desenvolvimento. Muitas vezes, a melhor escolha poderá ser seguir recomendações pontuais de cada uma das filosofias, uma mistura entre elas ou até mesmo nenhuma delas. Em outros casos, uma delas pode se adequar perfeitamente às necessidades do projeto, de forma que uma delas seja seguida à risca. Ao conhecer e entender todas as possibilidades para projeto de software, cada caso poderá ser estudado em específico e discussões com os outros membros da equipe poderão ser iniciadas, de forma a escolher qual a melhor opção de projeto.

3.2 - Separação em camadas

A separação do sistema em camadas é uma recomendação da maior parte das estratégias de desenvolvimento de software. Ela permite agrupar os componentes a partir de suas funcionalidades, o que aumenta o suporte à reutilização dos componentes e consequentemente aumenta a coesão e reduz o acoplamento do sistema. Além disso, a divisão em camadas proporciona também mais facilidade em termos de manutenção do software. Ao ter uma divisão clara das camadas, é mais intuitivo para um desenvolvedor onde cada funcionalidade está localizada e o local onde uma manutenção deve ser feita.

É importante notar que esta separação das camadas é feita logicamente e é totalmente independente da separação física das partes do sistema. É muito comum encontrar sistemas em que todas as camadas lógicas se encontram hospedadas em um único servidor.

Uma estratégia muito comum de separação consiste em um sistema de 3 camadas:

- **Camada de Apresentação** - É a camada que será exposta para os clientes, sendo responsável pelas interações com os usuários. Pode ser considerada uma porta através da qual os clientes podem acessar as regras especificadas na camada de negócios. Esta camada pode conter, por exemplo, uma *API REST* para oferecer a interação com o usuário ou uma interface web através do qual os clientes poderão navegar na aplicação. As APIs serão explicadas mais a fundo no capítulo 6.
- **Camada de Negócios** - Contém a implementação das regras de negócio propriamente ditas. Possui muitos nomes possíveis, de acordo com a filosofia utilizada, como por exemplo a camada do Domínio especificada no DDD.
- **Camada de Dados** - É a camada responsável pela persistência e o mapeamento dos dados para dentro do sistema. Um exemplo de utilização desta camada é através de repositórios, que serão responsáveis por obter ou gravar as informações em um banco de dados. É muito importante que esta camada não possua nenhuma inteligência ou implementação de regra de negócios.

Algumas funcionalidades, como logs e gerenciamento das configurações da aplicação, devem ser utilizadas em diversas camadas, ou até mesmo em todas as camadas do sistema. Um conceito interessante abordado pela filosofia do DDD que pode ser apropriado em outras filosofias é a criação de uma camada separada, chamada de **cross-cutting**, que é responsável por lidar com este tipo de operações.

Outra vantagem de uma separação de camadas feita apropriadamente é a independência entre elas. Por exemplo, se em algum momento após o lançamento do sistema, for decidido que o *framework* de acesso aos dados utilizado não é o melhor possível, apenas a camada de dados poderá ser substituída, sem que haja nenhum impacto ou alteração necessária na camada de negócios. Isto minimiza chances de erros, diminuindo consideravelmente a rigidez e fragilidade do sistema.

Capítulo 4

Padrões de Projeto

Criar um projeto de software orientado a objetos é uma tarefa complicada. Deve-se ter em mente que o sistema deve atender a todos os requisitos perfeitamente, tendo flexibilidade para crescer e receber novos requisitos. Além disso, deve manter a clareza do código para que futuros desenvolvedores possam entendê-lo e aplicar manutenções de forma fácil e segura. Além de considerar tudo isso, devemos ainda nos preocupar em entregar o sistema de forma rápida, evitando o que é chamado de *Big Design Up Front*, que ocorre quando o desenvolvedor deseja cobrir todos os possíveis problemas e requisitos futuros que possam surgir, e isso acaba atrasando muito a entrega do sistema.

Para otimizar ao máximo a tarefa de projetar o sistema de forma escalável e robusta, os desenvolvedores têm em mãos todo um leque de padrões e práticas que já os ajudaram a resolver diversos problemas no passado. Estes padrões foram criados através de muita prática e experimentação para resolver problemas que se repetiam através dos diversos sistemas implementados.

Neste capítulo, alguns padrões muito recorrentes no desenvolvimento de software serão enunciados, sendo feita uma explicação breve sobre a funcionalidade e o contexto no qual estes padrões são utilizados. Não entraremos em detalhes a respeito da implementação destes padrões, uma vez que a implementação pode ser feita de diversas formas e com o auxílio de vários *frameworks*, dependendo do contexto e da aplicação.

Para um guia mais completo dos padrões, pode ser consultado GAMMA, E., HELM, R., et al 1994. Este livro, conhecido como o Gang of Four, é uma das maiores referências a respeito de padrões de projetos em software orientado a objetos.

4.1 - Repositórios

Como vimos no capítulo 3, é altamente recomendado em aplicações web que a camada de acesso aos dados seja independente das outras camadas do sistema. Em muitos destes sistemas, a camada de dados é responsável por acessar um banco de dados relacional, como o Sql Server, MySql ou o Oracle. O padrão de Repositório é utilizado para definir como será feito o acesso às informações na camada de dados do sistema, auxiliando assim, na separação das camadas. Desta forma, é responsável por obter os dados e mapear para as entidades, que serão utilizadas na camada de negócios para as operações relacionadas às regras de negócio do sistema. Além disso, é responsável pela persistência, onde os dados são inseridos ou atualizados no banco.

É importante que os repositórios não possuam nenhum processamento de regras de negócios, para garantir o princípio da responsabilidade única. Dessa forma, garantimos que a única responsabilidade do repositório é a de manipular os dados, e toda a regra de negócio deverá ser implementada nas outras camadas do sistema.

4.2 - Unit Of Work

O padrão da unidade de trabalho, ou *Unit of Work*, é responsável por coordenar as alterações feitas nas entidades obtidas pelos repositórios, de forma a garantir a persistência correta de todas as informações no banco. Para isso, gerencia possíveis alterações feitas em um objeto existente no banco, verificando atualizações ou exclusões nos dados, assim como inserções de novas informações no banco de dados.

Uma importante responsabilidade da unidade de trabalho é o gerenciamento das chamadas *transações* do banco de dados. Elas são responsáveis por garantir a consistência de todas as informações que estejam sendo alteradas no banco. Por exemplo, quando um processamento é responsável por alterações em diversas entidades, é importante que as

alterações sejam feitas no banco de forma atômica. Isto significa que todas as operações devem ser feitas com sucesso ou falha, em conjunto. Por exemplo, digamos que estamos cadastrando um novo cliente no sistema, e para isso temos que inserir tanto os dados pessoais do cliente, como seu endereço. Em um banco de dados relacional, é comum que estas sejam entidades distintas, cada uma possuindo uma tabela e um repositório para lidar com o mapeamento dos dados. Na prática, isto significa que os dados cadastrados serão inseridos em momentos distintos do código. Vamos dizer que os dados pessoais serão inseridos primeiro, e em seguida será inserida a informação a respeito do endereço.

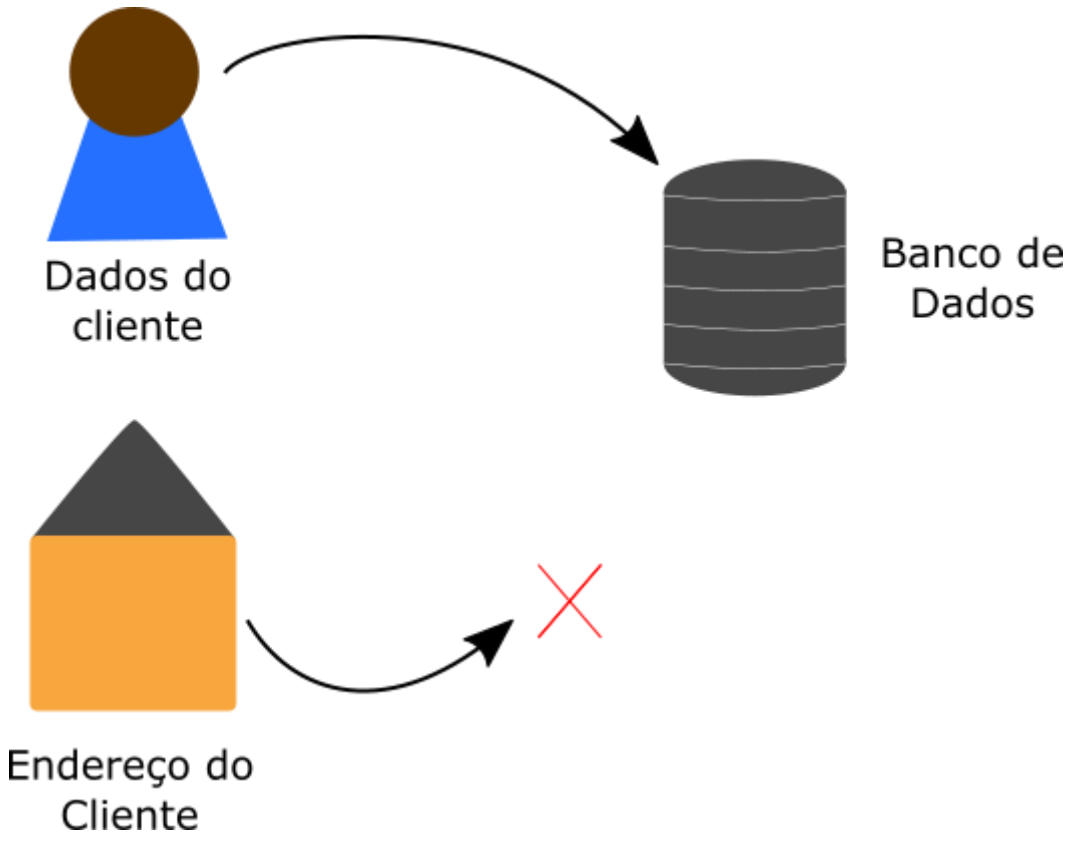

Figura 4 - Inserção dos dados no banco

Agora digamos que a persistência dos dados do cliente foi feita com sucesso, mas ao inserir os dados do endereço, ocorreu um erro de comunicação com o banco de dados, e o sistema não foi capaz de inserir estes dados no banco. É importante que o sistema não insira os dados corrompidos no banco, mas sim que ele reverta a operação de inserção dos dados pessoais do comprador. É aí que uma transação do banco de dados e a unidade de trabalho

entram. A unidade de trabalho irá reverter a transação (operação chamada de *rollback* da transação), de forma que nenhum dos dados seja inserido. A partir daí, a aplicação poderá tratar o erro e informar ao cliente que o cadastro não pode ser efetuado com sucesso.

4.3 - Injeção de dependências

Quando definimos o conceito de acoplamento, foi dito que um sistema de zero acoplamento é essencialmente um sistema inútil. É essencial que objetos interajam entre si para que qualquer sistema possa executar uma operação ou uma lógica de negócios. Porém, como são definidas estas interações?

Tradicionalmente, cada objeto é responsável por definir e instanciar todas as suas *dependências* - os objetos do sistema com os quais ele interage. O problema desta lógica é que ela aumenta muito o acoplamento, dificultando testes e reutilizações de uma classe de forma isolada de suas dependências.

O padrão de Injeção de Dependências foi criado exatamente para dar uma forma de isolar um objeto de todas as suas dependências, reduzindo o acoplamento do sistema. Basicamente, a ideia consiste em trazer a definição das dependências dos objetos para fora de suas classes, e, como o próprio nome já diz, injetá-los nestes objetos para a utilização. Isto significa que os objetos não serão responsáveis por criar ou obter as suas dependências. Em vez disso, eles a receberão a partir da injeção.

Existem várias formas de se executar este padrão, além de diversos frameworks que auxiliam o desenvolvedor a isso. A forma mais básica de demonstrar o padrão é através da injeção feita pelos construtores. Neste caso, toda a parte da definição dos serviços seria externalizado e centralizado em um único local, como por exemplo, a camada de *cross-cutting* mencionada no capítulo 3. Toda classe que tiver alguma dependência deverá possuir um construtor que espera as interfaces de todas estas dependências, por onde elas serão injetadas.

Outras opções são o padrão conhecido como *Service Locator* ou um container de injeção de dependências.

4.4 - Factory

Este é um padrão utilizado para abstrair a lógica de criação de objetos em um componente separado do cliente. Com isso, os clientes ficam livres para implementar os objetos sem precisar depender de uma implementação específica.

É um padrão cuja lógica para implementação é bem simples. Uma interface define um comportamento genérico para um grupo de implementações. A instanciação da forma concreta de uma destas implementações fica concentrada em um objeto chamado de Factory. O método que irá retornar um objeto desta implementação espera um parâmetro que indicará qual a implementação que foi solicitada.

Um exemplo simples de uso real deste padrão é para abstrair uma lógica de logs do sistema. Nesse caso, deveríamos ter uma interface *ILogger* e diversas implementações: *FileLogger*, *EmailLogger* e *DbLogger*. Para este caso específico, possuímos um *enum* no sistema que representa as possíveis formas de se armazenar um log. Para instanciar um objeto *Logger*, é solicitada à factory uma implementação, onde o *enum* apropriado foi informado como parâmetro para o método de criação. A factory saberá qual implementação deverá retornar de acordo com o enum retornado.

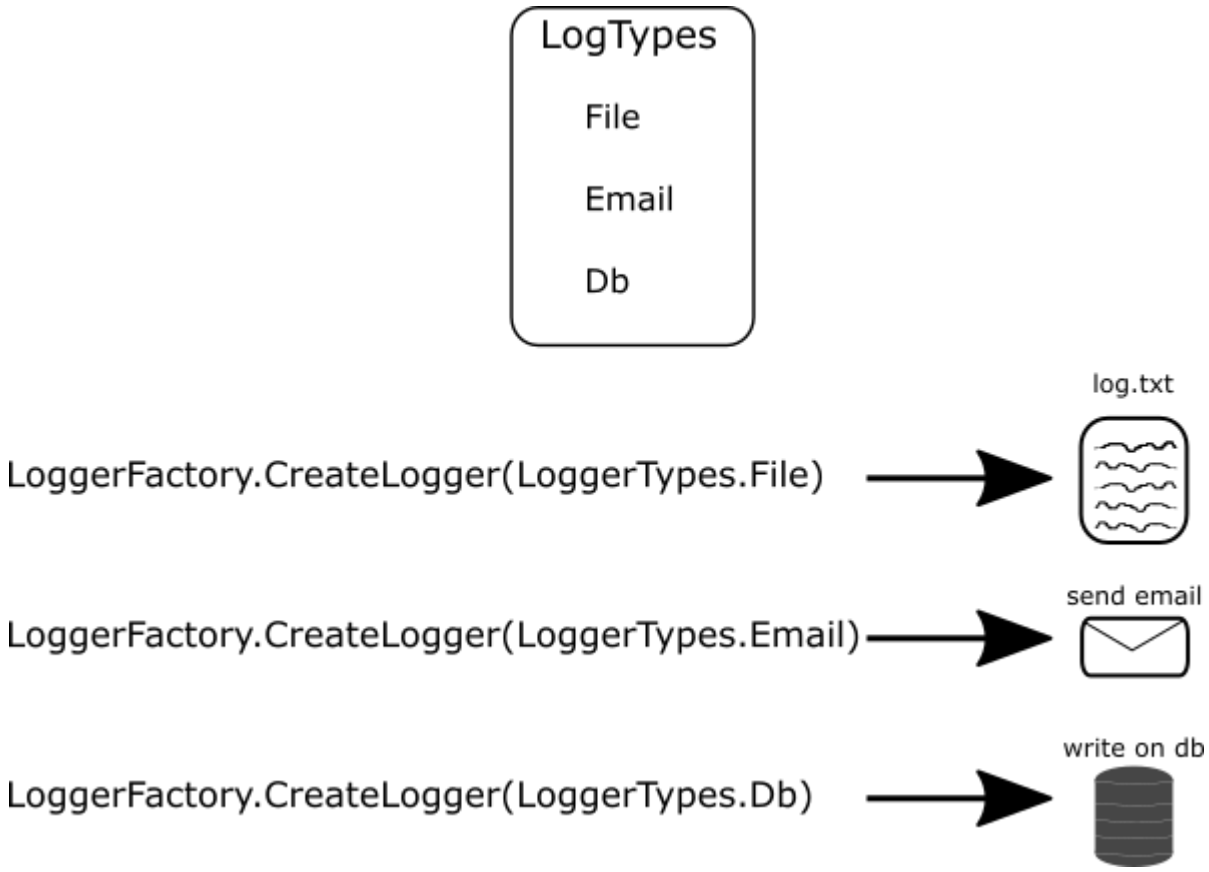

Figura 5 - Utilização da Factory para criação de um Logger

Para executar o método de log, o cliente deverá chamar um método público da interface *ILogger*. De acordo com a implementação retornada pela factory, o sistema executará a função corretamente, armazenando-o em um arquivo, enviando um email ou gravando no banco de dados.

