UNIVERSIDADE FEDERAL DO PARÁ

LUIZ EDUARDO ALVES DE ALCÂNTARA

UM *FRAMEWORK* PARA PLATAFORMAS DE MICROSERVIÇOS DISTRIBUÍDOS

Belém
2016

LUIZ EDUARDO ALVES DE ALCÂNTARA

UM *FRAMEWORK* PARA PLATAFORMAS DE MICROSERVIÇOS DISTRIBUÍDOS

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Sistemas de Informação, da Faculdade de Computação, na Universidade Federal do Pará, como parte dos requisitos necessários para a obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Josivaldo de Souza Araújo

Belém
2016

LUIZ EDUARDO ALVES DE ALCÂNTARA

UM *FRAMEWORK* PARA PLATAFORMAS DE MICROSERVIÇOS DISTRIBUÍDOS

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Sistemas de Informação, da Faculdade de Computação, na Universidade Federal do Pará, como parte dos requisitos necessários para a obtenção do grau de Bacharel em Sistemas de Informação.

Belém, 7 de Março de 2017

BANCA EXAMINADORA

Professor Doutor Josivaldo de Souza Araújo
Orientador - UFPA/ICEN/Faculdade de Computação

Professora Doutora Fabíola Pantoja Oliveira Araújo
UFPA/Campus Castanhal/Faculdade de Computação

Professor Mestre de Ciências Inácio Leite Gorayeb
Membro - FAMAZ

Aos meus avôs e avós, minha mãe e meu irmão, que descansem em paz; Ao meu filho, que tenha um futuro próspero e feliz.

AGRADECIMENTOS

À minha avó, que cuidou de mim quando minha família se dividiu.

Ao meu pai, que me ensinou tolerância e curiosidade.

À minha mãe, que me ensinou humildade e força, além da importância da educação. Queria muito me ver formado pela UFPA, assim como ela.

Ao meu irmão, que em vida me ensinou alegria e otimismo, e na morte me ensinou a fragilidade e a urgência de amar e cuidar.

Ao meu padrasto, que me proveu de recursos para descobrir minha vocação e me ensinou que não é só o sangue que identifica a família, e que a ética está não só nas grandes decisões, mas também nas pequenas escolhas de cada dia.

À minha sobrinha, que me ensinou a importância de proteger e educar a próxima geração, em busca de uma sociedade melhor.

Ao meu primo Fernando, que me ensinou resiliência e dedicação.

À minha tia Silvia, pela amizade e cuidado à minha mãe nos últimos momentos.

Ao meu filho, que deu sentido a minha vida, e a mudou para melhor, assim como mudou minha forma de ver o mundo, também para melhor.

À minha Ana Paula Giliberti, que cuida de mim e de meu filho com amor e zelo, e me inspira a ser uma pessoa melhor a cada dia.

Aos meus colegas de SERPRO e TRE-PA que me receberam tão bem e ajudam a me tornar um profissional melhor e acreditar no serviço público federal.

Ao meu orientador neste trabalho que acreditou em meu valor e teve paciência em face de minhas dificuldades para organizar meu cronograma.

A Universidade Federal do Pará e seus professores e funcionários, que apesar das dificuldades e batalhas intermináveis, sempre estão presentes, dia após dia, em nome do futuro do Brasil.

"Tudo deve ser feito tão simples quanto possível; mas não simplório." (Albert Einstein)

RESUMO

Os esforços em transformar uma linguagem de programação genérica como Java em uma ferramenta completa para desenvolver aplicações Web tem tornado muito difícil acompanhar sua crescente curva de aprendizado por conta de diversos *frameworks* acoplados que tem se tornado padrão de desenvolvimento. Um caminho difícil de ser mudado, por conta de sua popularidade em diversos segmentos, reforçada pelo marketing agressivo de gigantes como a Oracle, sua atual proprietária. Este trabalho propõe, em contrapartida, o uso de plataformas distribuídas em microserviços e funções independentes de APIs (inclusive para acesso a bancos de dados) cujas implementações deverão seguir as especificações de um *framework* projetado para permitir a baixa curva de aprendizagem, o rápido desenvolvimento, maior economia de recursos de TI e a total compatibilidade com todas as plataformas desenvolvidas sob suas diretrizes, independente de quais linguagens de programação serão usadas, tanto para implementação da plataforma quanto para os serviços disponibilizados, que podem pertencer a qualquer categoria padrão de mercado, como gestão de usuários e acesso autenticado, criptografia forte independente de protocolo, repositório de arquivos e gestão inteligente de armazenamento, comunicação distribuída síncrona e assíncrona através de mensageria, modelagem de saída dinâmica de interface ao usuário, *streaming* multimídia em tempo real e execução de *procedures* de banco de dados com retorno direto à camada do cliente.

Palavras-chave: *Framework* de Implementação. Plataforma de Sistemas Distribuídos. Microserviços. Funções *Serverless*

ABSTRACT

The efforts to transform a generic programming language like Java in a complete tool to develop Web applications has been turning too hard to keep following its growing learning curve because of its many coupled frameworks which have become standard development. It's a difficult path to change, due to its popularity in many segments, reinforced by the aggressive marketing of giants like Oracle, its current owner. This paperwork proposes, in counterpart, the use of distributed platforms into Microservices and API independent functions (inclusive for database access) in which the implementations should follow the specifications of a framework designed to have a low learning curve, fast development, higher economy of IT resources and full compatibility with all platforms developed under its guidelines, independently of each programming languages will be used, both for the implementation of the platform and for the services provided, which may belong to any standard market category, such as user management and authenticated access, cryptograph that is strong and protocol independent, file repository and intelligent storage management, synchronous and asynchronous distributed communication through messaging, dynamic user interface modeling, Real-time multimedia streaming and execution of database procedures with a direct return to client layer.

Keywords: Implementation Framework. Distributed Systems Platform. Microservices. Serverless Functions

LISTA DE ILUSTRAÇÕES

Figura 1 - Quebra de arquiteturas, de monolítica até serverless 26
Diagrama 1 - Tabelas do banco e as referências entre elas 43
Diagrama 2 - A Plataforma em uma arquitetura não-distribuída 75
Diagrama 3 - A Plataforma em uma arquitetura parcialmente distribuída 75
Diagrama 4 - A Plataforma em uma arquitetura totalmente distribuída 76
Figura 2 - Telas de login e operação do Sistema Acesso Servidor 78
Figura 3 - Telas de operação e configuração do sistema TRE-TV 79
Figura 4 - Exemplo de saída do comando /relógio do TRE-TV 80
Gráfico 1 - Tamanho do aplicativo e das respostas HTTP 82
Gráfico 2 - Tempo mínimo de carregamento da página 82
Gráfico 3 - Tempo de carregamento dos serviços do TRE-TV 83
Tabela 1 - Total de especificações do projeto de framework 84
Quadro - Padrões de modelagem e convenções no bancos de dados 94
Quadro - Padrões de nomenclatura da dados comuns no banco 95
Quadro - Linguagens de programação para a camada cliente 105

LISTA DE QUADROS

Quadro 1 - Camadas do Java EE e algumas de suas especificações 22
Quadro 2 - Frameworks Java por categoria de função 23
Quadro 3 - Parâmetros básicos para requisições da plataforma 33
Quadro 4 - Ordem de passagem de parâmetros via URL limpa 35
Quadro 5 - Convenções e Configurações do Framework 41
Quadro 6 - Banco de dados básico do Framework 44
Quadro 7 - Banco de dados para gestão e operação de serviços 44
Quadro 8 - Banco de dados para gestão de licenças 45
Quadro 9 - Banco de dados para serviços extras 46
Quadro 10 - Listagens tipo ENUM para o banco de dados 47
Quadro 11 - Estrutura principal do objeto $platformScope 52
Quadro 12 - Funções JavaScript para acesso à Plataforma 53
Quadro 13 - Funções JavaScript para criptografia de dados 53
Quadro 14 - Módulos do Gestor de Requisições 69

LISTA DE ABREVIATURAS E SIGLAS

AJAX	*Asynchronous Javascript and XML*
AMQP	*Advanced Message Queuing Protocol*
ANSI	*American National Standards Institute*
API	*Application Programming Interface*
APEX	*Oracle Application Express*
ASP	*Active Server Pages*
BD	Banco de Dados
BIN	*Binary File*
CDN	*Content Delivery Network*
CGI	*Computer Generated Imagery*
CoC	*Convention Over Configuration*
CORS	*Cross-Origin Resource Sharing*
CPU	*Central Process Unit*
CRM	*Customer Relationship Management*
CRUD	*Create, Read, Update, Delete*
CSS	*Cascading Style Sheets*
CSV	*Comma-Separated Values*
DAO	*Data Access Object*
DB	*Data Base*
DBA	*Data Base Administrator*
DDoS	*Distributed DoS Attack*
DHTML	*Dynamic HTML*
DI	*Dependency Injection*
DLL	*Dinamic Link Library*
DOM	*Document Object Model*
DoS	*Deny Of Service*

DOS	*Disk Operational System*
DDNS	*Dynamic Domain Name Server*
EJB	*Enterprise JavaBeans*
EPP	Empresa de Pequeno Porte
EE	*Enterprise Edition*
EXE	*Executable File*
FAQ	*Frequently Asked Questions*
FK	*Foreign Key*
FTP	*File Transfer Protocol*
GPS	*Global Positioning System*
GRP	Gerente de Requisições da Plataforma
HTML	*HyperText Markup Language*
HTTP	*HyperText Transfer Protocol*
HTTPS	*HTTP Secure / HTTP over SSL*
IBM	*International Business Machines*
IDE	*Integrated Development Environment*
IP	*Internet Protocol*
IoC	*Inversion of Control*
J2EE	*Java 2 Platform Enterprise Edition*
JAAS	*Java Authentication and Authorization Service*
JAX	*Java API for XML*
JDBC	*Java Database Connectivity*
JDK	*Java Development Kit*
JMS	*Java Message Service*
JPA	*Java Persistence API*
JPQL	*Java Persistence Query Language*
JRE	*Java Runtime Environment*

JS	JavaScript
JSF	*JavaServer Faces*
JSON	*JavaScript Object Notation*
JSONP	*JSON with Padding*
JSP	*JavaServer Pages*
JSTL	*JavaServer Pages Standard Tag Library*
KB	*Kilobyte*
LOG	Arquivo com histórico de atividades de um sistema
MIN	Minutos
MOM	*Middleware Oriented Message*
MPE	Micro e Pequena Empresa
MPS.BR	Melhoria do Processo de Software Brasileiro
MS	Microsoft
MIT	*Massachusetts Institute of Technology*
MVC	*Model, View, Controller*
MVVM	*Model–view–viewmodel*
NAT	*Network Address Translation*
OCCI	*Open Cloud Computing Interface*
ORM	*Object-Relational Mapping*
P2P	*Peer-To-Peer*
PBQP-Sw	Programa Brasileiro da Qualidade e Produtividade em Software
PC	*Personal Computer*
PDF	*Portable Document Format File*
PHP	*Personal Home Page/forms interpreter*
PK	*Primary Key*
QoS	*Quality of Service*
RAD	*Rapid Application Development*

RC4	*Rivest Cipher 4*
RDBMS	*Relational Database Management System*
RFC	*Request For Comments*
RIA	*Rich Internet Application*
RMI	*Remote Method Invocation*
RPC	*Remote Procedure Call*
RTCP	*RTP Control Protocol*
RTF	*Rich Text File*
RTP	*Real-time Transport Protocol*
RTSP	*Real Time Streaming Protocol*
SCV	Sistema de Controle de Versionamento
SGBD	Sistema Gerenciador de Bancos de Dados
SLA	*Service Level Agreement*
SO	Sistema Operacional
SOAP	*Simple Object Access Protocol*
SP	*Stored Procedure*
SQL	*Structured Query Language*
SRTP	*Secure Real-time Transport Protocol*
SSH	*Secure Shell*
SSL	*Secure Sockets Layer*
TCP	*Transmission Control Protocol*
TCU	Tribunal de Contas da União
TLS	*Transport Layer Security*
TRE-PA	Tribunal Regional Eleitoral do Pará
TV	Televisão
TXT	*Text File*
UDP	*User Datagram Protocol*

UML	*Unified Modeling Language*
UOL	Universo *Online*
URI	*Uniform Resource Identifier*
URL	Uniform Resource Locator
URN	*Uniform Resource Name*
VoIP	*Voice Over IP*
W3C	*World Wide Web Consortium*
WAR	*Web application ARchive*
WML	*Website Meta Language*
WS	*Web Service*
WSDL	*Web Services Description Language*
WWW	*World Wide Web*
XHTML	*Extensible HTML*
XLS	Arquivo no formato do Microsoft Excel
XML	*Extensible Markup Language*
XSLT	*Extensible Stylesheet Language Transformations*
ZIP	Arquivo compactado no formato ZIP (conotativo de Zipper)

SUMÁRIO

1 **INTRODUÇÃO** 12
1.1 OBJETIVO GERAL 14
1.2 OBJETIVOS ESPECÍFICOS 14
1.3 METODOLOGIA 15
1.4 ORGANIZAÇÃO 16
1.5 TRABALHOS RELACIONADOS 16
1.6 RESULTADOS ESPERADOS 17
2 **FUNDAMENTAÇÃO E CONCEITOS** 18
2.1 A NATUREZA DAS LINGUAGENS DE PROGRAMAÇÃO 18
2.1.1 **Interpretação e compilação de linguagens de programação** 18
2.2 A LINGUAGEM JAVA E SEUS *FRAMEWORKS* 19
2.2.1 **Plataforma Java EE** 19
2.2.1.1 Camadas da Plataforma Java EE 21
2.2.2 ***Frameworks* necessários para desenvolver aplicações na Web** 22
2.2.3 **Vantagens e desvantagens de Java EE** 23
2.3 A COMPUTAÇÃO DISTRIBUÍDA 24
2.3.1 **Balanceamento de carga** 24
2.3.2 **Microserviços** 25
2.4 CONSIDERAÇÕES FINAIS DESTE CAPÍTULO 26
3 **DESENVOLVIMENTO DO PROJETO** 27
3.1 A ARQUITETURA DA PLATAFORMA 27
3.1.1 **Funções da Plataforma** 27
3.1.2 **Modalidades de Serviços da Plataforma** 29
3.2 O LADO DO SERVIDOR 31
3.2.1 **Gerente de Requisições** 32
3.2.1.1 Parâmetros básicos para envio de requisições na Plataforma 33
3.2.1.2 Especificações para Formato de Requisições 34
3.2.1.3 Especificações do Gerente de Requisições para TCP e UDP 36
3.2.1.4 Fluxo de Requisições na Plataforma 36
3.2.1.5 Os conceitos de Sequencias e Buffer de Requisições 38
3.2.1.6 Catálogo de serviços disponíveis na Plataforma 39
3.2.2 **Banco de Dados** 40
3.2.2.1 Convenções e Configurações 40
3.2.2.2 Especificações do Banco de Dados 41
3.2.2.3 Tabelas de banco de dados preexistentes no *Framework* 42
3.2.2.3.1 *Tabelas de Usuários, Sessões e Procedures* 43

3.2.2.3.2 *Tabelas de Configuração e Gestão de Serviços* 44
3.2.2.3.3 *Tabelas de Controle de Licenças e Créditos de Usuários* 45
3.2.2.3.4 *Tabelas de Aprimoramento da Plataforma* 45
3.2.2.3.5 *Tabelas para Enumeradores (Listas)* 46
3.3 O LADO DO CLIENTE 48
3.3.1 **Especificações de Compatibilidade** 49
3.3.2 **Bibliotecas de Auxilio ao Desenvolvimento** 50
3.3.2.1 Javascript para clientes Web ou Mobile 51
3.3.2.2 Outras linguagens para clientes no Modo Desktop 54
3.4 SEGURANÇA 54
3.4.1 **Especificações de Segurança para Comunicação** 55
3.4.2 **Especificações de Segurança para Configuração** 56
3.5 SERVIÇOS, MICROSERVIÇOS E FUNÇÕES SERVERLESS 56
3.5.1 **Serviços de Sessões e Acesso de Usuários** 57
3.5.1.1 Especificações de Serviço 57
3.5.1.2 Ações de Serviço e seus Parâmetros 58
3.5.2 **Serviços de Armazenamento e Uso de Arquivos** 59
3.5.2.1 Especificações de Serviço 59
3.5.2.2 Ações de Serviço e seus Parâmetros 60
3.5.3 **Serviços de Bancos de Dados** 61
3.5.3.1 Especificações de Serviço 62
3.5.3.2 Ações de Serviço e seus Parâmetros 63
3.5.4 **Serviços Customizados Acoplados na Plataforma** 64
3.5.4.1 Especificações de Serviço 64
3.5.4.2 Ações de Serviço e seus Parâmetros 66
3.5.5 **Serviços de Gestão da Plataforma** 66
3.5.5.1 Especificações de Serviço 66
3.5.5.2 Ações de Serviço e Seus Parâmetros 67
3.5.6 **Serviços do Núcleo da Plataforma** 68
3.5.6.1 Especificações do Serviço 68
3.5.6.2 Ações de Serviço e Seus Parâmetros 70
3.5.7 **Serviços em Sequência** 70
3.5.7.1 Especificações do Serviço 71
3.5.7.2 Ações de Serviço e Seus Parâmetros 72
3.6 CONSIDERAÇÕES FINAIS DESTE CAPÍTULO 72
4 **APLICAÇÕES E RESULTADOS** 73
4.1 APLICAÇÕES TEÓRICAS 73
4.1.1 **Públicos Alvo** 73

4.1.2 **Escalabilidade** 74
4.2 APLICAÇÕES PRÁTICAS 76
4.2.1 **Testes de Laboratório** 77
4.2.2 **Implementação Parcial** 77
4.2.2.1 Sistema de Acesso do Servidor 78
4.2.2.2 Sistema de TV Institucional 79
4.3 RESULTADOS OBTIDOS 81
4.3.1 **Resultados das Implementações Parciais** 81
4.3.2 **Resultados do Desenvolvimento do Projeto do *Framework*** 84
4.4 CONSIDERAÇÕES FINAIS DESTE CAPÍTULO 84
5 **CONCLUSÃO** 86
REFERÊNCIAS 88
APÊNDICE A 92
APÊNDICE B 94
APÊNDICE C 96
APÊNDICE D 101
APÊNDICE E 103
APÊNDICE F 105
APÊNDICE G 106

1 **INTRODUÇÃO**

Desde a concepção dos primeiros computadores como o ENIAC (1946), do tamanho de uma casa, até a atualidade dos dispositivos móveis, o software esteve presente como a interface principal que nos permite programar as ações destas complexas máquinas. No início, eram utilizados rudimentares cartões perfurados para passar instruções ao hardware, mas depois foram criando e evoluindo as linguagens de programação para refletir à alta disponibilidade e poder de processamento que os computadores modernos alcançaram (KELLY, 2003).

A velocidade desta evolução, nos parâmetros da Lei de Moore, fez com que os paradigmas sofressem várias alterações durante as últimas décadas. Passaram de computadores exclusivos, cujo software era desenvolvido especificamente para tal unidade, para microcomputadores feitos em larga escala, cuja arquitetura padronizada permitiu o desenvolvimento de aplicativos que agora podiam ser executados em milhões de computadores com o mesmo código de máquina. As linguagens de programação seguiram o mesmo caminho, padronizando-se para atender a demanda mundial por novas soluções para proprietários residenciais e corporativos (BORKAR; CHIEN, 2011, com adaptações).

Entretanto, com os impulsos da globalização e da Internet, os sistemas de informação sofreram outra mudança de paradigma. Se antes eles eram desenvolvidos em sua totalidade para executar localmente em arquiteturas de hardware específicas, e utilizadas por apenas um usuário por máquina, agora eles residem, na sua maioria, em servidores remotos, para serem executados por um número indefinido de usuários, de forma paralela, utilizando as mais variadas plataformas de hardware e software para acessar esses aplicativos (BYRNE, 2013).

Desta vez, as linguagens de programação e suas camadas de apoio não tiveram total êxito em acompanhar esta evolução, e o que se vê é o uso exagerado de bibliotecas, frameworks de linguagens de programação e muitas camadas intermediárias de software, só para permitir a criação de aplicativos modernos usando a mesma infraestrutura e padrões estabelecidos nas décadas de 80 e 90, como HTML e HTTP, que foram concebidos para transmitir dados estáticos, mas que hoje são a base para aplicações dinâmicas remotas (BORKAR; CHIEN, 2011).