Capítulo 5

Conceitos Básicos em Redes de Computadores - O protocolo HTTP

Antes de prosseguir com as discussões sobre a implementação do *back-end* de um sistema web, é importante definir alguns conceitos a respeito de redes e de como as comunicações são estabelecidas na web. Este é um tópico extremamente vasto, porém apenas alguns conceitos chaves que serão utilizados mais a frente serão apresentados nesta seção.

5.1 - O protocolo HTTP

O protocolo HTTP é um protocolo da camada de aplicação de redes. Ele define regras e convenções utilizadas para a comunicação entre um cliente e um servidor. Para isso, se baseia no conceito de requisições (*requests*) e respostas (*responses*). A primeira versão documentada deste protocolo é o HTTP 0.9, documentada no W3C. Foi revisto na versão HTTP 1.0, em 1996 (RFC 1945), novamente em 1999 na versão 1.1 (RFC 2616) e, mais recentemente, com a versão 2.0, em 2015 (RFC 7540). Aqui, não entraremos em detalhes das alterações mais recentes do protocolo (HTTP 2.0), uma vez que os conceitos que serão necessários para este texto têm como principal base a versão 1.1 do protocolo.

5.2 - Mensagens HTTP

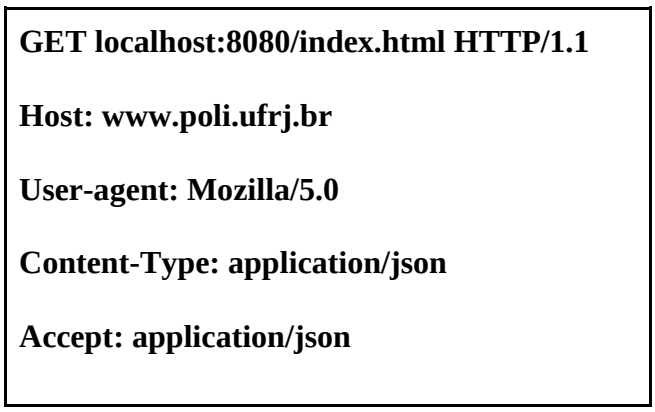

Figura 6 - Mensagem HTTP

O exemplo acima demonstra a estrutura de uma mensagem HTTP. A primeira linha é chamada de linha da requisição, e contém informações do **Verbo HTTP**, a **URI** para onde a requisição será enviada e a **versão do protocolo** sendo utilizado. Em seguida, temos 5 linhas de **cabeçalho,** chamadas de **Headers.** Eles são uma coleção de pares chave-valor que podem conter, entre outros dados, informações sobre o navegador do cliente, sobre o servidor, etc. Seguido das linhas de cabeçalho vem, opcionalmente, o **corpo** da mensagem. Ele contém a entidade que se deseja transmitir, que pode estar em diversos formatos, conforme será visto mais adiante, na subseção 5.3.

Vale notar que, como pode ser visto no exemplo, a mensagem HTTP está escrita em texto corrido com caracteres ASCII, de forma que é legível por humanos. As mensagens HTTP seguiam esse padrão até sua versão 1.1, no RFC 2616. Porém, uma das alterações definidas no RFC 7540 foi a alteração da mensagem para a forma binária. Isto torna a análise da mensagem pelos compiladores mais fácil e torna as mensagens mais compactas e menos suscetíveis a erros. Em compensação, a leitura das mensagens se torna ilegível para olhos humanos.

5.3 - MIME Types ou Media Types

Os Media Types são formados por dois identificadores, que descrevem o tipo e o subtipo de uma entidade. Eles são utilizados para definir qual é o formato em que o conteúdo no corpo da mensagem HTTP é enviado. Alguns exemplos muito comuns são:

- application/json
- application/xml
- image/png
- text/html

5.4 - Métodos (ou Verbos) HTTP

O protocolo HTTP define 8 métodos principais para lidar com a interação entre os sistemas:

OPTIONS - É utilizado para obter informações sobre as opções de comunicações disponíveis na URI contida na requisição.

GET - Utilizado para obter quaisquer informações que são representadas pela URI informada na requisição.

HEAD - É idêntico ao método GET, porém a resposta não deve conter nenhuma informação no corpo da mensagem. Apenas os metadados contidos nos headers da mensagem serão enviados na resposta para o cliente.

POST - É utilizado para enviar informações para o servidor, definindo a entidade enviada na requisição como uma nova subordinada ao recurso identificado pela URI. Um exemplo de utilização é para fornecer os dados de um formulário para o servidor.

PUT - É utilizado para que a entidade representada na requisição seja armazenado na URI informada. Na prática, isto pode providenciar dois cenários: primeiro, caso a URI aponte

para um recurso existente, a entidade informada na requisição deve ser considerada como uma versão atualizada do mesmo. Caso ele ainda não exista, e a URI seja capaz de ser definida como um novo recurso, ele permite que este recurso seja criado e armazenado nesta URI.

DELETE - É utilizado para solicitar a exclusão do recurso representado pela URI ao servidor.

TRACE - É utilizado para executar um teste de loopback no servidor. Neste teste, o servidor irá apenas refletir a mensagem de volta para o cliente. Ele é utilizado para testes e diagnósticos, para saber o que está sendo recebido pelo servidor.

PATCH - Foi definido separadamente no RFC 5789. Consiste em uma funcionalidade para modificação parcial do recurso representado pela URI.

5.5 - Códigos de estado HTTP (*Status Codes*)

Os códigos de estado são um código de 3 dígitos utilizados para representar como o servidor se comportou em relação à requisição. O primeiro dígito deste código representa uma categoria onde a resposta se enquadra, da seguinte forma:

- **1xx: Informação**. Indica que a requisição foi recebida e o processamento irá continuar.
- **2xx: Sucesso**. A requisição foi recebida, entendida e o processamento foi feito com sucesso.
- **3xx: Redirecionamento**. Indica que alguma ação posterior será necessária para finalizar o processamento da requisição.
- **4xx: Erro do cliente**. Indica que a requisição contém algum erro de sintaxe. Por isso, o processamento não poderá ser efetuado.
- **5xx: Erro do servidor**. Indica que a requisição estava correta, mas ocorreu algum erro no servidor que impossibilitou o processamento com sucesso da requisição.

Alguns dos códigos de status mais utilizados em comunicações com sistemas web são:

- **200 - OK -** Indica que o processamento da requisição foi efetuado com sucesso. O corpo da requisição deverá conter a informação resultante deste processamento, cujo conteúdo depende do verbo chamado e do processamento efetuado.
- **201 - Created -** Indica que o processamento foi efetuado com sucesso e resultou em um novo recurso disponível no sistema. Na mensagem de resposta deve ser enviado um header **Location**, que indica a URI onde este novo recurso está disponível.
- **204 - No Content -** O servidor foi capaz de processar a requisição com sucesso, mas não precisa retornar nenhuma entidade no corpo da resposta.
- **304 - Not Modified -** Utilizado para o mecanismo de cache, quando o servidor deseja informar que não houve alterações em um recurso.
- **400 - Bad Request -** O servidor foi incapaz de processar a requisição devido a um erro na sintaxe da requisição. O cliente não deve tentar enviar a requisição novamente sem que tenha sido feita alguma alteração no conteúdo da requisição.
- **401 - Unauthorized -** É necessário que o usuário esteja autenticado para que ele possa acessar aquele recurso. O cliente poderá tentar enviar a requisição novamente após adicionar a ela um header **Authorization**, contendo as informações para autenticação.
- **403 - Forbidden -** O servidor entendeu a requisição mas se recusou a completar o processamento. Esta requisição não deve ser enviada novamente, mesmo com um header **Authorization** na mensagem. Caso desejado, o servidor poderá enviar detalhes do motivo da recusa no corpo da resposta.
- **404 - Not Found -** O servidor foi incapaz de encontrar a URI informada na requisição e não existe informação se esta condição é temporária ou permanente.

- **422 - Unprocessable Entity -** Utilizado para erros de validação do sistema.
- **500 - Internal Server Error -** O servidor encontrou algum erro interno durante o processamento, que o impossibilitou de finalizá-lo com sucesso.
- **502 - Bad Gateway -** Ocorreu um erro onde o servidor recebeu uma mensagem inválida de um outro servidor necessário para completar o processamento da requisiçao.
- **503 - Service unavailable -** O servidor foi incapaz de finalizar o processamento com sucesso devido a uma sobrecarga ou uma manutenção temporária.

Para uma lista completa dos códigos de status HTTP, o RFC 2616 deve ser consultado.

5.6 - Headers

O RFC 2616 define diversos campos de Header padrões para a comunicação HTTP. Alguns dos mais utilizados são:

- **Accept -** Limita os formatos esperados para a resposta da requisição.
- **Authorization -** Contém as credenciais do agente que irá enviar a requisição, quando a autenticação é solicitada pelo servidor.
- **Content-Type -** Indica qual é o formato da entidade enviada no corpo da mensagem.
- **Location -** É usado para o redirecionamento do cliente para uma outra url necessária para finalizar a requisição. Além disso, em mensagens de resposta com status 201 (Created), este header deverá indicar a URI onde o novo recurso pode ser encontrado.

Capítulo 6

Desenvolvimento de sistemas Web - Back end. Application Programming Interfaces

6.1 - SOA ou APIs?

Tanto APIs quanto a Arquitetura Orientada a Serviços (ou SOA), oferecem funcionalidades semelhantes, porém com algumas particularidades. Tanto os *Web Services*, centrais na discussão sobre SOA, quanto as APIs são, essencialmente, interfaces para publicar um sistema *back-end*. Ambos ainda possuem grande utilização no mercado. Atualmente, o SOA vem tendo uma maior aceitação para utilizações internas, em comunicações servidor a servidor. Enquanto isso, as APIs provêm uma maneira simples e rápida de conexão com aplicativos web e mobile. Por este motivo, não entraremos em detalhes sobre a arquitetura orientada a serviços ou o protocolo SOAP neste texto. Em vez disso, focaremos na arquitetura de Web APIs, utilizando o estilo arquitetural REST.

6.2 - O que é uma api?

Uma API - Application Programming Interface, ou interface de programação de aplicativos - é o que expõe as funcionalidades de um sistema. Ela é uma interface que permite a troca de informações entre os sistemas, independentemente de suas tecnologias ou detalhes das implementações.

Uma aplicação envia uma requisição à API, informando o que ela deseja fazer. Pode ser uma consulta aos dados do sistema, um cadastro de uma nova informação, uma atualização de dados, etc. Por exemplo, em uma API de cadastramento de usuários, poderão ser solicitadas a criação de um novo usuário, uma lista de todos os usuários do sistema, as informações de um

usuário, etc. A partir daí, a API irá retornar uma resposta à aplicação, indicando se o processamento da requisição foi feito com sucesso ou não, e retornando quaisquer dados relevantes a ela.

6.3 - Por que fazer uma API?

Ao separarmos o sistema entre sua implementação em si e uma API, somos capazes de fornecer diversas implementações, para diversas tecnologias e dispositivos, que serão capazes de utilizar nosso sistema. Além disso, temos a liberdade de fazer manutenções e atualizações na implementação do sistema e disponibilizar uma versão atualizada para todos estes dispositivos sem a necessidade de qualquer atualização deste lado. Por exemplo, podem ser distribuídos aplicativos para diversos aparelhos *smartphones*, como por exemplo, dos sistemas operacionais Android, iOS ou Windows Phone. Digamos que após o lançamento do aplicativo, foi encontrado um bug em alguma funcionalidade específica da regra de negócios do sistema, que é executada no servidor. Se não tivéssemos uma API, teríamos que solucionar o problema para cada uma das distribuições do aplicativo. Mais ainda, para que um cliente que já possuísse o aplicativo instalado obtesse a versão corrigida, ele seria obrigado a atualizar o aplicativo em seu celular. Ao isolar a implementação do sistema da API, temos apenas um ponto de manutenção, e a versão atualizada estará imediatamente disponível para todos aqueles que consomem a API, sem a necessidade da atualização dos aplicativos.

6.4 - REST:

6.4.1 - Os conceitos:

REST é um acrônimo para Representational State Transfer. Ele é um estilo arquitetural que foi definido Roy Thomas Fielding em FIELDING, R.T., 2000. Um estilo arquitetural

consiste em uma abstração para indicar quais são as características de uma API. O estilo REST pode ser definido por 6 restrições:

Client-Server

É necessário que exista uma comunicação cliente servidor, onde o servidor ofereça funcionalidades e possa consumi-las.

Stateless

É necessário que a comunicação cliente-servidor não mantenha estados. Isto significa que quaisquer requisições que serão feitas a partir do cliente não devem depender de qualquer requisição anterior. Quaisquer requisições devem conter todas as informações necessárias para que o servidor efetua o processamento necessário.

Cache

É necessário que todas as respostas do servidor informem explicitamente se ela é cacheável ou não.

Sistemas em camadas

Esta restrição orienta que o desenvolvimento do sistema seja feito em camadas, como explicado no capítulo 3. Isto traz as diversas vantagens mencionadas neste capítulo, como por exemplo permitir execuções no sistema em tempo de execução sem interromper o serviço.

Código sob demanda (opcional)

Os sistemas no lado do cliente (como navegadores ou apps) podem ser atualizados sob demanda utilizando um código enviado dos provedores do serviço (servidor) aos consumidores (clientes). Isto pode ser alcançado por exemplo através de *plugins* para os navegadores ou códigos *javascript*.

Interface Uniforme

É necessário que todos os provedores e consumidores de um serviço baseado na arquitetura REST compartilhem de uma mesma interface única. Todos os componentes do sistema devem ser abstraídos em uma única interface que é exposta aos clientes. Esta restrição define algumas práticas que devem ser seguidas para a definição desta interface:

Identificação dos Recursos

Um recurso deve ser representado através da URI, e as operações nos recursos devem ser representadas pelos verbos HTTP.

Representação dos Recursos

Os dados no corpo das requisições e respostas devem ser representados por *Media Types* (ou identificador MIME), conforme mencionado no capítulo 5. Alguns dos exemplos mais comuns para comunicações em sistemas web são *application/xml* e *application/json*.

Mensagens autocontidas

As mensagens de uma comunicação usando a arquitetura REST devem ser feitas através dos *headers* e do corpo da requisição, seguindo o protocolo HTTP. As ações solicitadas são identificadas pelo verbo HTTP enviado na requisição. Aplicações REST devem possuir mensagens autocontidas, o que significa que todas as informações necessárias para o processamento da requisição pela aplicação estão reunidas em cada uma das requisições enviadas. Isto condiz com a restrição mencionada acima de que um sistema REST não deve guardar os estados da comunicação.

Hypermedia

Existe uma característica definida pelo acrônimo HATEOAS, que significa "*Hypermedia as the engine of application state*". Ela consiste em descrever para o cliente quais são as ações suportadas pelo serviço através de links e formulários enviados nas respostas às suas requisições. Na prática, podemos pensar nisso como uma navegação em um website, onde nos ao acessar uma página, nos deparamos com diversos links que nos permitem seguir para outras páginas. Em um paralelo com uma máquina de estados, podemos considerar a página atual como um estado, e os links como diversas transições para outros estados.