Neste ambiente, vem se destacando cada vez mais a linguagem Java, por ter código aberto, ser constantemente evoluída por uma comunidade internacional de empresas e desenvolvedores autônomos, e principalmente por aceitar múltiplas plataformas de hardware e software sem mudança no código fonte das aplicações, já que os aplicativos são executados por um interpretador ao invés de serem compilados em código de máquinas específicas (DEITEL, 2010, com adaptações).

Porém, até mesmo o Java, foi baseado em uma linguagem bem mais antiga: o C, e sua biblioteca base possui apenas um suporte primitivo a requisições remotas, e nenhum outro suporte nativo ao que é necessário para desenvolver aplicações Web. Por isso, somente oito anos depois da criação da linguagem, foi desenvolvida a plataforma chamada de J2EE, que continha um *framework* de bibliotecas para Web e um servidor de aplicações corporativas escaláveis. Dezesseis anos depois, em sua oitava versão, a plataforma Java ainda depende de vários outros *frameworks* oficiais e outros desenvolvidos pela comunidade, para resolver problemas de segurança, rede, consistência de sessões de usuário, camadas de aplicativo e de negócio, abstração de dados e desenvolvimento de interface com o usuário.

Atualmente, a lei de Moore está emperrada em fatores físicos, como o calor, que limitam o aumento da velocidade dos processadores. Apesar disso, foram atingidas velocidades de processamento tão altas, que aplicações residenciais e corporativas de médio porte não encontram dificuldades para funcionar nem mesmo em pequenos smartphones. Segundo Borkar e Chien (2011), esta zona de conforto tem mudado o foco dos projetistas de software, da antiga preocupação com o desempenho e consumo de memória, para o alto investimento em interface com o usuário – efeitos visuais e sonoros – que individualmente parecem inofensivos, mas que podem engessar o funcionamento de hardware e software em termos de escalabilidade.

O grande número de *frameworks* utilizados um sobre o outro também é um fator de perda de desempenho, pois inclui, em cada novo projeto, milhões de linhas de código em recursos que nem sempre são utilizados em sua totalidade. Este desperdício pode ser, em alguns casos, humanamente impossível de detectar, necessitando de ferramentas que atestem o consumo de hardware necessário para executar a aplicação, assim como aumento no consumo de energia e consequentemente o aquecimento de todo o sistema.

Neste processo, o que é plenamente notável ao entrar na comunidade Java, é o aumento da curva de aprendizagem, que de acordo com De ROSA (2013) , é a constatação do aumento da dificuldade de aprender ou dominar plenamente todos os processos e normas de desenvolvimento, todas as ferramentas e recursos disponibilizados à equipe de desenvolvimento e ao indivíduo, considerando o grande número de tecnologias e *frameworks* inseridos na demanda, cada um com suas regras, peculiaridades e configurações individuais e os resultados do entrelaçamento entre todos eles. É possível comparar isso à diferença entre operar um carro automático e operar um avião intercontinental moderno.

Pode-se imaginar que quanto mais complexos e poderosos os sistemas computacionais se tornaram, mais difícil seria o desenvolvimento de aplicações. Mas se fosse verdadeira esta premissa, também poderia ser mais complexa a sua operação, mas o que acontece é o contrário. Também percebe-se que o desenvolvimento de código aberto tem oferecido trégua aos desenvolvedores do mundo todo ao compartilharem suas soluções, evitando o retrabalho no desenvolvimento das funções mais genéricas às mais específicas (NOYES, 2010).

É possível dar o próximo passo na evolução do software, desprendendo as aplicações da Web e criando novas tecnologias mais condizentes com a arquitetura de sistemas, ao invés de usar uma antiga tecnologia para texto com hiper-ligações. Porém o mundo de hoje está tão imerso na Web, que uma mudança drástica nos paradigmas poderia custar fortunas, dividir opiniões, e demoraria muito tempo para se fazer uma migração completa.

Enquanto isso, é possível pensar em soluções melhores para desenvolver aplicativos Web mais ecológicos – que não consumam mais recursos de hardware do que deveriam, e cuja curva de aprendizado para seu projeto e desenvolvimento seja mais baixa – acessível a programadores novatos e experientes.

1.1 OBJETIVO GERAL

É no contexto destas problemáticas, que surge a ideia contida neste trabalho, cujo objetivo geral é projetar uma proposta de arquitetura de plataforma de serviços, modelada em forma de *framework* padronizado, para que possa ser implementada em qualquer linguagem de programação e banco de dados desejado, mantendo-se as características principais, que são:

- Programar e executar aplicativos como Microserviços independentes entre si;
- Permitir o projeto e criação de software focado no modelo de negócio;
- Oferecer camada dupla de segurança independente do protocolo utilizado;
- Permitir gerenciamento padronizado de usuários, pela plataforma;
- Permitir a redundância de servidores e espelhamento de configurações e bases de dados, bem como distribuir seu conteúdo em qualquer modelo;
- Permitir uma baixa curva de aprendizado a todos os aspectos da plataforma;
- Permitir o baixo consumo de recursos energéticos e de hardware;
- Permitir a inversão de controle dos requisitos não funcionais, da aplicação para a plataforma.

1.2 OBJETIVOS ESPECÍFICOS

Para este projeto, será necessário modelar várias tecnologias, todas convergindo para o propósito principal, mas que pareçam uma só entidade, transparente para os desenvolvedores, operadores e usuário final. Portanto, para os objetivos específicos deste trabalho, tem-se:

- Projetar um serviço para gerenciar sessões de usuário, transportar dados e mediar serviços da plataforma;
- Projetar um modelo de comunicação criptografada independente de protocolo;
- Criar três modelos de bancos de dados, um para a configuração da plataforma, um para gerência de usuários e outro para a o modelos de negócio e serviços disponíveis;
- Projetar um modelo administrativo flexível que permita tanto opções de customização, como o métodos de Convenção sobre Configuração, que permitam uma boa curva de aprendizado para todos os níveis profissionais;
- Apresentar cenários de aplicação destas propostas;
- Disponibilizar o projeto em repositórios públicos, para ser compartilhado e melhorado pela comunidade internacional;

1.3 METODOLOGIA

Este projeto é resultado de longas reflexões teóricas, observações técnicas, desenvolvimento prático, coleta de requisitos, conversas em fóruns profissionais e tem como base mais de quinze anos de trabalho do autor em desenvolvimento de sistemas para computadores, web e dispositivos móveis, incluindo pesquisas bibliográficas e cursos nas áreas de gerência de projetos e desenvolvimento Web.

Serão detalhadas as especificações para o desenvolvimento deste *Framework*, bem como o comportamento padrão dos objetos criados, que poderão ser desenvolvidos em qualquer linguagem de programação compatível com a arquitetura cliente/servidor e qualquer Sistema Gerenciador de Banco de Dados (SGBD) que permita a criação de *stored procedures*, e que seja regrada a fim de que qualquer versão desenvolvida da Plataforma tenha capacidade de se comunicar e compartilhar recursos e serviços entre si. Serão sugeridas tecnologias *open-source* gratuitas para facilitar sua replicação pela comunidade de desenvolvimento.

1.4 ORGANIZAÇÃO

Este trabalho foi dividido em três capítulos, além da Introdução e Conclusão, e são organizados de acordo com as necessidades informacionais de um projeto de desenvolvimento de sistemas. São eles:

- **Capítulo 2 - Fundamentação e Conceitos:** enumera-se as fundamentações teóricas, justificativas técnicas e conceitos introdutórios necessários ao entendimento científico das intenções deste trabalho;
- **Capítulo 3 - Desenvolvimento do Projeto**: desenvolvem-se os modelos e as soluções de projeto de cada objeto do *framework*, necessários à criação de plataforma de microserviços, enumerando vantagens técnicas e efetivas, seus contextos, problemas, vantagens, estrutura lógica, dependências, padrões relacionados e exemplos de funcionamento e diagramas necessários;
- **Capítulo 4 - Aplicações e Resultados:** demonstra-se a usabilidade deste modelo de serviço no mundo real, aplicando situações de concorrência de usuários, escalabilidade, segurança, disponibilidade, além de serviços, instituições, indivíduos e outras tecnologias que podem beneficiar-se deste modelo de plataforma e desenvolvimento de sistemas, comparando-o tecnicamente com os paradigmas utilizados atualmente;

1.5 TRABALHOS RELACIONADOS

Na década de 90, especialistas em orientação a objetos, arquitetura e arquitetura de sistemas começavam a utilizar o termo *Framework* para designar o ato de criar métodos para reutilização de design de projetos e objetos codificados.

Em 2014, Chris Richardson estava publicando um artigo intitulado ***Pattern: Microservices Architecture***, e o resultado deste trabalho foi a criação de sua própria plataforma chamada ***Eventuate***, onde ele aplica os conceitos de microserviços para resolver problemas de gerenciamento de dados distribuídos.

No mesmo ano, a ***Amazon Cloud Computing*** começou a dar suporte ao chamado ***Serverless Computing***, que apesar do nome, utiliza servidores para executar funções sob demanda, o que é algo muito parecido com a ideia de executar *stored procedures* de bancos de dados, mas seu objetivo era poder cobrar os usuários por cada chamada a estas funções, ao invés de tráfego ou tempo de uso.

Em 2016 a **IBM** lançou o produto ***OpenWhisk***, que é uma plataforma serverless com código aberto, assim como a ***Google Cloud Functions***, que segue a mesma ideia da IBM.

Desde 2003 a ***Oracle Corporation*** desenvolve e distribui o produto ***Application Express* (APEX)**, nomeado anteriormente de Oracle HTML DB, cujo principal propósito é permitir o desenvolvimento rápido de aplicações voltadas para aquisição, edição e apresentação de dados, e funciona exclusivamente com o banco de dados proprietário da Oracle. Seu funcionamento é bem simples e baseia-se em janelas do tipo *Step-by-step Wizard* (como passos de mágica) para criar as janelas, campos dos formulários e relatórios para conectá-los às tabelas do banco de dados. Arquitetura bem parecida com a prática de desenvolvimento sem APIs e com o serviço de modelagem de saída para clientes, também proposto neste trabalho.

Todos estes trabalhos seguem uma filosofia de baixo acoplamento, economia de recursos, independência de serviços e distribuição de processamento. Mas não foram encontrados, até agora, trabalhos concorrentes ao que se propõe este Projeto de *Framework*, especialmente na forma como sugere o uso de bancos de dados como fonte de funções acessíveis sem APIs.