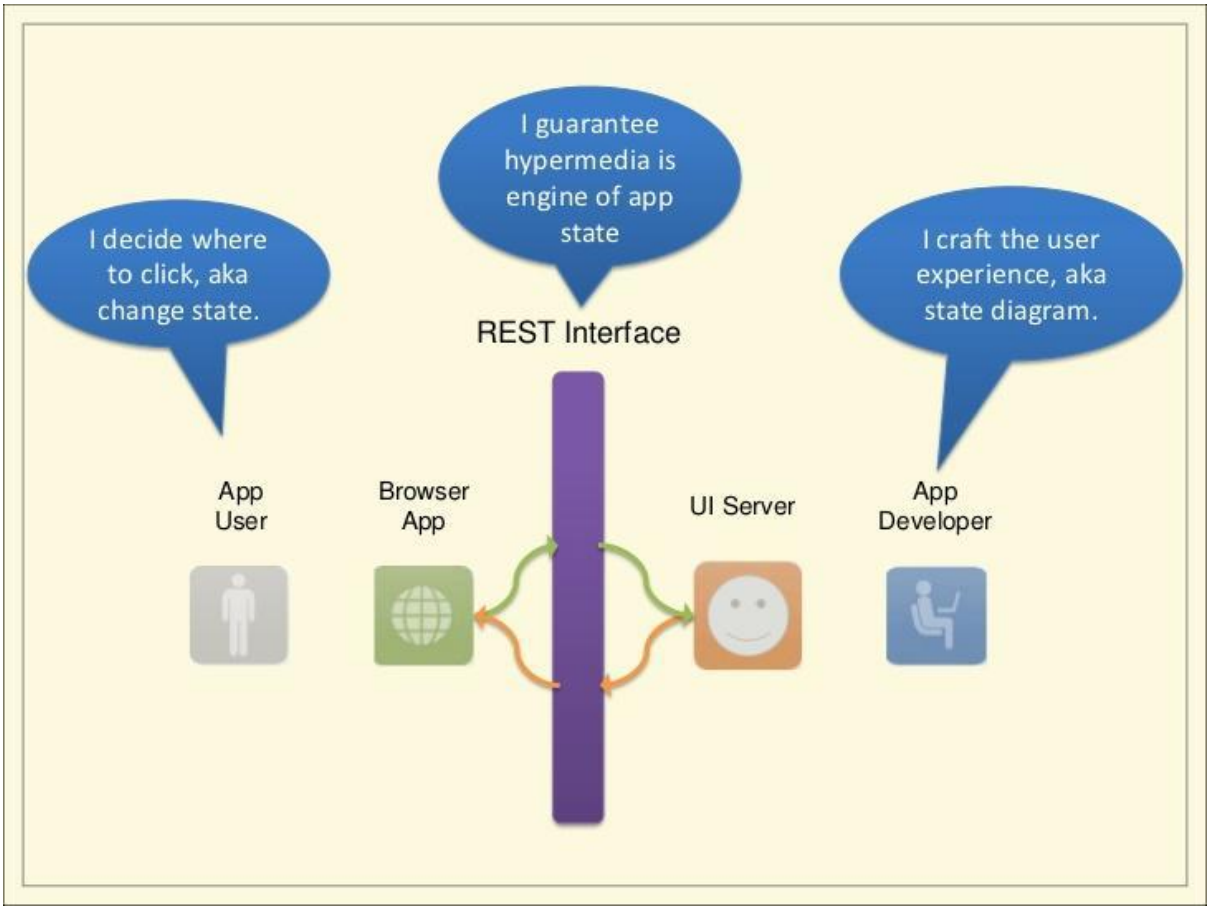

Figura 7 - Interface REST em páginas Web (APIGEE)

Esta restrição ainda é fruto de muito debate, já que muitas APIs públicas violam esta regra. Apesar de ser um modelo de imenso sucesso para a navegação na web, este modelo ainda não atende perfeitamente o projeto de APIs. Um dos principais motivos é que essencialmente as interfaces de usuário web e as APIs são diferentes, como podemos evidenciar nos seguintes diagramas:

Em páginas web, existe uma interface REST separando o desenvolvedor - que cria a máquina de estados para a experiência do usuário - e o usuário final do sistema - que decide onde clicar e como navegar através da máquina de estados. Dessa forma, podemos afirmar verdadeiramente que a *hypermedia* está atuando como o motor da aplicação, já que estes links estão sendo utilizados para a navegação na página.

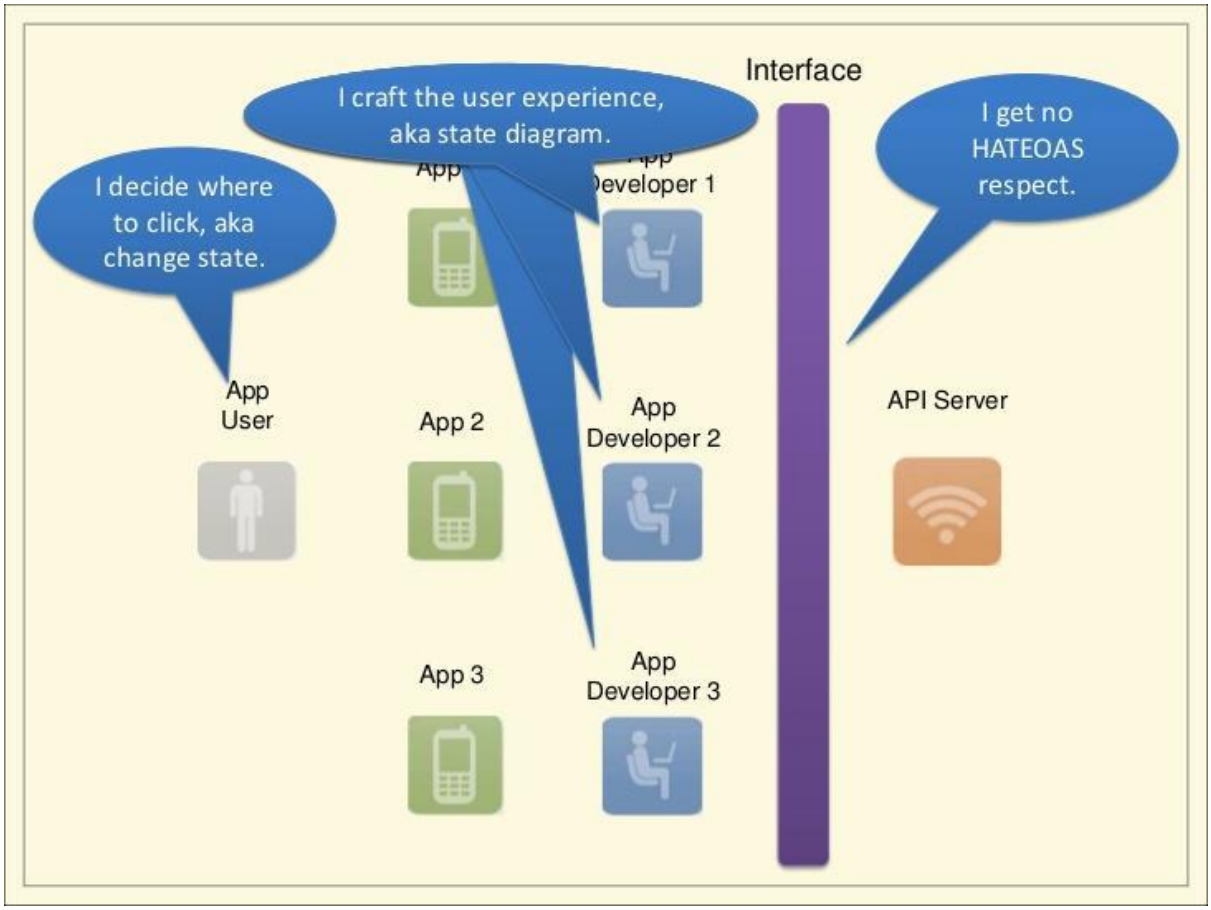

Figura 8 - Interface REST em APIs

Já no caso das APIs, não existe essa separação entre o desenvolvedor e o usuário final através de uma API REST. Dessa forma, os links não estariam sendo disponibilizados para que o usuário final possa navegar através da máquina de estados. Em vez disso, os links seriam entregues pelo servidor da API para que vários desenvolvedores criassem suas aplicações, que então seriam disponibilizadas para um usuário final. Este usuário é quem decidiria como

navegar através dos estados da aplicação. Os links entregues do servidor da API para os desenvolvedores não estariam atuando como o real motor da aplicação.

6.4.2 - Melhores práticas para desenvolvimento de uma API RESTful

Antes de iniciar as discussões a respeito de boas práticas, vale lembrar que nenhuma das informações deve ser levada como uma regra. Todas são baseadas em estudos, hábitos de desenvolvimento de pessoas com experiência e histórias de APIs de sucesso. As ideias de boas práticas devem servir como um guia que indica como começar a desenvolver. Mas toda prática é sempre passível de melhorias e aperfeiçoamentos, e é comum que todos estes padrões continuem evoluindo e sofrendo alterações com o tempo.

O mais importante para iniciar o desenvolvimento de uma API é ter sempre em mente que ela está sendo feita para ser utilizada por clientes. Portanto, é sempre de extrema importância que qualquer decisão de projeto tomada, seja feita pensando na simplicidade, clareza e facilidade de integração. De nada adianta ter uma API repleta de funcionalidades e das tecnologias mais modernas, se nenhum cliente é capaz de entendê-la e utilizá-la. A maior indicação de sucesso para uma API é a rapidez com que os clientes são capazes de integrar e começar a usufruir de suas funcionalidades.

1. **Navegação pela URL**

A API deve ser totalmente explorável através da URL. Todos os recursos que podem ser acessados na API devem ser definidos através da URL.

2. **Recursos**

As funcionalidades da API devem ser disponibilizadas através de **recursos**, que são representados a partir de substantivos que fazem sentido para um cliente consumindo a API. As operações que podem ser feitas nos recursos devem ser indicadas através dos verbos HTTP e **não através de verbos na url.** Isto aumenta muito a simplicidade de entendimento da API, uma vez que padroniza todas as chamadas às operações. Verbos na URL podem acabar saindo

de controle conforme a API cresce, uma vez que os recursos estariam sendo adicionados conforme as necessidades. Em uma API que consulta dados dos usuários de uma rede social, por exemplo, poderíamos acabar em uma situação como essa:

- /GetUser
- /GetUserAddress
- /GetUserFriends
- …

Ao padronizar as URLs através dos substantivos, para cada recurso desejado, temos duas URLs base, da seguinte forma:

- /User
- /User/{id}

Onde {id} deve ser substituído por um código identificador do recurso.

Unindo estas duas URLs base com os 5 principais verbos HTTP - GET, POST, PATCH, PUT e DELETE, temos o que é chamado de operações CRUD - Create, Read, Update, Delete - fornecendo o seguinte leque de funcionalidades para cada recurso:

Tabela 1 - Métodos HTTP em uma API

Recurso	**POST**	**GET**	**PATCH**	**PUT**	**DELETE**
/User	Cria um usuário	Lista todos os usuários	Atualiza parcialmente todos os usuários*	Atualiza todos os usuários*	Exclui todos os usuários*
/User/1234	Retorna uma mensagem de erro	Obtém os detalhes do usuário de id 1234	Atualiza parcialmente o usuário 1234 se ele existir. Se não, retorna uma mensagem de erro	Atualiza o usuário 1234 se ele existir. Se não, retorna uma mensagem de erro	Exclui o usuário de código 1234.

Vale notar que as funcionalidades de atualização e exclusão em massa (PATCH /Users, PUT /Users e DELETE /Users) muitas vezes não são implementadas, e podem retornar uma mensagem de erro. Isso ocorre pois são operações irreversíveis e executá-las em todas as entradas do sistema pode causar danos irreversíveis ao sistema se não executados com extremo cuidado. A partir daqui, deixaremos de considerar estas funcionalidades e preferiremos seguir com uma abordagem de retornar erro neste caso.

Requisições para criação e atualização de um recurso (POST, PUT e PATCH) devem retornar sempre uma versão atualizada do recurso na resposta. Além disso, requisições de criação devem retornar o status HTTP 201 (Created), além do header Location indicando a URL onde o recurso se encontra.

3. Nomeando os recursos

Uma dúvida comum é quanto aos nomes dos recursos, se devem ser dados no singular ou no plural. É comum encontrarmos APIs de sucesso seguindo qualquer um destes padrões. A regra mais importante é ser consistente dentro de sua aplicação. É de grande importância seguir um padrão, seja ele os nomes no singular ou no plural. Neste texto, será adotado o padrão de nomes no plural, sem se preocupar com palavras cujo plural é irregular (por exemplo, utilizar *persons* em vez de *people* para um recurso que representa pessoas).

4. Relações entre os recursos

Quando existe uma relação mais complexa entre os recursos, onde, por exemplo, um usuário possui uma coleção de endereços, estes recursos podem ser obtidos ao aninhar as relações na URL, da seguinte forma:

Tabela 2 - Métodos HTTP representando relações entre recursos em uma API

Recurso	**POST**	**GET**	**PATCH**	**PUT**	**DELETE**
/Users/1234/Addresses	Cria um endereço para o usuário 1234	Lista todos os endereços do usuário 1234	Erro	Erro	Erro
/Users/1234/Addresses/123	Erro	Obtém os detalhes do endereço 123 para o usuário 1234	Atualiza parcialmente o endereço 123 do usuário 1234, caso existir	Atualiza o endereço 123 do usuário 1234, caso existir	Exclui o endereço 123 do usuário 1234, caso exista

5. Ações que não se encaixam entre operações de CRUD

A ideia geral é sempre tentar mapear estas ações como um subrecurso. Por exemplo, na API do Github(GITHUB API), a ação de adicionar ou remover um recurso *gist* aos favoritos pode ser mapeado da seguinte forma:

- PUT /gists/{id}/star
- DELETE /gists/:id/star

Caso não seja possível mapear de forma RESTful, como por exemplo, um endpoint para busca em diversos recursos da api, não há problema em fugir do padrão, desde que isto seja bem documentado e faça sentido do ponto de vista do cliente.

6. SSL

É sempre essencial utilizar SSL (JUNIOR, H.) para garantir a segurança nas comunicações. Atualmente, o acesso à Internet é feito de diversos lugares públicos, muitos dos quais não possuem qualquer tipo de segurança ou criptografia. Utilizar segurança na camada de transportes protege a sua API contra *eavesdropping* (KUROSE, J.F., ROSS, K.W.) e simplifica os esforços no desenvolvimento da autenticação do sistema.

7. Documentação

Esta é, discutivelmente, a parte mais importante de uma API. É extremamente desvalorizada pelos desenvolvedores, que muitas vezes negligenciam uma boa documentação

apenas para trabalhar cada vez mais na parte técnica da API. Em compensação, em conversas com pessoas da área de projeto e de negócios, a importância da documentação é muito mais reconhecida. Uma API, independentemente de ter sido desenvolvida nos melhores padrões ou de conter as melhores funcionalidades, só possui um real valor se houver clientes integrados e satisfeitos, de forma que ela possa continuar crescendo e evoluindo. Sem uma boa documentação, será difícil atrair novos clientes, além de aumentar muito as chances de que suas integrações com o sistema sejam feitas de forma incorreta, o que aumenta enormemente as chances de problemas e a necessidade de constantes suportes técnicos.

8. Versionamento

O versionamento da API é de grande importância. Ele dá a flexibilidade de se implementar alterações na API sem afetar os clientes, o que proporciona muito mais agilidade no crescimento da API.

O ideal é manter uma política de atualizações claras para todos os clientes da API, onde as alterações sejam feitas oferecendo um período de transição onde as versões antigas também são suportadas.

Já a respeito de como o versionamento é feito, não existe uma regra universal. Para garantir que a API seja totalmente explorável através da URL, de acordo com os princípios de uma arquitetura REST, este versionamento deveria ser feito através da URL, como um dos exemplos a seguir:

https://api.meusistema.com/v1.0/Users

https://api.meusistema.com/1.0/Users

Porém, também é comum se encontrar o versionamento feito nos headers, ou até mesmo uma abordagem mista. A API de uma grande empresa de pagamentos online, *Stripe*, é um exemplo dessa última abordagem, onde as *major versions* são feitas através da URL, porém

minor versions e *bug fixes* podem ser escolhidas através de um header customizado informando a data da versão da API desejada (STRIPE API).

9. Filtros, ordenamento e limitação dos campos para os resultados

Para aumentar a flexibilidade e dar aos clientes a opção de customizar os dados que serão recebidos pela API, é muito interessante acrescentar opções de filtros, ordenamento e buscas nos resultados de uma requisição. Estas customizações podem ser feitas a partir dos parâmetros de *query*. Estes parâmetros são informados ao final da url, após inserir um sinal de interrogação (“?”) que indica o início destes parâmetros.

Os **filtros** podem ser implementados de forma intuitiva, simplesmente definindo o nome do parâmetro como o nome do filtro. Por exemplo, para filtrar apenas os clientes do sexo feminino recebidos de uma API, pode ser feita uma chamada da seguinte forma:

GET /Clients?gender=female

O **ordenamento** pode ser feito através de um parâmetro *sort* inserido na url. As lógicas aceitas por este parâmetro dependem das funcionalidades da API e devem ser bem documentadas. Um sinal de menos antes do valor escolhido pode ser utilizado para indicar ordem decrescente. Além disso, ordenamentos complexos podem ser feitos ao informar mais de um argumento para o parâmetro *sort*, separados por vírgula. Alguns exemplos são:

GET /Clients?sort=-name

Ordena os clientes em ordem decrescente através de seus nomes.

GET /Clients?sort=-name,created_at

Ordena os clientes em ordem decrescente através de seus nomes. Nomes que começam com a mesma letra são ordenados de forma crescente a partir da data de criação dos clientes.

A opção de **limitar quais campos serão retornados** pela API também é uma opção que traz flexibilidade, reduz o tráfego na rede e agiliza as chamadas à API. Isto pode ser feito

informando um parâmetro *fields* nos parâmetros de query, indicando os campos que serão recebidos separados por vírgula, da seguinte forma:

GET /Clients?fields=Name,Address,Phone

10. Formato das respostas

Os principais formatos utilizados para o retorno dos dados em uma API são XML e JSON. O suporte ao XML ainda é bastante utilizado, porém já existem diversos autores que começam a recomendar que novas APIs não deem mais este suporte. A utilização do formato JSON já ultrapassou com grande margem a do XML, como pode ser verificado no gráfico a seguir, obtido a partir do *Google Trends*, comparando a popularidade das buscas entre os termos "*xml api*" e "*json api*":

Figura 9 - Comparação de popularidade de busca entre os termos "xml api" e "json api"

Este gráfico não mostra valores absolutos de número de vezes que o termo foi buscado. Em vez disso, faz uma comparação entre a popularidade da busca de cada um dos termos. Podemos ver que em Julho de 2016, ponto final do gráfico, o termo *"json api"* foi 4 vezes mais popular em buscas do que o termo *"xml api"* (o primeiro se encontra no valor 100 enquanto o segundo marca 25 na escala vertical).

11. Padrões de nomenclatura dos campos das mensagens

Existem 3 possíveis formatos para se utilizar na nomenclatura dos campos do corpo das mensagens:

- *snake_case* - Palavras separadas por um "_", todas iniciadas em letras minúsculas.
- *camelCase* - Primeira letra do campo minúscula, primeira letra das palavras seguintes maiúsculas, sem separação entre as palavras
- *PascalCase* - Primeira letra de cada palavra no nome do campo com letra maiúscula, sem separação entre as palavras.

Não existe uma regra universal a respeito de qual destes utilizar, podendo ser encontradas APIs de sucesso em cada um destes padrões. Porém, vale notar que as convenções do *Javascript* utilizam o padrão *camelCase*. Desta forma, se o formato padrão de retorno da API for *JSON*, é uma boa recomendação seguir esta nomenclatura.

12. Compressão

O suporte ao gzip pode reduzir consideravelmente o tráfego para efetuar chamadas à API, onde a compressão da resposta pode economizar até 80% de banda.

13. Paginação

Conceito extremamente importante para as respostas da API. O RFC 5988 define que a paginação deve ser feita através da utilização do header *Link*. Um bom exemplo de implementação desta forma de paginação é a API do Github. As informações de paginação são inseridas da seguinte forma:

```
Link: <https://api.github.com/user/repos?page=3&per_page=100>; rel="next",
<https://api.github.com/user/repos?page=50&per_page=100>; rel="last"
```

Onde os possíveis valores para *rel* são:

Tabela 3 - Valores indicando a relação entre os links para a paginação

Valor	**Descrição**
next	Link para a próxima página
last	Link para a última página
first	Link para a primeira página
prev	Link para a página anterior

14. Proteção contra abusos

Para evitar que um cliente chame de forma indiscriminada a API, é padrão utilizar um contador do limite de requisições em um dado período de tempo. Caso aquele cliente específico tenha excedido o número máximo permitido de requisições, a API deve retornar o código 429 (*Too Many Requests*). Para manter o cliente informado em todas as requisições a respeito do número máximo permitido, podem ser informados três headers na resposta:

- *X-Rate-Limit-Limit* - Número total de requisições permitidas em um dado período
- *X-Rate-Limit-Remaining* - Número de requisições restantes para o cliente no período atual
- *X-Rate-Limit-Reset* - Número de segundos restantes no período atual, antes de reiniciar o contador de requisições.

15. Cache

Existem dois métodos de *caching* já incorporados no HTTP: *ETag* e *Last-Modified*. Dentre estes métodos, o *ETag* possui a vantagem de não depender do formato de datas enviado pelo cliente. Para implementar o *Last-Modified*, a API deveria estar preparada para receber qualquer um dos 3 diferentes formatos de data aceitos pelo protocolo HTTP. Por isso, é recomendado utilizar o método *ETag*.

Para utilizar este método, é necessário incluir nas respostas da requisição um header *ETag* contendo um *hash* dos dados retornados pela API. O cliente deverá guardar este valor. Na requisição seguinte, deverá enviar este *hash* em um *header If-None-Match*. Se os dados não sofreram nenhuma alteração, a API deverá retornar o código HTTP 304 (*Not Modified*).

16. Tratamento de Erros

As mensagens de erro retornadas pela API devem ser representadas da mesma forma que as entidades do sistema, porém com um conjunto próprio de propriedades que indicam claramente o motivo do erro. O código HTTP retornado deve também representar corretamente o erro, de acordo com o protocolo HTTP e a lista de códigos mencionada no capítulo 5. Os erros normalmente podem ser separados em duas categorias - códigos 4xx para erros do cliente e erros 5xx para erros do servidor.

O objeto de erro deve conter um campo contendo uma mensagem sobre o erro, um campo contendo uma descrição mais detalhada quando necessário, e um campo contendo um código de erro customizado. É importante que exista na documentação uma tabela contendo descrições sobre estes códigos de erros, além de uma orientação de como prosseguir para solucioná-los. Não existe uma regra a respeito de como serão definidos estes códigos de erro. Isto fica a cargo do arquiteto da API.

Algumas APIs incluem ainda um link apontando para a página da documentação que explica como solucionar o problema encontrado. Esta é uma abordagem interessante para aumentar as chances de que o cliente possa resolver o problema facilmente e por conta própria.

Para erros de validação dos verbos POST, PATCH e PUT, é interessante ainda adicionar uma coleção detalhando os erros, indicando quais campos o causaram e o motivo. Dessa forma, podemos ter o seguinte cenário:

```
1. {
2.     "code": "VAL001",
3.     "message": "Validation Error",
4.     "errors": [
5.         {
```

```
            "code": "USE001",
            "field": "username",
            "message": "The username cannot contain special characters"
        },
        {
            "code": "USE002",
            "field": "password",
            "message": "The password cannot be blank"
        }
    ]
}
```

6.5 - Visão de negócios

Na seção anterior, foram apresentadas diversas funcionalidades que agregam grande valor à API, ao fornecer grande flexibilidade, segurança e facilidade na integração. Entretanto, é de grande importância ter em mente de que a API é um sistema vivo, que vai invariavelmente evoluir e amadurecer com o tempo. Dessa forma, deve-se sempre balancear aquilo que é um requisito indispensável para o funcionamento da API daquilo que é uma melhoria. Com isso em mente, deve-se pensar em lançar a API com todas as funcionalidades que são realmente *essenciais* para o seu correto funcionamento. A partir daí, pode-se acrescentar cada um dos outros pontos como *melhorias técnicas*, e implementá-los após o sistema já estar no ar e disponível para os clientes. Com isso, pode-se ganhar *feedbacks* a respeito do estado atual e de ideias para melhorias no sistema. Além disso, os clientes já poderão iniciar a integração e ir usufruindo das funcionalidades existentes, e no caso de uma API para o meio corporativo, as receitas já começarão a ser obtidas, podendo ser reinvestidas na melhoria e no crescimento da API.

Além das boas práticas relacionadas aos conceitos técnicos do desenvolvimento das APIs, devemos ainda considerar as boas práticas em termos de negócios. Algumas delas são:

1. **Possibilitar o autosserviço dos desenvolvedores e usuários da API**

O autosserviço consiste em proporcionar uma experiência em que o cliente poderá ser independente em todas as fases da integração. Desde o momento de se cadastrar para obter

credenciais de autenticação, até a integração e solução de quaisquer problemas, não é necessário que o usuário tenha que entrar em contato com nenhum órgão de suporte técnico ou comercial. Se uma API é madura o suficiente para isto, invariavelmente ela atrairá mais clientes e terá grande popularidade. Para isso, é necessário que a API esteja muito bem documentada e suas chamadas sejam intuitivas. Além disso, é necessária uma infraestrutura onde o cliente poderá contratar os serviços de forma automática, como um portal onde ele possa solicitar o cadastramento para os serviços.

2. **Priorizar as necessidades e preferências dos desenvolvedores clientes**

Ter um mecanismo capaz de receber *feedbacks* daqueles que utilizam o seu serviço é essencial. Apenas ao ouvir as opiniões de quem integrou com o sistema será possível realmente ter certeza se os objetivos desejados foram atingidos. A integração é simples? A documentação é clara? O que pode melhorar? Quais recursos poderiam ser disponibilizados? Qualquer desenvolvedor gostaria de ter um canal de comunicação com aqueles que oferecem um serviço através do qual estas sugestões possam ser feitas. Dessa forma é garantido que o seu sistema se tornará cada vez mais popular e terá cada vez mais pessoas utilizando-o.

3. **Incentivo à colaboração da comunidade de desenvolvedores**

Uma empresa que disponibiliza SDKs, plugins ou outros sistemas em código aberto, incentivando a colaboração da comunidade pública de desenvolvedores, ganha imensa credibilidade. Em primeiro lugar, ao disponibilizar um bom código de forma pública, os desenvolvedores poderão analisar o código e ter uma ideia dos padrões de qualidade da empresa. Em segundo lugar, estas ferramentas auxiliam muito os desenvolvedores em suas integrações. E por último, porém não menos importante, eles poderão contribuir para que as ferramentas se tornem cada vez melhores.

Capítulo 7

Conceitos Básicos de Banco de Dados

A área de estudos sobre banco de dados é também uma área extremamente vasta, digna de diversos livros para aqueles que desejem se aprofundar. Alguns exemplos de referências são RAMAKRISHNAN, R., GEHRKE, J., 2003 e SILBERSCHATZ, A. KORTH, H., et al. Entretanto, para iniciar o desenvolvimento do back end de um sistema web é necessário ao menos um conhecimento básico do engenheiro nesta área, e por isso iremos destacar aqui os conceitos mínimos necessários para este desenvolvimento.

7.1 - Banco de dados relacional

A grande maioria dos sistemas que precisam armazenar e manipular informações em um banco de dados utiliza a estrutura dos bancos relacionais. Sua ideia foi proposta em 1970 por Edgar F. Codd, no artigo “A Relational Model of Data for Large Shared Data Banks” (CODD, E.F., 1970).

A estrutura de um banco de dados é basicamente definida através de **tabelas**, onde cada linha representa uma entrada de um recurso desejado. Cada coluna representa uma característica deste recurso. Fazendo um paralelo com a programação orientada a objetos, podemos considerar que uma tabela representa uma classe do sistema. As colunas da tabela representam as propriedades do objeto, e as linhas representam uma instância, ou objeto, desta classe.

Para estabelecer a ligação entre as tabelas, dois conceitos são essenciais:

- As **chaves primárias** é um identificador único do elemento representado pela linha da tabela.

- As **chaves estrangeiras** relacionam um objeto da tabela ao outro. São valores guardados em uma coluna da tabela, que apontam para uma linha de outra tabela através da chave primária desta última.

Um exemplo da estrutura de tabelas, chaves primárias e chaves estrangeiras pode ser observado no diagrama a seguir:

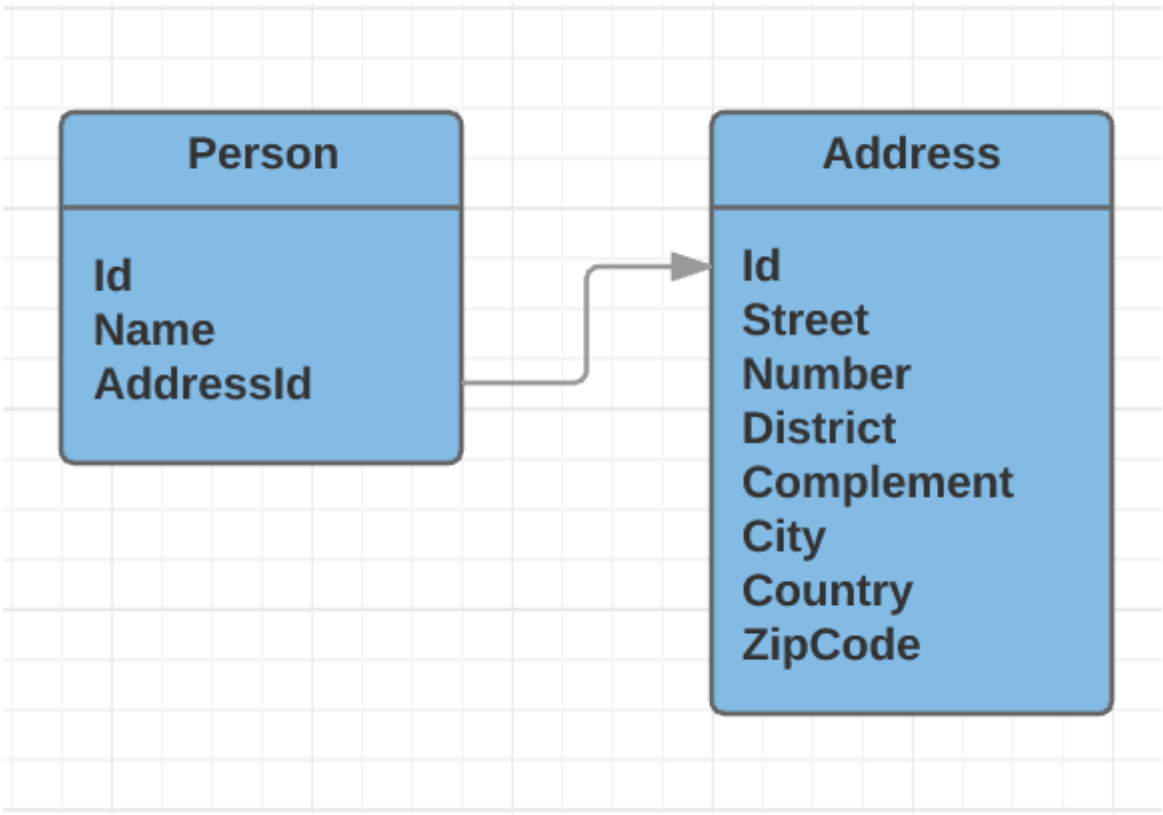

Figura 10 - Diagrama de classes representando um relacionamento entre tabelas

Neste caso, os campos *Id*, *Name* e *AddressId* são as colunas da tabela *Person*. Ao inserir dados nesta tabela, formamos suas linhas. Os campos *Id* representam as chaves primárias. O campo *AddressId* representa uma chave estrangeira que relaciona a tabela *Person* com a tabela *Address*.