1.6 RESULTADOS ESPERADOS

Após o desenvolvimento do projeto deste *framework*, espera-se que as plataformas desenvolvidas com ele sejam capazes de:

- Estimular a quebra de sistemas monolíticos em Microserviços distribuíveis pela rede, mudando o foco do desenvolvimento na camada de controle da aplicação e descentralizando-a para processar os modelos de negócio diretamente nos bancos de dados, assim como processar a interface de usuário totalmente na camada do cliente;
- Diminuir o tamanho do servidor (serverless);
- Diminuição da curvas de aprendizagem gerais e específicas;
- Diminuição do consumo dos recursos de hardware e software na execução da Plataforma e seus serviços instalados;
- Padronização do acesso aos recursos de serviços acoplados;
- Padronização da administração de recursos, serviços e usuários;
- Aumento da segurança e integridade dos bancos de dados envolvidos;
- Aumento da segurança na troca de informações entre clientes e servidores;
- Aumento da produtividade no desenvolvimento de aplicações.

2 **FUNDAMENTAÇÃO E CONCEITOS**

Para perceber a necessidade de criação do tipo de plataforma proposta neste trabalho, é necessário entender os mecanismos de funcionamento de sistemas Web como são atualmente. Neste capítulo serão mostrados o funcionamento das linguagens de programação e seus *frameworks* para a Web, os modelos de servidores de aplicações mais utilizados, e os conceitos das tecnologias disponíveis que são usadas hoje e as necessárias para desenvolver este projeto.

2.1 A NATUREZA DAS LINGUAGENS DE PROGRAMAÇÃO

Uma das principais metas das linguagens de programação é que programadores tenham uma maior produtividade, permitindo expressar suas intenções mais facilmente do que quando comparado com a linguagem que um computador entende nativamente (código de máquina). Assim, as linguagens de programação são projetadas para adotar uma sintaxe de nível mais alto, que pode ser mais facilmente entendida por programadores humanos. São ferramentas importantes para que programadores e engenheiros de software possam escrever programas mais organizados e com maior rapidez (NORTON; AITKEN; WILTON, 1993, com adaptações).

2.1.1 **Interpretação e compilação de linguagens de programação**

Uma linguagem de programação pode ser convertida (ou traduzida) em código de máquina por compilação ou interpretada por um processo denominado interpretação. Em ambas ocorre a tradução do código fonte para código de máquina. Se o método utilizado traduz todo o código fonte do programa, para só depois executá-lo, então afirma-se que o programa foi compilado e que o mecanismo utilizado para a tradução é um compilador (que por sua vez nada mais é do que outro programa). A versão compilada do programa tipicamente é armazenada, de forma que o programa pode ser executado um número indefinido de vezes sem que seja necessária nova compilação, o que compensa o tempo gasto na compilação e aumenta seu desempenho. Isso acontece com linguagens como Pascal e C. Este código fonte passa por várias camadas de tradução antes de ser convertido ao código de máquina da arquitetura em que será executado (SOMMERVILLE, 2011).

Se o texto do programa é executado à medida que vai sendo traduzido, como em JavaScript, Python ou PHP, num processo de tradução de trechos seguidos de

sua execução imediata, então afirma-se que o programa foi interpretado e que o mecanismo utilizado para a tradução é um interpretador (outro programa). Programas interpretados são geralmente mais lentos do que os compilados, mas são também geralmente mais flexíveis, já que podem interagir com o ambiente mais facilmente e não possui forte acoplamento com a memória física da máquina que o executa (SOMMERVILLE, 2011, com adaptações).

É neste contexto que foi surgindo a ideia de criar máquinas virtuais, para facilitar ainda mais o processo produtivo, a execução e o gerenciamento de sistemas num ambiente mais controlado, em termos de especificações físicas e lógicas. Segundo DEITEL (2010), uma máquina virtual é um programa de computador que simula (emula) um computador real, o qual utiliza partes dos recursos do computador hospedeiro. Algumas linguagens de programação, como Java e Scala, ao invés de terem seu código compilado ou interpretado para uma arquitetura de máquina real, são interpretados ou compilados para a arquitetura da máquina virtual. Isto permite que um mesmo programa possa ser executado em qualquer arquitetura (hardware e sistema operacional) que possua uma versão executável desta máquina virtual. Segundo Smith (2016), esta vantagem ajudou a tornar Java à linguagem de programação mais utilizada atualmente.

2.2 A LINGUAGEM JAVA E SEUS *FRAMEWORKS*

Java é uma linguagem de programação criada nos anos 1990 na empresa Sun Microsystems, tendo como foco a resolução de problemas comuns às linguagens de programação na época, como: gerenciamento de memória e ponteiros; uma base de código diferente para cada sistema operacional e/ou arquitetura de hardware; e o alto custo financeiro de ferramentas proprietárias de desenvolvimento de software (DEITEL, 2010).

Um dos objetivos específicos da Sun em criar uma plataforma com sua própria linguagem era permitir seu uso em dispositivos simples como TVs, videocassetes, liquidificadores, geladeiras, carros entre outros (CAELUM, 2016). Mas em pouco tempo a empresa mudou o foco para desenvolvimento de aplicações seguras para a Web com a criação do Applet Java executando miniaplicativos do lado do cliente, e logo depois passou a figurar entre uma das plataformas mais utilizadas em servidores Web. Em 2016, de acordo com o site oficial do Java, esta plataforma é utilizada em mais 2,5 bilhões de dispositivos móveis, entre celulares, GPS e outros, cumprindo seu objetivo inicial.

2.2.1 **Plataforma Java EE**

Diferenças importante que devem ser consideradas são a Linguagem Java e a Plataforma Java. Esta última é o conjunto de recursos necessários para executar aplicações escritas em Java (e outras linguagens como Groovy e Scala). Trata-se de seu compilador, um interpretador (a máquina virtual Java), bibliotecas de classes, que permitem ao Java acessar bancos de dados, Internet e recursos de hardware e do sistema operacional etc., além de ferramentas específicas para suas edições: Standard, Enterprise, Micro, Card e FX.

A Java Enterprise Edition é a edição da plataforma Java voltada para o desenvolvimento de aplicações corporativas e para a Internet, e é objeto de estudo deste trabalho, como objeto de comparação de desempenho, praticidade e outros quesitos, em relação à arquitetura aqui proposta. Suas principais ferramentas são: o servidor de aplicações Web, que é um sistema que controla todas as aplicações que respondem às requisições de usuários remotos e executa os programas Java e serviços dos quais eles dependem; e os *frameworks* desenvolvidos para facilitar tarefas repetitivas e abstrair modelos primitivos da linguagem Java em formas mais modernas de desenvolvimento. Estes *frameworks* foram desenvolvidos, em sua maioria, por uma comunidade internacional de programadores e entusiastas da linguagens, e alguns deles foram tão bem disseminados e aceitos que acabaram por se tornar padrões de mercado e registrados como padrão oficial pela comissão responsável pelo Java e sua plataforma (GONÇALVEZ, 2007, com adaptações).

O Java EE foi concebido para resolver problemas inerentes ao desenvolvimento de aplicações para o ambiente da Internet, que o Java sozinho não tinha capacidade de suportar. Além disso, foi levado em consideração que, uma padronização era necessária para evitar que cada desenvolvedor criasse sua própria solução para estes problemas, pois a padronização com código aberto é um dos principais motivos pelo qual o Java é tão popular: evitar reinventar a roda, para que o foco seja a produtividade e o modelo de negócio da aplicação desenvolvida; mas o fato de a comunidade poder participar nos rumos das especificações e enviar solicitações de mudança é outra característica de sucesso da plataforma (AQUINO JR, 2002). Neste sentido, os principais problemas do desenvolvimento Web a serem gerenciados pela plataforma Java EE são, de acordo com (SOARES, 2011):

- **Concorrência**: os aplicativos Web podem ser acessados por muitos usuários simultaneamente, e isso exige um maior controle por parte da aplicação para garantir que o sistema se comporte de forma eficiente e segura. A definição de métodos que guiem o tratamento da concorrência em sistemas Web é essencial para a execução de programas em ambiente concorrente;

- **Escalabilidade**: é a capacidade de um serviço na Web, e sua infraestrutura como um todo, suportar o aumento do número de usuários, requisições, espaço em disco, acesso à memória, processamento, relações e conteúdo de banco de dados e outras características, sem interferir no seu funcionamento regular. Em outras palavras, é quando está preparado para *crescer*;
- **Segurança**: sistemas web podem ser acessados por usuários remotos de qualquer parte do planeta, o que exige maior comprometimento com o controle de acesso e outras medidas de segurança passivas e ativas, para prevenir acesso não autorizado e invasões maliciosas;
- **Disponibilidade**: é a capacidade de operar por períodos indefinidos, sem interrupção, ou com interrupções mínimas e programadas, apenas para manutenção ou atualização de componentes, com o objetivo de nunca comprometer a capacidade de acesso aos usuários e outros serviços que dependam de sua infraestrutura web.

2.2.1.1 Camadas da Plataforma Java EE

Esta plataforma especifica quatro camadas para desenvolvimento de sistemas Web: **Cliente, Apresentação, Negócio e Dados**. Conforme as versões foram avançando, novos *frameworks* foram desenvolvidos e se tornaram padrão oficial para facilitar o trabalho de desenvolvedores em cada uma destas camadas. O Java EE é como um casulo que envolve os aplicativos Java antes, durante e após sua execução. O **Quadro 1** mostra um resumo desta estrutura:

- **Camada Cliente**: tem o papel de uma interface de entrada e saída para interação do sistema com o usuário e é executada na máquina do cliente. Esta camada, em aplicações Web, é implementada com apoio do Web browser, que tem basicamente o papel de interpretar e apresentar o conteúdo gerado pela Camada de Apresentação (geralmente HTML e JavaScript). Ela interage com a apresentação utilizando protocolos como HTTP e HTTPS;
- **Camada de Apresentação ou Web**: é a primeira camada do servidor de aplicação e tem o papel de disponibilizar os serviços da Camada de Negócio para o ambiente Web, oferecendo conteúdo estático e conteúdo dinâmico gerado pelos componentes Web. Geralmente o conteúdo gerado por esta camada é HTML, mas a mesma pode gerar qualquer formato suportado pelo protocolo HTTP, tais como XHTML, WML, XML etc.;
- **Camada de Negócio**: neste modelo, representa o núcleo do sistema, e é nela onde estão implementadas todas as regras de negócio da aplicação. Ou seja: as ações que geram ou mudam o estado das informações sensíveis;

- **Camada de Dados**: é responsável pelo gerenciamento dos dados do sistema. Pode ser vista como a infraestrutura necessária para gerenciamento dos recursos da aplicação. O SGBD é um exemplo de infraestrutura localizada nesta camada.