Os bancos relacionais possuem uma linguagem própria para a execução de buscas nos dados de uma tabela: *query language*. Uma *query*, ou consulta, é uma expressão onde podem ser inseridos critérios de busca para obter os dados de um banco de dados através de uma ou

mais tabelas simultaneamente. Os resultados da busca são retornados também no formato de uma tabela, que não está armazenada fisicamente no banco. Ao retornar desta forma, os resultados podem ser utilizados como entrada para uma nova busca, caracterizando as chamadas *subqueries*.

Além das buscas, são ainda definidas mais três formas de acesso aos dados, relacionadas à escrita no banco de dados. São elas a **inserção**, a **exclusão** e a **modificação** dos dados.

Alguns exemplos de sistemas gerenciadores de bancos de dados relacionais são Sql Server, MySql, PostgreSql e Oracle.

7.2 - Banco de dados não relacionais

Além do conceito de banco de dados relacionais, existe também uma grande vertente que vem apoiando a utilização de bancos não relacionais. Nesta estrutura, os dados são armazenados em documentos completos, como se fossem arquivos. Um exemplo de utilização desta forma que vem tomando muita força recentemente é para o armazenamento dos logs dos serviços. Além disso, ainda existe um grande movimento para o aumento no uso desta forma de banco de dados nos sistemas inteiros.

Neste texto não entraremos detalhes destes bancos de dados. Para mais informações, existem diversos livros, artigos e discussões que podem ser consultados, como por exemplo BUCKLER, C., 2015, FOWLER, M., 2013 e PARKER, Z., POE, S, et. al.

Capítulo 8

Tutorial para a criação de um sistema

Como prova de conceito, este capítulo apresentará um passo a passo para a criação back end de um sistema web simples. Para isso, serão utilizadas a linguagem *C#* e o *framework .Net*, além do sistema gerenciador de banco de dados *SQL Server 2016 Express Edition*. A *IDE* utilizada será o *Microsoft Visual Studio Community 2015*.

O sistema será uma agenda telefônica simples, que simplesmente recebe e armazena o nome e o telefone de uma pessoa. Conterá ainda, funcionalidades de listar todos os nomes existentes em seus registros ou obter um registro específico a partir de um identificador único.

8.1 - Criação do Sistema

8.1.1 Criação da arquitetura do sistema

Para começar o desenvolvimento, deveremos criar o primeiro projeto e a solução. No *Visual Studio,* clique em *File -> New -> Project.* Escolha o *template Class Library*. O nome do projeto será *MinhaAgenda.Data*, e o nome da solução será *MinhaAgenda*.

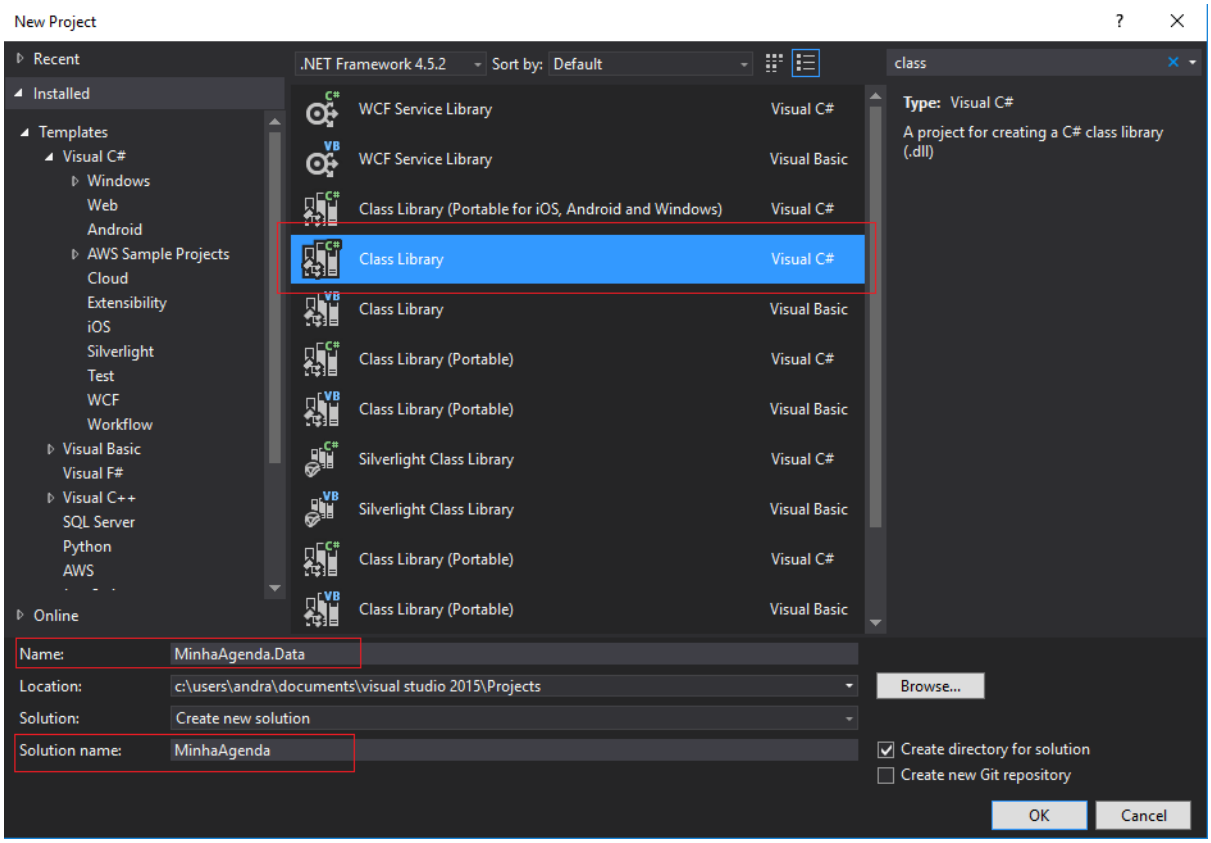

Figura 11 - Criação de uma solução e um projeto no Visual Studio 2015

Na solução criada, deverão ser incluídas 4 pastas que representarão as camadas do sistema. Os nomes das pastas devem ser os seguintes:

"1 - Presentation"

"2 - Services"

"3 - Data"

"4 - Cross Cutting"

As pastas das soluções do *Visual Studio*, diferentemente das pastas dos projetos, não são criadas fisicamente no diretório das soluções. São apenas virtuais, para dar uma organização melhor à solução. Fisicamente, será criada uma pasta para cada projeto da solução. Já as pastas criadas dentro do projeto, serão também criadas fisicamente nos diretórios dos projetos.

O projeto *MinhaAgenda.Data* deve ser movido para a pasta "*3 - Data*". Este projeto será responsável pelo acesso ao banco de dados.

Na pasta *"1 - Presentation"*, devem ser adicionados mais dois projetos. O primeiro será uma *DLL* (*Class Library*), chamada *MinhaAgenda.Models*, que conterá os modelos para entrada e saída de dados da API, além de suas validações. O segundo será chamado de *MinhaAgenda.API*. Este projeto é do tipo *ASP.NET Web Application*, contendo a implementação da API. Na criação deste último, deverá ser escolhido o template *Web API*, e a configuração de autenticação deve ser alterada para "*No Authentication*".

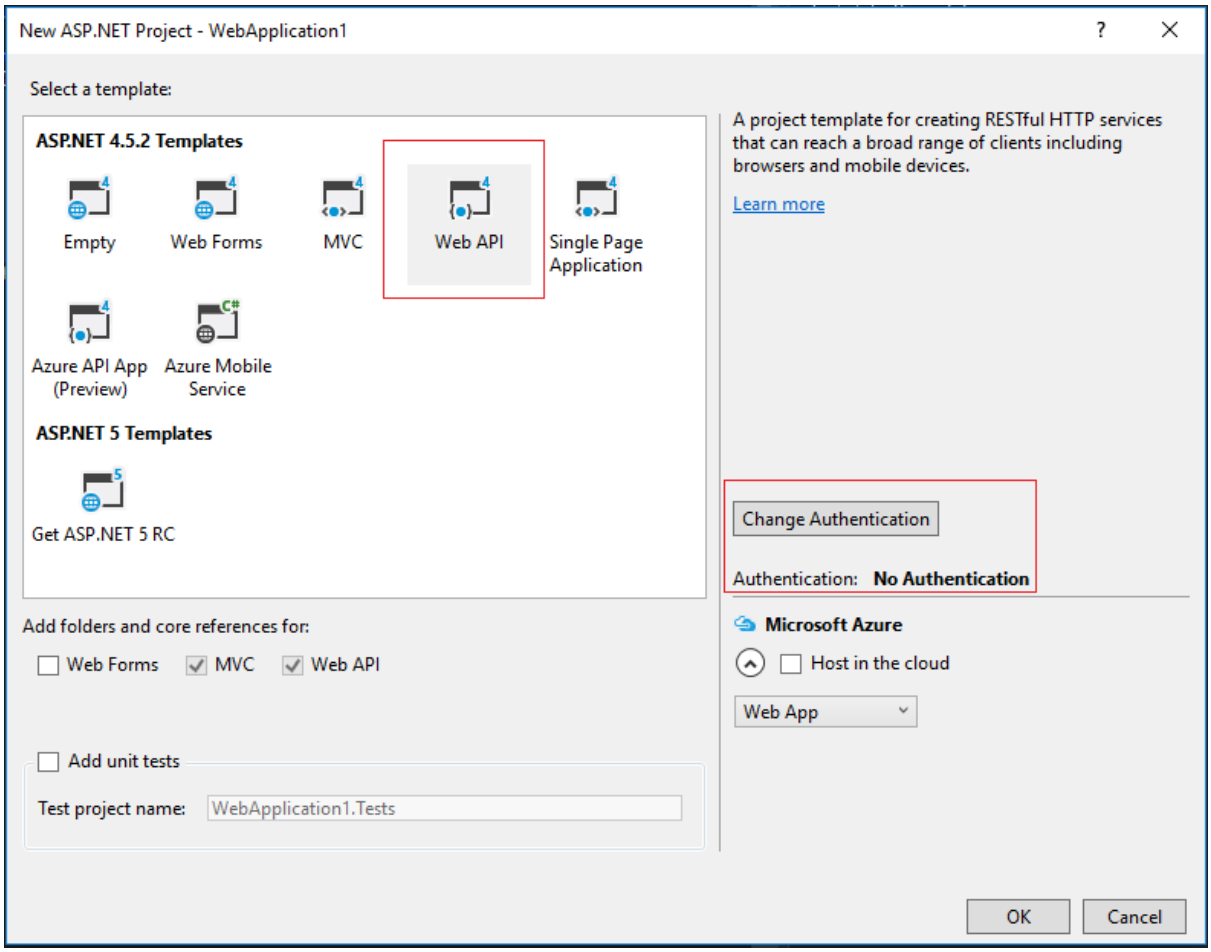

Figura 12 - Criação de um projeto Web API

Na pasta *"2 - Services"*, deverá ser criada uma *DLL* chamada de *MinhaAgenda.Services*. Este projeto é responsável pela camada de negócios.

Na pasta "*4 - Cross Cutting"*, deve ser criada uma *DLL* chamada de *MinhaAgenda.IoC*. Este projeto será responsável pelas configurações iniciais para a injeção de dependências.

A estrutura das pastas e projetos deverá ficar assim:

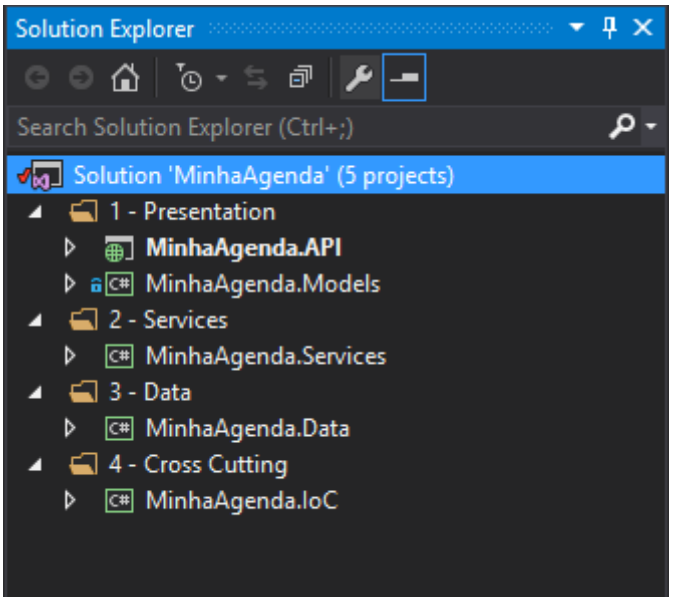

Figura 13 - Estrutura de pastas e projetos da solução

8.1.2 - Implementando o sistema

Vamos começar pela criação da entidade do banco de dados. No projeto *MinhaAgenda.Services*, crie uma pasta chamada *Entities*. Dentro dela, crie uma classe chamada *Person*. Ela será uma classe bem simples, com apenas três propriedades: *Name*, *Phone*, e a chave primária, *Id*.

```
1. public class Person
2. {
3.     public int Id { get; set; }
4.     public string Name { get; set; }
5.     public string Phone { get; set; }
6. }
```

Em seguida, criaremos todas as interfaces de comunicação com as outras camadas. Para isso, adicione neste mesmo projeto uma pasta *Interfaces* contendo duas subpastas: *Repositories* e *Services*.

Na pasta *Repositories*, crie uma interface pública chamada *IPersonRepository*, com três métodos:

```
1.      public interface IPersonRepository
2.      {
3.          void InsertPerson(Person person);
4.          Person GetPersonById(int id);
5.          IEnumerable<Person> GetAllPersons();
6.      }
7.
```

Ao criar esta interface, a *IDE* irá mostrar um erro dizendo que o identificador "*Person*" não foi encontrado. Para resolver, basta parar o cursos em cima deste identificar e apertar "*ctrl* + .". Com isso, o Visual Studio irá sugerir que o *namespace* apropriado seja adicionado

Na pasta *Services*, crie uma interface pública chamada *IPersonService*, também com três métodos:

```
1.      public interface IPersonService
2.      {
3.          Person CreatePerson(Person person);
4.          Person GetPersonById(int id);
5.          IEnumerable<Person> GetAllPersons();
6.      }
```

Em seguida, será criada a implementação do serviço, que contém a parte principal de negócios do sistema. Para esse sistema, esta implementação será bem simples, contendo apenas as chamadas aos repositórios. Na raiz do projeto *MinhaAgenda.Services*, deve ser criada a classe *PersonService*, que implementa a interface *IPersonService*, da seguinte forma:

```
1.  public class PersonService : IPersonService
2.      {
3.          private readonly IPersonRepository _personRepository;
4.          public PersonService(IPersonRepository personRepository)
5.          {
6.              _personRepository = personRepository;
7.          }
8.          public Person CreatePerson(Person person)
9.          {
10.             _personRepository.InsertPerson(person);
11.             return person;
12.         }
13.         public IEnumerable<Person> GetAllPersons()
14.         {
15.             var persons = _personRepository.GetAllPersons();
16.             return persons;
17.         }
18.         public Person GetPersonById(int id)
19.         {
20.             var person = _personRepository.GetPersonById(id);
21.             return person;
22.         }
23.     }
24.
```

No construtor desta classe, recebemos uma implementação da interface *IPersonRepository*, que será recebida através da injeção de dependências.

Seguindo para a camada de dados, utilizaremos o *framework Dapper* (DAPPER) para gerenciar o acesso ao banco de dados. Para incluí-lo no projeto, deve ser utilizado o gerenciador de pacotes *NuGet* (NUGET).

Clique com o botão direito no projeto *MinhaAgenda.Data*, em seguida em "*Manage NuGet Packages*". Na aba *Browse*, procure por *Dapper* e clique em *Install* para adicionar ao projeto.

Apenas uma pasta, chamada *Repositories*, deve ser adicionada ao projeto. Ela conterá uma única classe chamada *PersonRepository*:

```
1.     public class PersonRepository : IPersonRepository
2.     {
3.         private readonly IDbConnection _dbConnection;
4.         public PersonRepository()
5.         {
6.             string connectionString =
    @"Server=localhost\SQLEXPRESS;Database=AgendaDb;Trusted_Connection=True;";
7.             _dbConnection = new SqlConnection(connectionString);
8.         }
9.         public IEnumerable<Person> GetAllPersons()
10.        {
11.            string query = "SELECT * FROM Person";
12.            _dbConnection.Open();
13.            var persons = _dbConnection.Query<Person>(query);
14.            _dbConnection.Close();
15.            return persons;
16.        }
17.        public Person GetPersonById(int id)
18.        {
19.            string query = @"SELECT * FROM Person
20.                            WHERE Id = @Id";
21.            _dbConnection.Open();
22.            var persons = _dbConnection.Query<Person>(query, new { Id = id });
23.            _dbConnection.Close();
24.            return persons.FirstOrDefault();
25.        }
26.        public void InsertPerson(Person person)
```

```
27.         {
28.             string query = @"INSERT INTO Person
29.                             (
30.                                 Name,
31.                                 Phone
32.                             )
33.                             VALUES
34.                              (
35.                                 @Name,
36.                                 @Phone
37.                              )";
38.             _dbConnection.Open();
39.             var id = _dbConnection.Query<int>(query, new
40.             {
41.                 Name = person.Name,
42.                 Phone = person.Phone
43.             });
44.             _dbConnection.Close();
45.         }
46.     }
```

Este classe deve possui uma dependência da classe *Person*, que está em um projeto separado na solução. Para adicionar o namespace necessário, devemos adicionar no projeto *MinhaAgenda.Data*, uma referência ao projeto *MinhaAgenda.Services*. A versão 2015 do *Visual Studio* consegue adicionar simultaneamente essa referência e o *namespace* através do mesmo atalho mencionado acima, "*ctrl* + .". Entretanto, é importante que se conheça a forma manual de adicionar uma referência em um projeto. Para isto, basta clicar com o botão direito em *References*, no projeto desejado, e clicar em *Add Reference*. Na aba *Projects*, basta selecionar o projeto desejado. Além disso, devemos ainda lembrar de adicionar o namespace do *Dapper*, para incluir a definição do método de extensão "*Query* ", na interface

IDbConnector. Isso pode ser feito facilmente através do atalho "*ctrl* + *.*" com o cursor sobre a chamada para este método.

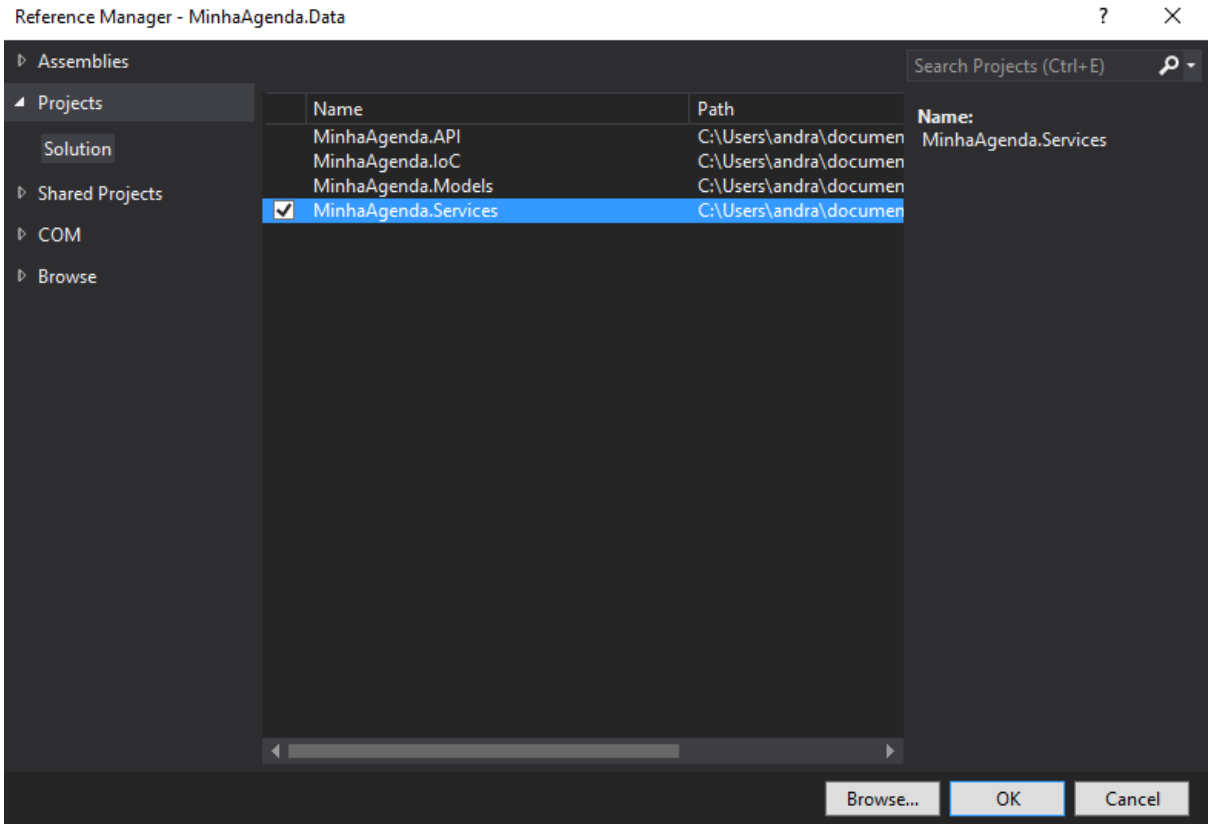

Figura 14 - Adicionando uma referência de projeto

Iniciando o desenvolvimento das funcionalidades da camada de apresentação, iremos implementar o projeto *MinhaAgenda.Models*. Este projeto possui duas funcionalidades básicas: contém os modelos que serão utilizados para receber e retornar os dados na API e contém as classes responsáveis pela validação destes modelos.

Para a validação, utilizaremos o *framework Fluent Validation* (FLUENT VALIDATION). Ele deve ser adicionado através do *NuGet*, de forma análoga ao *Dapper*.

Para começar as implementações, vamos criar a pasta *Person*, que conterá os modelos relacionados a esta entidade. Nesta pasta, são adicionadas duas classes: *CreatePersonModel* e *GetPersonModel:*

```
1.  public class CreatePersonModel
2.  {
3.      public string Name { get; set; }
4.      public string Phone { get; set; }
5.  }
```

```
1.  public class GetPersonModel
2.  {
3.      public int Id { get; set; }
4.      public string Name { get; set; }
5.      public string Phone { get; set; }
6.  }
```

Em seguida, criaremos os modelos responsáveis pelas respostas de erros da API. Para isso, deve ser adicionada outra pasta à raiz do projeto, chamada "*Report*", onde duas classes devem ser adicionadas: *Error* e *ErrorDetails.*

```
1.  public class Error
2.  {
3.      public string Code { get; set; }
4.      public string Message { get; set; }
5.      public string Description { get; set; }
6.      public IList<ErrorDetails> Details { get; set; }
7.  }
```

```
8.  public class ErrorDetails
9.  {
10.     public string Code { get; set; }
11.     public string Message { get; set; }
12.     public string Field { get; set; }
13. }
```

Além disso, devemos também criar uma pasta *Validation*, contendo uma classe *CreatePersonValidator*. Esta classe deve herdar de *AbstractValidator<CreatePersonModel>*,

de forma a implementar as validações do *Fluent Validation* para o objeto de criação de uma nova pessoa. Sua implementação é bem simples, contendo as validações desejadas no construtor da classe:

```
1.  public class CreatePersonValidator: AbstractValidator<CreatePersonModel>
2.      {
3.          public CreatePersonValidator()
4.          {
5.              RuleFor(person => person.Name).NotNull().WithErrorCode("VAL01")
6.                                            .NotEmpty().WithErrorCode("VAL01");
7.              RuleFor(person => person.Phone).NotNull().WithErrorCode("VAL02")
8.                                             .Length(10, 11).WithErrorCode("VAL02");
9.          }
10.     }
```

As chamadas ao método "*WithErrorCode*" servem para adicionar um código de erro customizado, como foi recomendado no capítulo 6.

Em seguida, devemos desenvolver as funcionalidades do projeto *MinhaAgenda.API*. Primeiramente, devemos incluir os pacotes *Fluent Validation* e *Simple Injector* através do gerenciador de pacotes *NuGet*. O *Simple Injector* será responsável pela implementação da injeção de dependências.

As classes existentes na pasta "*Controllers*" podem ser excluídas, pois não iremos utilizá-las. Podem ser excluídas também as pastas *Views* e *Fonts*, inteiramente.

Feito isso, devemos implementar os objetos que serão responsáveis pelo mapeamento entre os contratos da API e as entidades do sistema. Para isso, definiremos uma pasta *Mappers* na raiz do projeto, e incluiremos uma classe estática chamada *PersonMappers*:

```
1.  public static class PersonMapper
2.  {
3.      public static Person MapCreatePersonModelToEntity(CreatePersonModel model)
4.      {
5.          if (model == null) return null;
6.          var person = new Person()
7.          {
8.              Name = model.Name,
9.              Phone = model.Phone
10.         };
11.         return person;
12.     }
13.
14.     public static GetPersonModel MapEntityToGetPersonModel(Person person)
15.     {
16.         if (person == null) return null;
17.         var model = new GetPersonModel()
18.         {
19.             Name = person.Name,
20.             Phone = person.Phone
21.         };
22.         return model;
23.     }
24. }
```

Aqui, devemos adicionar referências a dois projetos da solução: *MinhaAgenda.Models* e *MinhaAgenda.Services*.

Ainda na pasta *Mappers*, deve ser adicionada uma classe estática *ErrorMapper*. Ele será responsável por mapear quaisquer erros de validação para o modelo da API.

```
1.  public static class ErrorMapper
2.  {
3.      public static Error MapValidationResultsToError(ValidationResult result)
4.      {
5.          var error = new Error()
6.          {
7.              Message = "Validation Errors",
8.              Code = "100",
9.              Details = new List<ErrorDetails>()
10.         };
11.         foreach(var validationError in result.Errors)
12.         {
13.             var errorDetails = new ErrorDetails()
14.             {
15.                 Message = validationError.ErrorMessage,
16.                 Code = validationError.ErrorCode,
17.                 Field = validationError.PropertyName
18.             };
                error.Details.Add(errorDetails);
19.         }
20.         return error;
21.     }
22. }
```

Após definidas as classes de mapeamento, devemos criar um controlador, que é responsável por receber as chamadas dos usuários para a API.

Para adicionar um novo controlador, clique com o botão direito na pasta *Controllers*, em seguida clicar em *Add -> New Controller*. O nome do controlador deve ser *PersonsController*, e sua implementação deverá ser da seguinte forma:

```
1.  [RoutePrefix("Persons")]
2.  public class PersonsController : ApiController
3.  {
4.      private readonly IPersonService _personService;
5.      private readonly IValidator<CreatePersonModel> _personValidator;
6.      public PersonsController(IPersonService personService,
7.                               IValidator<CreatePersonModel> personValidator)
8.      {
9.          _personService = personService;
10.         _personValidator = personValidator;
11.     }
12.     [HttpGet]
13.     [Route("")]
14.     public IHttpActionResult GetAllPersons()
15.     {
16.         try
17.         {
18.             var models = new List<GetPersonModel>();
19.             var persons = _personService.GetAllPersons();
20.             foreach (var person in persons)
21.             {
22.                 var model = PersonMapper.MapEntityToGetPersonModel(person);
23.                 models.Add(model);
24.             }
25.             return Ok(models);
26.         }
27.         catch (Exception ex)
28.         {
29.             return InternalServerError();
30.         }
31.     }
32.     [HttpGet]
33.     [Route("{id}", Name = "GetPersonById")]
```

```
34.         public IHttpActionResult GetPersonById(int id)
35.         {
36.             try
37.             {
38.                 var person = _personService.GetPersonById(id);
39.                 var model = PersonMapper.MapEntityToGetPersonModel(person);
40.                 return Ok(model);
41.             }
42.             catch (Exception ex)
43.             {
44.                 return InternalServerError();
45.             }
46.         }
47.         [HttpPost]
48.         [Route("")]
49.         public IHttpActionResult CreatePerson(CreatePersonModel model)
50.         {
51.             try
52.             {
53.                 var validationResults = _personValidator.Validate(model);
54.                 if (validationResults.Errors.Count > 0)
55.                 {
56.                     var errors =
    ErrorMapper.MapValidationResultsToError(validationResults);
57.                     return Content((HttpStatusCode)422, errors);
58.                 }
59.                 var person = PersonMapper.MapCreatePersonModelToEntity(model);
60.                 var insertedPerson = _personService.CreatePerson(person);
61.                 var insertedModel =
    PersonMapper.MapEntityToGetPersonModel(insertedPerson);
62.                 return CreatedAtRoute("GetPersonById", new { id = 1 },
    insertedModel);
63.             }
```

```
64.             catch (Exception ex)
65.             {
66.                 return InternalServerError();
67.             }
68.         }
69.     }
```

O atributo *RoutePrefix* na definição da classe e os atributos *Route* nos métodos são utilizados para se definir como será a url para as chamadas aos métodos deste controlador. A url para a chamada de cada método é feita da seguinte forma:

urlbase/RoutePrefix/Route/Parameters

Como estamos executando o sistema localmente, a URL base será definida como http://localhost:numerodaporta, onde o número da porta é definido automaticamente pelo visual studio. Com isso, temos por exemplo as seguintes URLs para o controlador *PersonsController*:

http://localhost:8086/Persons

http://localhost:8086/Persons/123

Os atributos *HttpGet* e *HttpPost* indicam o verbo HTTP que deve ser utilizado para se efetuar uma chamada àquele método.

Note que dependendo da sua versão do *.Net*, ao se gerar um controlador, o *template* padrão gerado faz com que o controlador herde da classe *Controller*. Devemos alterar para que a herança seja feita através da classe *ApiController*. Devemos ainda substituir o *namespace* S*ystem.Web.Mvc* pelo *namespace System.Web.Http*, onde a classe *ApiController* está definida.

As implementações do serviço e do validador serão recebidas automaticamente do construtor, através da injeção de dependências, pelo *framework Simple Injector*.

Seguindo para a injeção de dependências, devemos adicionar os pacotes *SimpleInjector*, *SimpleInjector.Integration.WebApi*, *Fluent Validation* e *Microsoft.AspNet.WebApi.Core* no projeto *MinhaAgenda.IoC*. Este último pacote contém a definição de uma classe de configuração da API que será utilizada na injeção de dependências.

Para a implementação da configuração, devemos adicionar uma classe estática chamada *IoCRegister* na raiz do projeto:

```
1.     public static class IoCRegister
2.     {
3.         public static Container Initialize(HttpConfiguration apiConfiguration)
4.         {
5.             var container = new Container();
6.             InitializeContainer(container);
7.             container.RegisterWebApiControllers(apiConfiguration);
8.             container.Verify();
9.             return container;
10.        }
11.        private static void InitializeContainer(Container container)
12.        {
13.            container.Register<IValidator<CreatePersonModel>,
    CreatePersonValidator>();
14.            container.Register<IPersonRepository, PersonRepository>();
15.            container.Register<IPersonService, PersonService>();
16.        }
17.    }
```

Um *container* de injeção de dependências é responsável por configurar e instanciar as interfaces que foram registradas nele. Para registrar as interfaces necessárias ao sistema, definimos o método estático *InitializeContainer*. No caso, temos três interfaces - *IValidator<CreatePersonModel>*, *IPersonRepository* e *IPersonService*, que serão

implementadas respectivamente pelas classes *CreatePersonValidator, PersonRepository* e *PersonService*.