Quadro 1 - Camadas do Java EE e algumas de suas especificações

Localização Física	**Camada da Arquitetura**	**MVC**	**Tecnologias e *Frameworks***
Máquina do Cliente	Camada do Cliente	Não	SO, Browser, HTML, JS
Servidor de Aplicação	Camada de Apresentação	Sim	Servlets, JAX, JSP, JSTL, JSF...
Servidor de Aplicação	Camada de Negócio	Sim	JAAS, EJB, JPA, JMS, RMI...
Servidor de BD	Camada de Dados	Sim	JDBC, DAO, HIB, ORM, JPQL...

Fonte: Alura (2016)

2.2.2 *Frameworks* necessários para desenvolver aplicações na Web

Segundo Gonçalvez (2007), os *frameworks* tornaram-se uma parte imperativa no desenvolvimento de sistemas para a Web, principalmente para a plataforma Java, onde se tornaram especificações oficiais, e são tidos quase que como parte da própria linguagem. Mesmo aquelas linguagens cuja aurora baseou-se na utilização em servidores para desenvolvimento de aplicações distribuídas precisam de muito retrabalho para desenvolver funcionalidades simples ou que sejam padrão em qualquer aplicativo Web atualmente.

Serão abordados cada um dos principais *frameworks* necessários no desenvolvimento Web em Java, mas apenas um por função, e levando em consideração seu destaque como especificação oficial na versão 7 do Java EE. Também é importante destacar que, cada *frameworks* citado acaba dependo do uso de outros *frameworks e/*ou bibliotecas. O **Quadro 2** cita os frameworks dentro de cada categoria, postergando os detalhes de seu funcionamento ao **Apêndice C**, a fim de evitar que se estenda demais o assunto neste capítulo.

Quadro 2 - Frameworks Java por categoria de função

Categoria	***Frameworks***	**Dependências**
Conectividade	Servlets	Java EE, Java Web Container
	Java Database Connectivity	Bibliotecas de Bancos de Dados
Persistência	Java Persistence API	JDBC
	Hibernate ORM	JDBC e JPA
	Enterprise JavaBeans	Java EE
Interface	JavaServer Pages	Java EE, Web Container, Servlets
	JavaServer Faces	Java EE, Servlets, JSP, EJB
	Componentes de JSF	JSF
Multitarefa	Spring MVC	Java EE, Servlets, JPA, EJB
	Spring IoC	Java EE

Fonte: O autor (2017)

2.2.3 **Vantagens e desvantagens de Java EE**

Desde sua versão 1.2 de 1999 até sua versão 1.7 de 2013, o Java EE passou de 10 especificações (entre APIs e *frameworks*) para 40 especificações oficiais, e este número crescerá ainda mais quando a versão 1.8 for oficializada em 2017 com a expectativa pelo menos mais 10 novas especificações (WIKIPÉDIA, 2016), totalizando um aumento de quinhentos por cento em seu tamanho.

Este enorme crescimento representa um consumo extra de recursos de hardware e software nas camadas do cliente e do servidor, o que é preocupante em termos de investimento e capacidade de escalabilidade da infraestrutura e também obriga o cliente a ter maiores gastos com equipamentos mais potentes e maior consumo de energia com a execução de milhares de linhas de código adicionais.

Em relação ao fator humano, este número exagerado de especificações e *frameworks* de apoio obriga os profissionais de desenvolvimento de sistemas a se especializarem demais na plataforma Java, ao mesmo tempo em que precisam conhecer e dominar todas estas tecnologias para poderem trabalhar no mesmo padrão de mercado. A curva de aprendizado desta plataforma é muito alta, o que requer recursos financeiros para treinamento, certificação, além de tempo para assimilar todo este conteúdo. O tempo ganho (com a produtividade da plataforma) se perde com o tempo que se leva para atingir o nível de maturidade necessário para desenvolver de acordo com as especificações e padrões de mercado.

Em resumo, a plataforma Java EE acaba se tornando uma solução monolítica para desenvolvimento de sistemas Web, uma vez que cada sistema desenvolvido nesta plataforma precisa utilizar muitas especificações de cada camada que for desenvolvida, e cada especificação é dependente de outra(s), tornando qualquer programa desenvolvido num produto padronizado que, precisará de ajustes muito grandes caso algum *framework* seja retirado, adicionado ou tenha sua versão atualizada (GUPTA, 2015). A maior parte do código acaba sendo de autoria de terceiros, ao invés de ser um produto da equipe de desenvolvimento. O conhecimento completo de todas as suas estruturas de funcionamento é, apesar de possível, extremamente improvável, dada a enorme quantidade de bibliotecas criadas para cada *framework*.

2.3 A COMPUTAÇÃO DISTRIBUÍDA

A arquitetura cliente/servidor, onde muitos computadores buscam informações e serviços em um computador central, aliado à capacidade de hipertexto da arquitetura Web, onde uma página pode enviar o usuário a várias outras páginas, em servidores diferentes, de países diferentes, é uma ilustração sólida do que é o paradigma da computação distribuída.

Assim como o cliente recebe as informações de forma transparente, como se todo o conteúdo viesse de um só lugar – a Web – o próprio servidor pode se beneficiar da computação distribuída quando também pode receber recursos (bancos de dados, repositórios multimídia, serviços) de diferentes origens, sejam elas locais ou remotas.

Segundo Andrew Tanenbaum (2007), é uma “coleção de computadores independentes que se apresenta ao usuário como um sistema único e consistente”. O princípio deste paradigma consiste em executar um único programa em vários computadores ao mesmo tempo. Melhor dizendo, diferentes objetos ou elementos de um programa são executados em processadores diferentes com o objetivo de solucionar o mesmo problema.

2.3.1 **Balanceamento de carga**

Outro conceito de computação distribuída de interesse deste trabalho é o de balanceamento de carga, que é uma técnica para distribuir a carga de trabalho uniformemente entre servidores da rede que estejam preparados para executar os serviços requisitados pelas máquinas clientes (KUROSE; ROSS, 2012).

Este procedimento pode ser feito de várias formas, dependendo de como é a solução desenhada para tal rede. Existem soluções padrões de mercado, assim como soluções customizadas mais exclusivas. O balanceamento de carga, quando bem desenvolvido e configurado, pode aumentar consideravelmente a disponibilidade de um serviço na rede.

Um exemplo popular é o do serviço de streaming de vídeos chamado Netflix. Nele, o cliente acessa o serviço por meio de computadores, celulares ou SmartTVs. O sistema que responde à primeira requisição está em um servidor HTTP, ou seja, trata-se de um aplicativo Web, onde o cliente se autentica para poder listar as opções de filmes e séries disponíveis. Quando este seleciona um vídeo para assistir, o sistema da Netflix então determina qual servidor de streaming é fisicamente mais próximo deste cliente. Assim, se o usuário estiver no Brasil, ele será conectado a um Servidor de Conteúdo Multimídia no Brasil ou de outro país da América do Sul, para maximizar as chances de uma conexão rápida e estável. O endereço deste servidor é passado ao aplicativo cliente, que faz então duas conexões diretas com este servidor: uma para enviar e receber comandos, e outra exclusivamente para receber os pacotes de streaming. Para o usuário do Netflix o processo é totalmente transparente, ou seja, ele não tem domínio ou conhecimento de quais partes do mundo estão chegando tais informações requisitadas, nem mesmo o fato de que vários servidores estão envolvidos.

2.3.2 **Microserviços**

Este conceito, como posto por Tanenbaum (2011), ratifica a percepção de um sistema como uma unidade monolítica. O objetivo deste trabalho é quebrar este conceito, aplicando o uso da computação distribuída não como fonte de serviços de software, mas como fonte de serviços para softwares.

Estes são chamados de Microserviços, e são fruto da transformação de uma ou mais aplicações monolíticas em blocos lógicos de software especialistas, e pela inversão de controle de certas camadas do software, como bancos de dados e interface com o usuário, que agora fazer papel de serviço e cliente do serviço, exatamente nesta ordem.

Alguns autores, como MACVITTIE (2016), vão mais além ao dizer que, até os Microserviços podem ser quebrados em funções, aumentando mais ainda a independência entre os módulos de uma plataforma de serviços distribuídos, conforme mostra a **Figura 1**. Este conceito é conhecido como ***Serverless*** (sem servidor), como um nome apenas simbólico, já que ainda é necessário um servidor para receber as requisições de funções e entregar seus resultados aos clientes.

Este conceito de funções é apresentado neste trabalho, à medida em que transforma-se cada banco de dados disponível em um serviço, com acesso restrito a suas *stored procedures*, onde é possível considerar cada uma como uma função independente deste serviço, onde o serviço existe apenas como um item de configuração na plataforma.

Figura 1 - Quebra de arquiteturas, de monolítica até serverless

Fonte: MACVITTIE (06/2016)

2.4 CONSIDERAÇÕES FINAIS DESTE CAPÍTULO

Os *frameworks* foram criados com a proposta de facilitar a vida dos desenvolvedores, adicionando ferramentas, bibliotecas de funções e classes e padrões de desenvolvimento para que o projeto seja conciso e siga uma metodologia de projeto e codificação que possa ser entendida e continuada por qualquer equipe que saiba trabalhar com aqueles *frameworks* escolhidos no projeto.

Isso significa que, sem os *frameworks* reunidos nas especificações do Java EE, esta linguagem de programação apresentaria as mesmas dificuldades que qualquer outra linguagem apresenta como ferramenta de desenvolvimento para um padrão (web) que foi concebido muito tempo depois da invenção da linguagem, como Pascal, Fortran e Cobol por exemplo.

3 **DESENVOLVIMENTO DO PROJETO**

Neste capítulo será desenvolvido o projeto de criação do *framework* para a plataforma de microserviços distribuídos, abordando as especificações das camadas de mecanismo de núcleo (*kernel engine*), de configuração, de banco de dados mínimo para funcionamento da plataforma e, por fim, de cada uma das modalidades de serviços que já foram idealizadas para a plataforma.

3.1 A ARQUITETURA DA PLATAFORMA

A plataforma se baseia numa arquitetura especificada pelo seu *Framework*. Uma arquitetura definida por serviços distribuídos previamente configurados na plataforma, com identificação, recursos e locais alternativos onde possam ser encontrados em caso de indisponibilidade de sua rota padrão.

Cada servidor que execute a plataforma estará seguindo as regras do *framework*, independente de quais serviços em comum ou exclusivos estejam sendo disponibilizados. Isto permite uma consistência necessária para que possam se comunicar, além de manter um padrão único capaz de popularizar seu uso como uma ferramente estável e consistente, não importando em que sistema operacional esteja sendo executado, nem em que linguagem de programação cada versão desenvolvida. Isso nos leva a seguinte proposição:

Se uma versão rodando em uma empresa no Japão foi compilada em Pascal para Windows, e outra versão rodando numa faculdade na Alemanha está executando em um contêiner Java para Linux - desde que as especificações do *framework* tenham sido seguidas, pode-se considerar que é a mesma plataforma e que elas podem se comunicar e compartilhar serviços.

3.1.1 **Funções da Plataforma**

Dois objetivos primários desta plataforma são: diminuição da curva de aprendizado no desenvolvimento de *softwares* (serviços) que comporão a plataforma, assim como a economia no consumo de recursos de tecnologia, como energia elétrica, *hardware* e sistema operacional. Por isso, a arquitetura da plataforma deverá permitir que algumas de suas funções possam ser desativadas, sendo que sua implementação poderá ser facultativa, sem prejudicar a integridade e capacidade de executar os serviços que seu administrador deseje disponibilizar e sem descaracterizar a plataforma (ferindo os princípios de seu *framework*).

- **Oferecer comunicação criptografada independente de HTTPS:** é excelente para serviços que precisem de privacidade aos usuários e instituições, mas pode ser dispensado em plataformas com serviços públicos que apenas informam, como por exemplo API de endereços por CEP;
- **Permitir o mesmo banco de dados às aplicações do *Web-Service*:** para servidores de pequeno porte, apenas um banco de dados usado tanto para a plataforma quanto para todos os microserviços e aplicações oferecidos por ela pode ser uma vantagem pela facilidade de configuração, mas é uma decisão de design do administrador da plataforma e seus desenvolvedores;
- **Oferecer *framework* MVC para classes de serviços das aplicações:** se a linguagem de programação usada no desenvolvimento da plataforma for interpretada ao invés de compilada, é possível usar a arquitetura MVC para criar microserviços e funções que funcionem juntamente com o Kernel da plataforma, mas seguindo as boas práticas de divisão de camadas. Caso não seja possível inserir mais classes controladoras na plataforma por ela estar compilada, isto não impede o uso de microserviços externos;
- **Controlar as sessões de acesso temporário na plataforma:** esta função é desnecessária caso não haja nenhum microserviço ou aplicação que trabalhe com sessões de usuário através de cookies da Web;
- **Controlar as sessões duráveis no banco de dados:** esta função é desnecessária caso não haja nenhum microserviço que trabalhe com sessões mantidas através registros em banco de dados de sessões, usadas principalmente para fornecer dados para aplicativos móveis.

As funções que devem ser obrigatoriamente implementadas para manter a consistência do *framework* da plataforma, são estas:

- **Ter um painel de controle para administradores dos serviços:** o painel de controle é muito importante para manter baixa as curvas de aprendizado de operação e configuração da plataforma, pois é muito mais fácil e prático administrar servidores e plataformas de serviços por meio visual do que utilizando terminais de comandos, o que não impede que uma versão de administração via console seja implementada;
- **Especificar o modelo de usuários, perfis e permissões:** mesmo que nenhum serviço, microserviço ou aplicação dentro da plataforma requeira uma autenticação de usuário para fornecer informações, ainda são necessárias credenciais administrativas e operacionais para a manutenção da plataforma, pois nenhum servidor remoto deve estar completamente aberto a acesso anônimo, o que seria uma grave falha de segurança;

- **Padronizar requisição de dados, arquivos e controles de serviços:** todos os dados solicitados e enviados de/para a plataforma devem seguir o mesmo padrão, independente de qual o recurso alvo, se um serviço ou mídia, se via HTTP, TCP ou UDP - todas as plataformas desenvolvidas com base neste *framework* deverão obedecer o mesmo padrão;
- **Permitir a integração de várias aplicações e módulos na plataforma:** a plataforma deve permitir uma integração entre todos os serviços disponíveis nela, assim como acontece num sistema operacional multitarefa, fazendo solicitações e trocando informações usando os protocolos disponíveis;
- **Oferecer ao cliente um catálogo de serviços:** a plataforma deve ser capaz de informar ao cliente que requisita, uma lista, no formato XML ou JSON, de todos os serviços públicos disponíveis, informando o caminho de acesso, os parâmetros, disponibilidade de criptografia e necessidade de autenticação, para cada serviço declarado na sua configuração.

Esta proposta de plataforma pode, a princípio, ser comparada com um aplicativo servidor HTTP, como Apache, por exemplo. Mas as similaridades se limitam a o trabalho com este protocolo Web e com a integração modular com outros aplicativos de serviços de aplicação como PHP e Ruby.

A plataforma se destaca no fato de que também opera sem HTTP, usando protocolos de rede como TCP e UDP (entre outros implementáveis) para receber e enviar requisições, o que retira vários limites técnicos. Além disso, esta plataforma pode, em algumas das modalidades de serviço, controlar diretamente os recursos que serão trabalhados, ao invés de delegá-los a módulos externos, o que dá mais segurança e garantia de disponibilidade aos clientes.

3.1.2 **Modalidades de Serviços da Plataforma**

A plataforma deve ser desenvolvida de forma a permitir virtualmente qualquer modalidade de serviço que possa ser oferecido num ambiente cliente-servidor. Isso significa que mesmo após a delimitação inicial de quais modalidades serão projetadas como padrão do *framework* qualquer desenvolvedor terá liberdade para projetar e desenvolver novas categorias de modalidade para a sua plataforma, de acordo com sua necessidade de negócio.

Inicialmente, o *framework* prevê oito (8) modalidades de serviços que contemplam as funcionalidades que satisfazem a maioria das necessidades dos usuários de recursos remotos.

Para satisfazer as regras de computação distribuída como balanceamento de

carga e disponibilidade, estas modalidades de serviços podem estar espalhadas por vários computadores ou até mesmo em várias redes diferentes, bem como possuir cópias de seus componentes executáveis e de seus recursos por entre estes ativos de rede. Toda esta malha deverá ser devidamente declarada nas configurações da plataforma ativa e de suas rotas de segurança e de balanceamento.

A seguir é apresentada a lista dos tipos de serviços oferecidos inicialmente:

- **Sessões e Acesso:** controla sessões de usuários ativos da plataforma, mantendo informações temporárias de aplicativos para estes usuários e verificando a identificação do cliente que está usando a sessão, para manter a conexão com um certo nível de segurança;
- **Arquivos e Armazenamento:** controla o acesso a recursos guardados, como texto, documentos, imagens, vídeos e outros tipos de arquivos binários. Este acesso é bidirecional e se dá por meio de download, upload e, no caso de arquivos não-binários, por meio de pesquisa com expressões regulares;
- **Bancos de Dados:** controla recursos processáveis de dados e modelos de negócio. A plataforma oferece acesso aos bancos de dados de forma diferente do usual. Ao invés de permitir o acesso a tabelas e views, o cliente só pode solicitar a execução de *stored procedures* e *funções*, que irão executar alteração de dados ou retornar uma tabela de informações. Isso aumenta a segurança do serviço na plataforma e permite criar microserviços sem precisar criar uma API para recuperar quaisquer dessas informações;
- **Módulos e Aplicativos:** controla recursos executáveis de aplicativos e microserviços. São aquelas funções que não podem ser programadas diretamente no banco de dados, pois são mais específicas, como por exemplo o envio de um e-mail ou a emissão de uma nota fiscal. Estes microserviços podem ser arquivos executáveis acoplados ou classes anexadas no código da plataforma, dependendo da linguagem utilizada para desenvolvê-la;
- **Painéis e Consoles:** ferramentas administrativas já inclusas para configurar a plataforma de maneira visual ou através de console. Estes serão desenvolvidos de forma modular, para evitar permitir a ativação ou desativação de acordo com os tipos de serviço disponíveis na plataforma;
- **Mensagens:** recursos de comunicação entre usuários de microserviços e aplicativos que utilizam o serviço de mensageria, para receber e enviar mensagens privadas ou de grupos de discussão, similar a WhatsApp, ICQ, Microsoft Messenger, Telegram, Google Groups, Yahoo Groups etc.;
- **Modelos de Saída para o Cliente:** scripts que recebem a saída de outros serviços e a insere em modelos visuais para o cliente. Uma forma de automatizar a resposta ao cliente que solicita um microserviços baseado

numa função de banco de dados, por exemplo. Desta forma o administrador da plataforma pode rapidamente criar novos serviços, criando apenas a lógica de negócio diretamente no banco de dados e depois criar uma apresentação para estes dados na forma de um template - o que é bem mais simples que desenvolver uma nova aplicação ou várias classes para lhe dar com as informações e sua experiência com o usuário;

- **Streaming Multimídia:** disponibiliza recursos de transmissão de dados contínuos com canais separados para controle e transmissão, de uma maneira mais simples que a habitual, onde o cliente precisa conhecer previamente as portas de comunicação e os comandos de controle da mídia

As modalidades citadas acima serão referenciadas e especificadas neste trabalho, sendo que sua implementação será abordada nos trabalhos consecutivos a este, a título de dissertação e tese, juntamente com outras novas modalidades de serviço que por ventura sejam exploradas no futuro.

3.2 O LADO DO SERVIDOR

Existem muitas arquiteturas para o desenvolvimento de *software* num ambiente cliente-servidor. O modelo mais comum, por ser um dos primeiros, e usado há décadas com várias linguagens como PHP, Perl, Java, Pascal entre outras, é aquele onde todo o código executável fica hospedado do lado do servidor, variando apenas no método geração da saída de dados para o cliente - que pode ser totalmente gerada dentro do código de programação ou estar disponível como templates HTML ou XML nos quais as informações dinâmicas são inseridas em pontos específicos destes arquivos (FORRISTAL, 2002). Este modelo é a definição do que é uma arquitetura monolítica, de difícil manutenção, onde seus componentes são extremamente dependentes um do outro.

Um modelo mais recente e moderno baseia-se na divisão de camadas, onde uma parte do trabalho de processamento é feito no servidor e a outra é feita no cliente. Isso foi possibilitado pela ascensão da linguagem JavaScript, que é executada nos navegadores Web e nos smartphones como alternativa aos aplicativos nativos das plataformas móveis (SESHADRI; GREEN, 2014).