No método *Initialize*, instanciamos e inicializamos o *container*. Em seguida, informamos a ele qual é o objeto de configuração da API (recebida como argumento deste método) e chamamos o método *Verify*, que valida se o container foi inicializado corretamente. Quaisquer problemas nesta etapa da configuração irão fazer este método disparar uma exceção, impedindo o sistema de seguir o processamento.

Com a utilização do *framework Simple Injector*, assim como com a utilização de diversos outros *frameworks* de injeção de dependências, estas configurações são o único momento da implementação em que precisamos nos preocupar com a injeção de dependências. A partir daí, ao definir o construtor de uma classe de forma que receba argumentos de interfaces registradas no *container*, as implementações serão recebidas neste construtor automaticamente.

Neste momento, finalizamos a implementação do *back-end* de nossa API. Entretanto, ainda não conseguimos enviar as requisições, uma vez que o banco de dados ainda não foi criado.

8.2 - Criação do banco de dados

8.2.1 - Instalação e criação do servidor

Para o sistema gerenciador de banco de dados, utilizaremos o *SQL Server 2016 Express Edition*. Para criar um banco local na máquina onde o sistema está sendo desenvolvido, devemos instalá-lo, assim como a ferramenta *Sql Server Management Studio*, que nos auxiliará no acesso e edição do banco de dados.

Após instalados, devemos abrir o *Sql Server Management Studio*, e em "*Nome do Servidor*", clicar em "*procurar mais*". Ali será possível ver os servidores locais, que foram criados na instalação do *Sql Server 2016.*

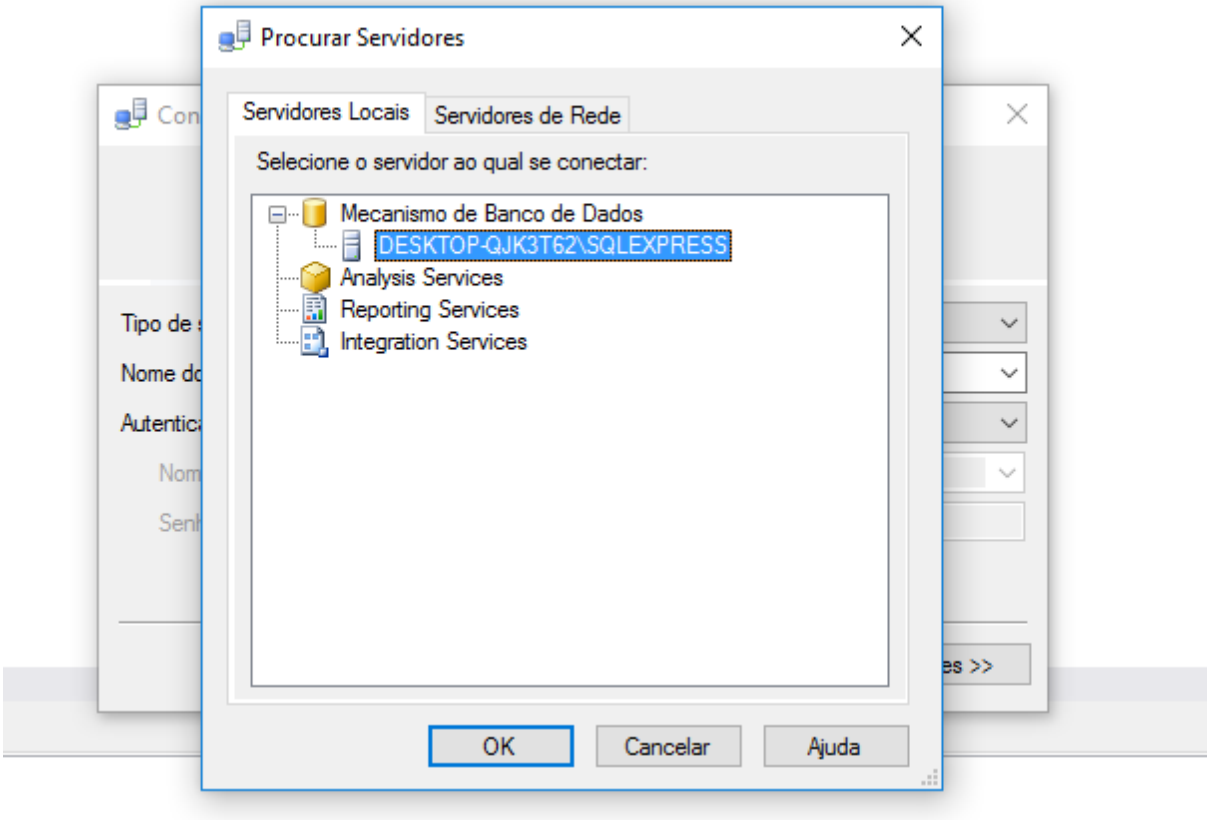

Figura 15 - Escolhendo servidor no SQL Server Management Studio

Selecione o seu servidor e clique *OK*, seguido de *conectar*.

No menu à esquerda, clique com o botão direito na pasta "*Banco de Dados*" e selecione "*Novo Banco de Dados*". Ali, basta escolher um nome para o banco e clicar em *Ok* para criá-lo. Se o banco ainda não estiver sendo mostrado na pasta, pressione *F5* com a pasta "*Bancos de Dados*" selecionada, para que o sistema se atualize.

8.2.2 - Criação da tabela

Para efetuar uma *query* no banco criado, clique com o botão direito nele e em seguida em *"Nova consulta"*. Para efetuar a criação da tabela *Person*, representada pela entidade de mesmo nome no código, devemos executar o seguinte script:

```
1. CREATE TABLE Person
```

```
2. (
3.     Id INT PRIMARY KEY IDENTITY NOT NULL,
4.     Name VARCHAR(56) NOT NULL,
5.     Phone VARCHAR(16) NOT NULL
6. )
```

A definição *Identity* no campo *Id* serve para que o valor para este campo seja gerado automaticamente na inserção de novas linhas na tabela. O *Sql Server* irá gerenciar este número de forma incremental. Dessa forma, para inserir uma nova entrada no banco, podemos apenas informar os campos que nos interessam - *Name* e *Phone*.

8.2.3 - Obtendo a string de conexão com o banco

A string de conexão (ou *connection string*), é responsável por definir a conexão do sistema com o banco de dados. Devemos atualizar o campo no construtor da classe *PersonRepository*, onde definimos o seu valor para informar à implementação da interface *IDbConnection*.

A maneira mais fácil de se obter a string de conexão com o banco é conectando o visual studio a ele. Para isso, clique em *Tools*, no menu superior da *IDE*, em seguida em *Connect to database*. Em *Server name*, utilize o mesmo valor usado para conectar ao banco no *Sql Server Management Studio* (muitas vezes a lista do *Visual Studio* não irá mostrar o nome do servidor em sua lista, porém você pode copiar e colar o valor mostrado pelo *Management Studio* - no caso do exemplo na imagem acima, *DESKTOP-QJK3T62\SQLEXPRESS)*.

Ao copiar o nome do servidor para o campo apropriado, o campo para selecionar o banco de dados será desbloqueado, e você poderá ver nesta lista o banco criado na etapa 8.2.1. Selecione o banco, clique em *Test Connection* para garantir a conexão pôde ser feita com sucesso, e clique em *OK*.

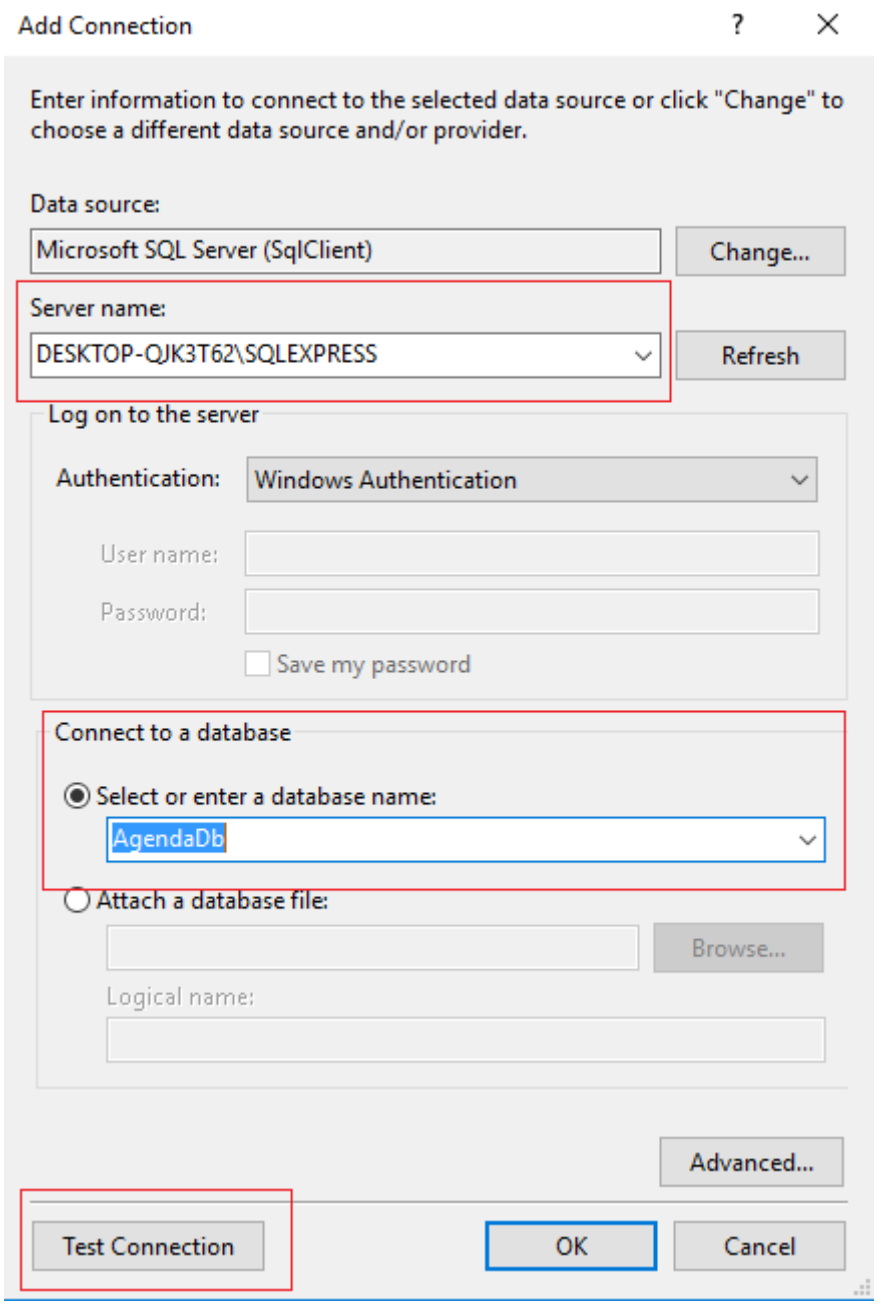

Figura 16 - Conexão do Visual Studio com o banco de dados

Após confirmar a conexão do *Visual Studio* com o banco, será mostrado o menu *Server Explorer* na *IDE* (caso não tenha sido mostrado, clique em *View -> Server Explorer*), contendo uma conexão com o banco de dados selecionado. Ali, clique com o botão direito em seu banco e em *Properties*. No menu mostrado, poderá ser obtida a string de conexão com o banco.

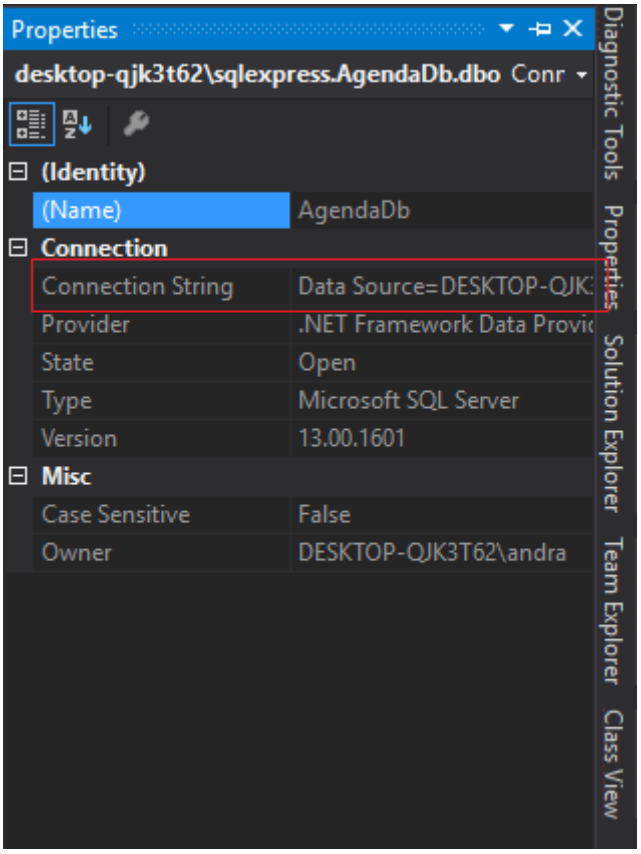

Figura 17 - Obtendo a string de conexão

O valor mostrado deverá ser substituído no construtor da classe PersonRepository, da seguinte forma:

```
1.  public PersonRepository()
2.  {
3.      string connectionString = @"INSERIR STRING DE CONEXÃO AQUI";
4.      _dbConnection = new SqlConnection(connectionString);
5.  }
```

8.3 - Testando a API

Para testar a API, utilizaremos o *software Postman*, que é capaz de enviar requisições *HTTP* para uma url desejada.

No *Visual Studio*, para executar a API, clique com o botão direito no projeto *MinhaAgenda.API* e selecione *"Mark as startup project"*. Em seguida, pressione *ctrl+F5* para iniciá-la. O seu navegador padrão será aberto, mostrando uma url *localhost*, da seguinte forma:

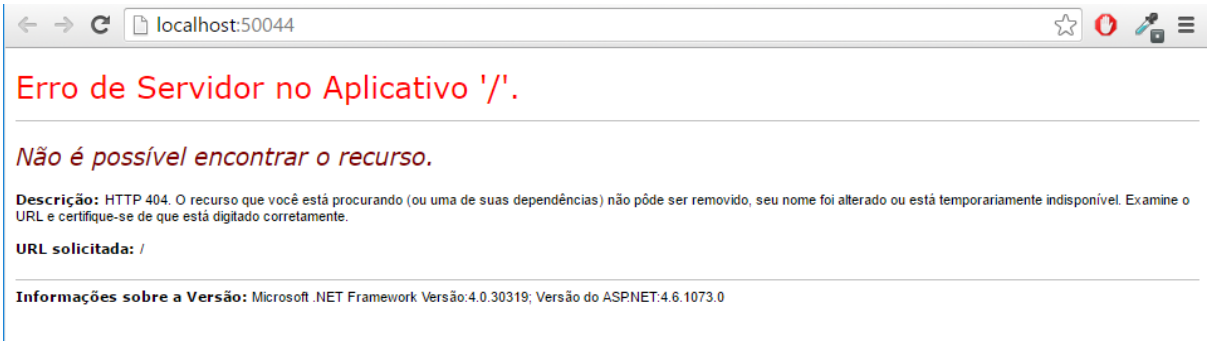

Figura 18 - Navegador em Localhost indicando a execução do sistema

Utilize esta url e porta para enviar requisições locais para a sua api.

Criação de uma pessoa:

Na interface do Postman, execute as seguintes instruções:

1. Alterar o verbo HTTP para POST
2. Inserir a url da api
3. Sob a aba Headers, inserir os *headers Content-Type* e *Accept* - Ambos com o valor *application/json*.
4. Sob a aba *Body*, selecionar a opção *raw* e inserir o seguinte *JSON*, representando o modelo de criação de uma nova pessoa:

```
{
    "Name": "Vitor de Andrade",
    "Phone": "21999999999"
}
```

Envie a requisição clicando em *Send*, e verifique se houve sucesso e se o código de status HTTP retornado foi 201 - Created.