Desta forma, o desenvolvedor pode desenvolver toda a interface com o usuário usando JavaScript como linguagem de programação, que executa na máquina do cliente, e faz requisições de informações do servidor, que agora se encarrega apenas das camadas de controle e modelo de negócios, e retorna ao

cliente os dados necessários cruamente, para que o código no cliente possa mudar a interface e exibir os dados sem a necessidade de que o servidor crie isto dinamicamente. Essa solução economiza uma grande carga de processamento do servidor, que agora é distribuída entre os clientes.

O *framework* proposto se beneficia desta arquitetura, visando a economicidade dos recursos do servidor; a modularidade para divisão de responsabilidade das equipes de desenvolvimento, interface e banco de dados; e à sua disponibilidade nas plataformas Web, Móvel e Desktop - mas também com um diferencial, que é:

- Buscar a desvinculação dos conceitos de API e Camada de Transporte HTTP, permitindo acessar os modelos de negócio sem a necessidade de construir uma nova função de API para cada novo modelo criado, permitindo trabalhar estes modelos como microserviços, ou como partes ainda menores, as funções sem servidor (*serverless functions*).

Uma API é uma biblioteca ou sistema com um conjunto de comandos disponíveis para uma aplicação ou mais aplicações; ou para um ou mais sites ou serviços na Web. Neste último caso, a API sempre depende do protocolo HTTP para receber as requisições a esses comando e subsequentemente enviar a saída gerada pelo processamento destas requisições (FORRISTAL, 2002). Por conta disso, a grande maioria das soluções de *software* abraça o protocolo HTTP e o controla de dentro de cada função de sua API para traduzir a requisição do cliente, unindo as duas tecnologias e tornando-as dependentes.

Nossa proposta para atingir este objetivo no *framework* é desenvolver um gerente de requisições que seja responsável por todas as etapas de cada requisição, com uma API própria que:

- Permita executar os microserviços e obter suas respostas, de forma transparente ao protocolo usado na requisição, isentando-os da necessidade de controle da camada de transporte e permitindo o uso de vários protocolos para acesso a estes recursos.

Este gerente de requisições é o mecanismo central da plataforma, pois é nele que são capturadas todas as requisições dos clientes e delega que microserviço será executado para atender esta demanda.

3.2.1 **Gerente de Requisições**

O Gerente de Requisições (GRP) não é um web-service. Mas ele possui, entre outras, a função de web-service. Ao mesmo tempo que deve responder requisições vindas através dos protocolos HTTP e HTTPS (que são os principais protocolos da Web), também deve responder às requisições vindas através do protocolo TCP e UDP, independentemente do serviço que o cliente esteja buscando.

3.2.1.1 Parâmetros básicos para envio de requisições na Plataforma

Para acatar uma requisição do cliente, a plataforma necessidade de uma quantidade mínima de informações obrigatórias. Há também outras informações opcionais, que permitem a customização dos resultados a serem obtidos.

Estas informações devem ser passadas junto com as requisições através de parâmetros, cada um com um nome exclusivo para não causar ambiguidade na interpretação pelo Gerente de Requisições. O **Quadro 3,** mostrado a seguir, define os parâmetros globais obrigatórios e opcionais.

Quadro 3 - Parâmetros básicos para requisições da plataforma

Identificador	Função	Opcional	Valores
categoria	Especifica qual a modalidade de serviço será solicitada	Depende	sessão, arquivo, banco, modelo, painel...
serviço	Informa qual microserviço ou *sequência* deverá ser executado pela plataforma	Não	Nome que identifique o microserviço
ação	Especifica qual ação dentro do microserviço será executada	Sim	Depende do tipo de serviço (ex: login)
entrada	Especifica qual é o formato dos dados de entrada, caso existam	Sim	json, xml, csv, plain, txt, bin, zip
saída	Instrui a plataforma a devolver a resposta em um formato específico	Sim	json, xml, csv, pdf, xls, html, txt, bin, zip
sessão	Informa um código único da sessão durável que está sendo usada	Depende	O hash único da sessão durável
servidor	Informa que a requisição é interna e vem de outro servidor que a redirecionou para balanceamento	Depende	O hash único do servidor origem

Fonte: O autor (2016)

Alguns parâmetros marcados como opcionais, se devem ao fato de que na configuração de cada microserviço registrado na plataforma deve constar quais os seus tipos padrões de entrada e saída, e quais os tipos permitidos além destes. Se o cliente omitir uma preferência, o padrão será adotado.

- Quando uma ação é omitida, a plataforma executa a ação configurada como padrão para aquele serviço, caso mais uma ação esteja disponível;
- Alguns microserviços podem solicitar obrigatoriamente que uma sessão esteja aberta e que o usuário a identifique na requisição;
- Especificar qual a categoria de serviço é obrigatório apenas se a plataforma estiver configurada para permitir nomes de serviços iguais em categorias diferentes. Mas mesmo caso isto seja proibido em uma plataforma, identificar o tipo de serviço pode ser uma boa prática para entender melhor o que será solicitado na requisição e qual tipo de resposta esperar do servidor.

3.2.1.2 Especificações para Formato de Requisições

Aceitar requisições HTTP permite à plataforma aproveitar a simplicidade e a padronização universal que este protocolo aplica ao ambiente Web como camada de transporte. Isto não só permite que a plataforma possa servir páginas HTML, como também JSON e XML, premissas para sua utilização como API para troca de dados utilizada por aplicativos remotos, sejam eles Desktop ou Móveis. As requisições são feitas através de um Identificador Uniforme de Recurso (URI) que determina a localização de um servidor, assim como a localização de um recurso dentro deste servidor: **esquema://domínio:porta/caminho/recurso?query_string#fragmento**.

A parte da URI que começa no esquema até a porta é chamado URL. A parte restante, que vai do caminho até o fragmento é chamado de URN. Nos itens da URI marcados como opcional, significa dizer que se o usuário omitir estes componentes, o *browser* ou o servidor adotará o padrão estabelecido, como por exemplo a porta 80 para protocolo HTTP.

Quando é dito que tanto caminho quanto recurso referem-se a locais e arquivos, é porque os servidores genéricos com configuração padrão seguem esta lógica. A alternativa chama-se **Reescrita de URL**, onde o servidor é configurado para trabalhar com Expressões Regulares para modificar a forma como o servidor Web interpreta uma URL recebida na requisição. Isso permite que o desenvolvedor padronize o formato de suas requisições, o que é justamente a mais importante regra de especificação para formato de requisições na Plataforma:

- Os itens da URN numa URI de uma requisição que esteja relacionada com um conteúdo dinâmico - ou seja, que chamam um serviço - não apontarão para um recurso em um local, mas sim, para este serviço, onde seus parâmetros serão passados e separados por barras de data "/" substituindo a *query_string*, mas não proibindo seu uso, inclusive na mesma requisição.

Utilizar o modelo de caminho para se referir aos parâmetros da requisição ao invés de uma *query_string* é conhecido como "URL limpa" e permite que uma requisição adote um padrão mais elegante e de fácil compreensão pelo usuário. Mesmo assim será permitido o uso de *query_string,* pois muitos aplicativos possuem rotinas que convertem dados em *query_string* antes de solicitar serviços da plataforma, inclusive formulários em sites são passados ao servidor desta forma quando utilizado o método HTTP/GET.

Ao utilizar o método de URL limpa para requisições, cada parâmetro passado com a barra de data é identificado numa ordem específica, sendo que certos parâmetros devem estar num dos níveis da URN, mostrados no **Quadro 4**.

Quadro 4 - Ordem de passagem de parâmetros via URL limpa

Ordem	Parâmetro
1º (ou ausente)	Modalidade de Serviço: /sessao, /arquivo/ db etc.
2º	Identificador do Microserviço: /login, /usuario, /notaFiscal etc.
3º	Uma Função de um Microserviço: /cadastrar, /emitir, /deletar etc.
4º em diante	Parâmetros da Função Solicitada: /parametro1/parametro2/parametro3...

Fonte: O autor (2016)

Duas especificações extras do *Framework* em relação a Parâmetros da quarta ordem em diante (parâmetros do microserviço ou função):

- É possível utilizar a ordem de aparecimento na URL para definir a ordem na qual eles serão inseridos no comando interno do servidor que irá executar o módulo ou procedure desejada;
- É possível ignorar a ordem destes parâmetros na URL, desde que obedecida a sintaxe extendida, que compreende o identificador do parâmetro e seu valor, separados por dois pontos, conforme o exemplo de URN a seguir:

- **/URL /sessão /info /usuário:eduardo /tipo:perfil**

3.2.1.3 Especificações do Gerente de Requisições para TCP e UDP

Assim como HTTP utiliza de regras e padrões para que todos os servidores que o utilizam entendam o que está sendo requirido, também será necessária a adoção de padrões de comunicação entre cliente e servidor para a comunicação usando protocolos TCP e UDP no nível do *software* da plataforma e do *software* cliente. Para isso será adotado o já existente Protocolo Simples de Acesso a Objetos (SOAP) que utiliza XML para demonstrar quais recursos remotos estão sendo solicitados. Este protocolo é padrão de mercado para sistemas distribuídos e possui uma curva de aprendizagem baixa. Sua vantagem é que ele também pode ser utilizado com HTTP. O caminho inverso também será aceito. Requisições no formato de uma URI também poderão ser enviadas como conteúdo de uma requisição TCP ou UDP. Outra implementação, com objetivo de simplificação de processos e economia de recursos de rede, é a adoção de JSON para criação do pacote de requisição via TCP/UDP, visto que sua sintaxe é mais simples e enxuta que XML.

3.2.1.4 Fluxo de Requisições na Plataforma

Em servidores Web padrões de mercado, como Apache, NGINX e JBoss, o fluxo de requisições segue um modelo simples e previsível, até se deparar com o recurso solicitado ou um código HTTP, dos quais pode ser um:

- Arquivo estático de um formato qualquer, como HTML, TXT, ZIP, PDF...;
- Arquivo executável ou interpretável, como EXE, DLL, PHP, WAR...;
- Código HTTP, como um redirecionamento 300, um erro 400 ou 500.