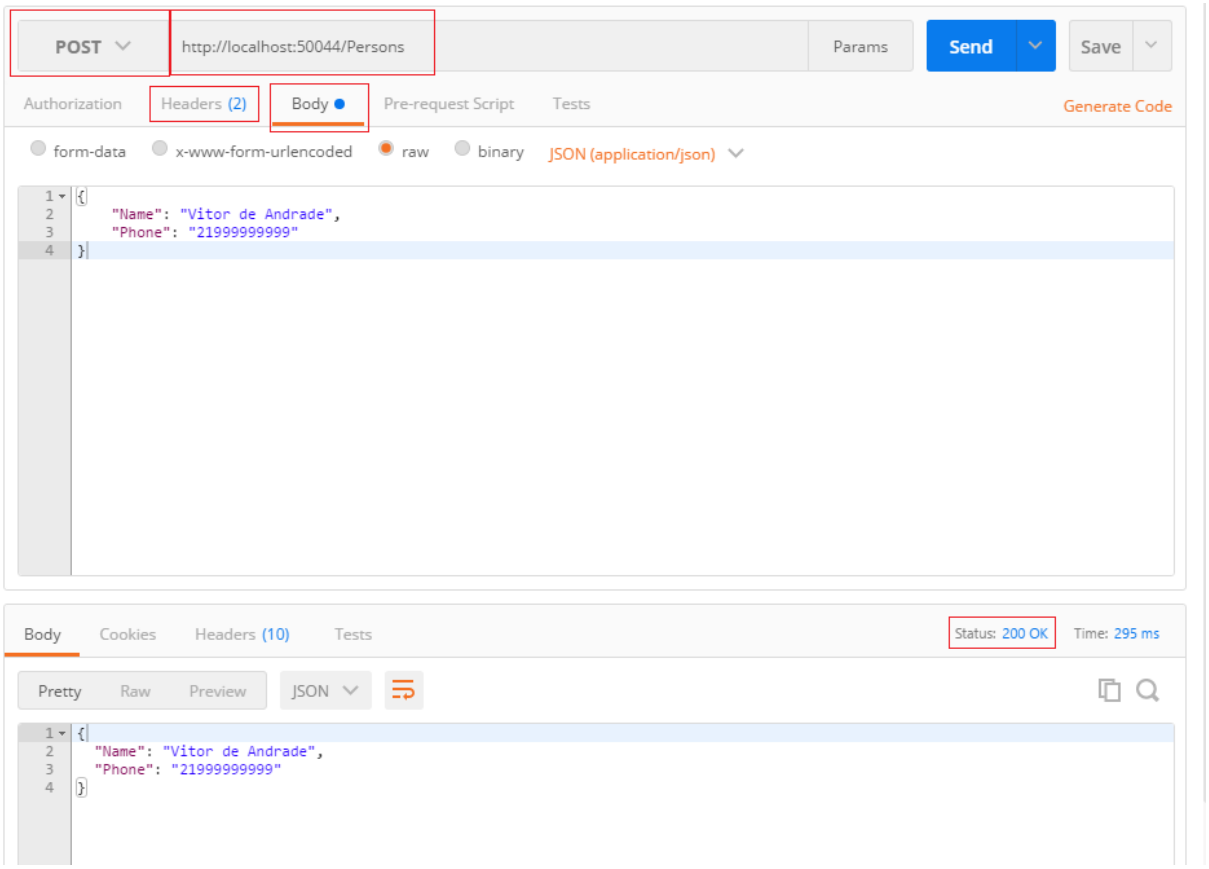

Figura 19 - Interface do Postman para criação de uma Pessoa no sistema

Listando todas as pessoas do banco

Para listar todas as pessoas no banco, basta gerar uma nova requisição no *Postman*, com o verbo HTTP GET, e enviá-la. Nas requisições GET, não precisamos definir nenhum corpo da requisição. Se desejado, pode ser definido o header *Accept*, para garantir que a API responderá à requisição utilizando o formato *JSON*.

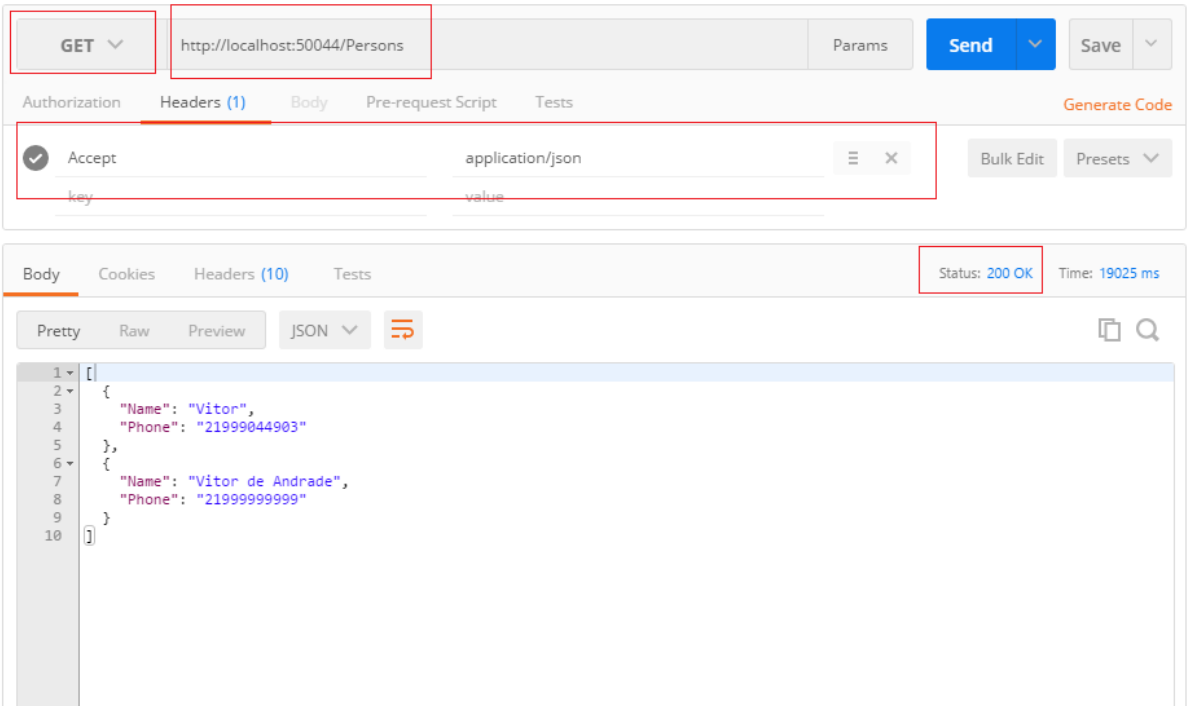

Figura 20 - Interface do postman para listagem de todas as pessoas no sistema

Capítulo 9

Conclusões

Neste projeto, introduzimos diversos conceitos importantes para o desenvolvimento do *back-end* de um sistema Web. Atualmente, estes sistemas são extremamente importantes, pois possuem um imenso alcance devido à grande popularização dos *smartphones* e trazem um grande poder de comunicação e tecnologia.

Inicialmente foram apresentados os conceitos da programação orientada a objetos. Esta é a metodologia de desenvolvimento que obteve o maior sucesso até hoje, e a grande maioria dos sistemas é baseada nela. Foram apresentadas também algumas sugestões de boas práticas de desenvolvimento através dos princípios S.O.L.I.D.

Em seguida, foram apresentadas algumas metodologias de projeto - DDD, TDD e BDD. Elas consistem em orientações que fornecem um guia para o desenvolvimento de um novo sistema. Apresentam assim, conceitos que otimizam o desenvolvimento, minimizam erros de sustentação no sistema e aumentam a clareza da comunicação entre desenvolvedores e equipes de negócios.

O quarto capítulo enunciou alguns dos padrões de projetos mais utilizados no desenvolvimento de um sistema web, que servem para solucionar problemas recorrentes nesta área, como por exemplo, o acesso ao banco de dados ou a redução do acoplamento em um sistema.

Em seguida, de forma a preparar o leitor para uma parte de grande importância nos sistemas web, foi apresentada uma introdução às redes de computadores, explicando alguns conceitos relacionados ao protocolo HTTP. Este protocolo é a base para o modelo de arquitetura REST, que foi apresentado no sexto capítulo.

O sexto capítulo introduziu o conceito de APIs, que são as interfaces de comunicação de um sistema com seus usuários. Através das APIs, podemos disponibilizar as funcionalidades de um sistema para diversos consumidores, como os navegadores em *laptops* e *desktops*, aplicativos para *smartphones,* etc. Além disso, foram apresentadas boas práticas no desenvolvimento de uma API utilizando a arquitetura REST.

Em seguida, apresentamos os conceitos básicos de banco de dados necessários para o desenvolvimento de um sistema. Foi explicado o que é um banco de dados relacional, e como ele é utilizado para se armazenar os dados de um sistema Web.

O capítulo final, um tutorial de desenvolvimento, sintetizou alguns dos conceitos apresentados anteriormente, de forma a permitir ao leitor a criação de um primeiro sistema web. A partir daí, os leitores interessados podem consultar as diversas referências apresentadas neste texto para se aprofundar e melhorar a arquitetura e o desenvolvimento, de forma a seguir com o crescimento de seus sistemas.

Além de todos os conceitos mencionados, ainda existem diversos outros tópicos extremamente interessantes que podem ser estudados de forma a garantir sistemas robustos, maturidade no trabalho em equipe e sistemas estáveis. Dentre estes tópicos, estão o controle de versão, integração contínua, conceitos de infraestrutura e servidores, *logs*, instrumentação, testes automatizados, metodologias de desenvolvimento ágil, desenvolvimento *mobile*, dentre outros.

Referências

ABNT, 2003, ISO IEC 9126: Engenharia de Software - Qualidade de produto. Parte 1: Modelo de Qualidade.

AHMED, K., 2016, **Journey to HTTP/2.** Disponível em: <http://kamranahmed.info/blog/2016/08/13/http-in-depth/>. Acesso em: 01 set. 2016

APIGEE, **HATEOAS 101: Opinionated Introduction to a REST API Style - Webcast**. Disponível em: <https://www.youtube.com/watch?v=6UXc71O7htc>. Acesso em 01 set. 2016.

BAILEY, D., 2010, **S.O.L.I.D Software Development, One Step at a Time**, Code Magazine, Jan/Feb 2010

BECK, K., 2002, **Test-Driven Development By Example.** Addison-Wesley.

BECK, K., 2004, **Extreme Programming Explained: Embrace Change,** Addison-Wesley.

BONDI, A.B., 2000, Characteristics of Scalability and Their Impact on Performance.

BUCKLER, C., 2015, **SQL vs NoSQL: The Differences**. Disponível em: <https://www.sitepoint.com/sql-vs-nosql-differences/>. Acesso em: 01 set. 2016.

CA TECHNOLOGIES, **A Guide to REST and API Design**

CA TECHNOLOGIES, **API Strategy and Architecture: A Coordinated Approach**. Disponível em: <https://www.ca.com/content/dam/ca/us/files/ebook/api-strategy-and-architecture-a-coordinated-approach.pdf>. Acesso em: 01 set. 2016.

CODD, E.F., 1970, **A Relational Model of Data for Large Shared Data Banks.**

DAPPER. Disponível em <https://github.com/StackExchange/dapper-dot-net>. Acesso em: 23 set. 2016.

EVANS, E., 2003, **Domain-Driven Design: Tackling Complexity in the Heart of Software.** Addison Wesley.

FIELDING, R.T., 2000, **Architectural Styles and the Design of Network-based Software**

FLUENT VALIDATION. Disponível em <https://fluentvalidation.codeplex.com/>. Acesso em: 23 set. 2016.

Architectures. Ph.D.

FOWLER, M., 2002, **Patterns of Enterprise Application Architecture**. Addison Wesley.

FOWLER, M., 2004, **Inversion of Control Containers and the Dependency Injection Pattern**. Disponível em: <http://www.martinfowler.com/articles/injection.html>. Acesso em: 01 set. 2016.

FOWLER, M., 2013, **NoSQL distilled: a brief guide to the emerging world of polyglot**. Addison-Wesley.

FOWLER, M., **Unit of Work**. Disponível em: <http://martinfowler.com/eaaCatalog/unitOfWork.html>. Acesso em: 01 set. 2016.

GAMMA, E., HELM, R. JOHNSON, R., et al., 1994. **Design Patterns: Elements of Reusable Object-Oriented Software.**

GITHUB API. Disponível Em: <https://developer.github.com/v3/>. Acesso em: 01 set. 2016.

HALL, G.M., 2014, **Adaptive Code via C#.**

IETF, 1998, **RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax.**

IETF, 1999, **RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1.**

IETF, 2010, **RFC 5789: PATCH Method for HTTP.**

IETF, 2010, **RFC 5988: Web Linking**

IETF, 1996, **RFC 1945: HyperText Transfer Protocol -- HTTP/1.0**

INTERNETLIVESTATS, Disponível em: <http://www.internetlivestats.com/>. Acesso em: 23 set. 2016

JONGERIUS, M.H., 2014, **The Seven Design Smells of Rotting Software**. Disponível em: <http://mhjongerius.tumblr.com/post/61853273412/the-seven-design-smells-of-rotting-software>. Acesso em: 01 set. 2016.

JUNIOR, H., 2012, **Transport Layer Security (TLS) e Secure Sockets Layer (SSL).** Disponível em <http://www.helviojunior.com.br/it/security/transport-layer-security-tls-e-secure-sockets-layer-ssl/ >. Acesso em: 23/09/2016.

KUROSE, J.F., ROSS, K.W., 2012, **Computer Networking - A Top Down Approach**, 6 ed., Pearson.

LASKEY, K.B., LASKEY, K., 2009, **Service Oriented Architecture**. Disponível em: <http://andreashellander.se/wp-content/uploads/2016/01/SOA.pdf>. Acesso em: 01 set. 2016.

MARTIN, R.C., 2000, **Design Principles and Design Patterns**.

MARTIN, R.C., 2006, **Agile Principles, Patterns and Practices in C#**. Prentice Hall.

MICROSOFT, **Layered Application Guidelines**. Disponível em: <https://msdn.microsoft.com/en-us/library/ee658109.aspx>. Acesso em: 01 set. 2016.

MICROSOFT, **The Repository Pattern**. Disponível em: <https://msdn.microsoft.com/en-us/library/ff649690.aspx>. Acesso em: 01 set. 2016.

MULESOFT, **What is an API?**. Disponível em <https://www.youtube.com/watch?v=s7wmiS2mSXY>. Acesso em: 01 set. 2016

MULLOY, B., 2012, **Web API Design: Crafting Interfaces that Developers Love** [ebook]

NORTH, D., 2006, **Introducing BDD**. Better Software Magazine. Disponível em: <https://dannorth.net/introducing-bdd/>. Acesso em: 01 set. 2016

NUGET. Disponível em <https://www.nuget.org/>. Acesso em: 23 set. 2016.

OLORUNTOBA, S., 2015, **S.O.L.I.D: The First 5 Principles of Object Oriented Design**. Disponível em <https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design>. Acesso em: 01 set. 2016.

PARKER, Z., POE, S., VRBSKY, S.V., 2013, **Comparing NoSQL MongoDB to an SQL DB**.

PIRES, E., 2012, **DDD, TDD BDD, Afinal, o que são essas siglas?.** Disponível em: <http://eduardopires.net.br/2012/06/ddd-tdd-bdd/>. Acesso em: 01 set. 2016.

POSTMAN. Disponível em <https://www.getpostman.com/>. Acesso em: 23 set. 2016.

RAMAKRISHNAN, R., GEHRKE, J., 2003, **Database Management Systems.** McGraw-Hill

SAHNI, V., **Best Practices for Designing a Pragmatic RESTful API**. Disponível em: <http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api>. Acesso em: 01 set. 2016.

SILBERSCHATZ, A. KORTH, H., SUDARSHAN, S., 2010, **Database System Concepts,** 6 ed., McGraw-Hill.

SIMPLE INJECTOR. Disponível em <https://simpleinjector.org>. Acesso em: 23 set. 2016.

STRIPE API. Disponível em: <https://stripe.com/docs/api#versioning>. Acesso em: 01 set. 2016.