Para diminuir a possibilidade de erros de resposta e padronizar os recursos oferecidos pela plataforma, o *framework* propõe duas regras que diferenciam seu fluxo de requisições daqueles apresentados servidores genéricos:

- O formato de entrada e saída de dados sempre poderá ser especificado pelo cliente, na requisição, e se não o fizer, será considerado o formato padrão que estiver registrado na configuração do microserviço perante a plataforma;
- A plataforma sempre será responsável pela codificação e decodificação da entrada e saída de dados, tanto os encriptados quanto os que possuírem formato diferente do solicitado, gerados pela própria plataforma ou por algum microserviço, retornando um erro caso esta conversão não seja possível;

Por causa destas funções internas da plataforma, os microserviços ficam mais enxutos e podem ser desenvolvidos com maior foco em suas funções exclusivas.

A **listagem** a seguir apresenta as etapas de funcionamento do Gerente de Requisições ao tratar uma chamada genérica de um cliente, especificando seu **módulo responsável** por cada etapa do processo.

1. Recebe uma requisição do cliente ou encaminhamento de outra plataforma, através de HTTP, TCP, UDP ou outra implementação (Monitor de Entrada);
2. Decodifica a encriptação do conteúdo da requisição (Motor de Criptografia);
3. Identifica o tipo de serviço solicitado e decodifica os parâmetros para organizá-los em nome do microserviço, nome da ação, parâmetros de ação ou caminho e nome de arquivo ou *uploads* (Decodificador de Entrada);
4. Verifica se o serviço solicitado existe no servidor, se tem redundância e se precisa de uma sessão com usuário autorizado para ser executada (Config);
5. Verifica se o servidor possui recursos físicos para executar a requisição ou redireciona para outro servidor (Balanceador de Carga);
6. Verifica se há uma sessão ativa no servidor ou se foi informada na requisição uma sessão durável ou login de usuário com autorização para o serviço e cria um arquivo com estes dados (Gestor de Sessões);
7. Codifica os parâmetros de entrada no formato solicitado pelo cliente e os insere no arquivo de sessão para módulos externos (Biblioteca de Formatos);
8. Inicializa o relógio de temporização de requisição, em *thread* separada, para contar o *timeout* da execução do microserviço, evitando que a requisição fique num *loop* infinito, sem dar resposta ao cliente (Gestor de Execução);
9. Executa o serviço solicitado, em uma *thread* separada, inserindo os parâmetros do cliente, e monitorando seu tempo de execução e saída de informações, que pode ser via buffer, arquivos ou das duas formas juntas. (pode ser um módulo externo ou interno, dependendo do serviço solicitado);
10. Finaliza as *threads* abertas assim que os serviços solicitados finalizarem seus trabalhos (Gestor de Execução);
11. Coleta os resultados criados pela saída do último serviço executado e os converte para o tipo de saída solicitado pelo cliente (Biblioteca de Formatos);
12. Monta o cabeçalho e corpo de resposta para o cliente (Gestor de Saída);
13. Criptografa a saída, caso esteja neste modo de dados (Motor de Criptografia);
14. Envia a saída do serviço como resposta ao cliente, pelo canal de entrada;
15. Se a conexão for HTTP, fecha a conexão. Se for conexão TCP, inicia um relógio de *timeout* para este tipo de conexão e a fecha caso atinja o tempo limite sem resposta ou se receber comando fechá-la (Monitor de Entrada);
16. Registra as atividades sobre a requisição e execução no LOG (Função Log);
17. Limpa a memória de todos os objetos criados, arquivos abertos e sub-conexões (Balanceador de Carga).

No total são 17 passos executados no processo completo de execução de uma requisição dentro da plataforma, onde são envolvidas em média 12 funções dentro da plataforma, incluindo a execução do microserviço, que pode ser interno ou externo ao núcleo, ou à própria plataforma.

O processamento mais rápido de uma requisição se dá quando esta não utiliza todas as funções, como recursos de criptografia e balanceamento de carga, além de estar requisitando um serviço que não solicita uma sessão existente, passando para apenas 12 passos na execução do GRP.

No **Apêndice D** encontram-se um diagrama ilustrando visualmente um exemplo de fluxo de requisições de um usuário à Plataforma, e também um diagrama com as classes de um Gerente de Requisições (caso fosse implementado em Java), identificando as chamadas que cada classe faz às funções necessárias para executar qualquer tipo de requisição da Plataforma.

3.2.1.5 Os conceitos de Sequencias e Buffer de Requisições

Se um aplicativo é quebrado em serviços, os serviços são quebrados em microserviços, e estes acabam sendo quebrados em funções, frequentemente será notado que algumas funções são constantemente relacionadas a outras, como no exemplo abaixo:

1. Um usuário requisita um relatório de despesas do primeiro trimestre em PDF;
2. Uma função é executada para retornar este relatório numa stored procedure;
3. Outra função é executada converter este relatório, de JSON para PDF;
4. Mais uma função é executada, para enviar este arquivo para o usuário.

Nota-se que pelo menos três funções estão envolvidas para concluir a requisição do usuário. Se todas as funções citadas estivessem inseridas num mesmo microserviço, eles seriam inúteis para servir a requisições diferentes, pois não poderiam ser utilizados individualmente, e se houvesse esta opção, chamando as funções de dentro do microserviço, este consumiria mais memória irresponsavelmente, pois não executaria nem metade de sua funções alocadas.

Manter as funções separadas fisicamente, além de ser econômico, permite que as mesmas sejam utilizadas de formas variadas e oportunas. Por tanto, pela necessidade de conectar serviços para gerar resultados complexos sem a necessidade de criação de dependências entre estes serviços, serão criadas duas especificações adicionais:

- **Sequências**: é um modo de operação solicitado pelo cliente, na forma de requisição à plataforma, e que contém uma lista de serviços a serem executados em sequência, mas que é acatado pelo Gerente de Requisições como sendo uma requisição única. A diferença entre este modo de operação e o modo padrão do Gerenciador é que ao final da execução de cada serviço indicado na sequência, o Gerenciador recolher seu resultado e o aplica como parâmetro de entrada no próximo serviço, que é executado na mesma thread do serviço anterior. Este modo se inicia na Etapa 8 do fluxo de requisição e adia a Etapa 10 até que o último serviço da fila tenha sido executado. Uma sequência deve ser registrada na configuração da Plataforma, com um identificador único e sua lista de serviços em modo URN;
- ***Buffer* de Requisições**: é uma pasta temporária criada especificamente para as requisições em sequência, com arquivos que contem os resultados do processamento de cada serviço dentro de uma sequência, que podem ser usados como parâmetro de entrada para outros serviços ou serem baixados pelo usuário cliente, enviados como anexos e-mail ou qualquer outra finalidade que caiba em uma função. Esta pasta deverá ser excluída por padrão ao final do fluxo de requisição, na Etapa 17, porém o modo Sequência pode adiar esta exclusão, desde que o cliente tenha solicitado na sequência, com o **parâmetro de adiamento**, que pode contar uma opção cronológica, como minutos, horas e dias, ou ainda que o conteúdo seja apagado somente no fim da sessão de usuário relacionada a este buffer ou após o conteúdo ser acessado N vezes pelo cliente (como download).

3.2.1.6 Catálogo de serviços disponíveis na Plataforma

Os web-services que estão em acordo com as especificações da W3C (que é o órgão regulador da Web) possuem um serviço chamado WSDL (*Web Services Description Language*), que é responsável por informar ao cliente quais são os serviços disponíveis neste web-service, quais são os parâmetros aceitos por eles, e quais os formatos possíveis de entrada e saída.

Este tipo de serviço, quando invocado, gera um arquivo XML ou JSON contendo as informações de cada serviço disponível na plataforma criada com base no *Framework*. É análogo a digitar um comando no console do sistema operacional usando o parâmetro ***/?*** para obter as instruções de uso do comando.

De maneira geral ele é acessado através de um comando */wsdl* sozinho na URL do web-service. Mas no caso desta plataforma, as especificações do *framework* aceitam que se use um parâmetro ***/**** para acessar este recurso, que deve ser o

último parâmetro, e pode ser utilizado em conjunto com outros parâmetros da plataforma para se obter um tipo diferente de informação em cada caso, como os serviços disponíveis por categoria, todas as especificações, parâmetros e saídas de um microserviço específico ou até mesmo de um único comando deste.

Também é possível utilizar o parâmetro **/**** para acessar o **manifesto** de um microserviço, que contém: a sua descrição, histórico de atualizações, contatos dos responsáveis pelo serviço, além de links para imagens referente ao serviço, como logos, capturas de telas de exemplo e outros recursos reutilizáveis.

3.2.2 **Banco de Dados**

Uma das funções mais importantes de um *framework* é sua capacidade de manter a padronização da forma e conteúdo dos objetos que são desenvolvidos com sua orientação e ferramentas (O'BRIEN, 2011). No caso do *Framework* para plataformas, está função é bem definida nas suas configurações.

Optou-se por utilizar bancos de dados relacionais para fazer a gestão das configurações e operações da plataforma, visto que uma das categorias de serviço mais importante deste é justamente a utilização de funções de bancos de dados sem desenvolvimento de APIs no servidor (*serverless functions*). Mesmo assim, os interessados em projetar sua plataforma usando tecnologias NoSQL poderá fazê-lo, desde que utilize as mesmas especificações estruturais para sincronizar com as configurações de outras plataformas do *Framework*.

3.2.2.1 Convenções e Configurações

Convention Over Configuration (CoC) é um modelo de desenvolvimento de *software* que busca diminuir o número de decisões que os desenvolvedores precisam tomar e visa ganhar simplicidade sem perder flexibilidade. Quando a convenção implementada pela ferramenta que utilizada corresponde ao comportamento desejado, o desenvolvedor faz menos esforço (ou não há esforço) no ritual de configuração. Somente se o comportamento desejado for distinto da convenção implementada é que se torna necessário elaborar configurações. Esta visão permite ao programador trabalhar num nível maior de abstração sem a necessidade de uma camada de configuração própria (MASSOL; ZYL, 2007).

Neste *Framework* procura-se equilibrar a necessidade de customização e configuração com a possibilidade de padronizar aquilo que não precisa de atenção da equipe de desenvolvimento para que esta materialize seus projetos e crie serviços eficientes para a Plataforma.

Neste sentido, o banco de dados do *Framework* foi pensado em sua totalidade, para que os desenvolvedores de plataformas não precisem tirar o foco do desenvolvimento de microserviços para sanar inconsistências ou deficiências, seja no banco ou nas configurações.

Alguns novos campos de configurações podem por ventura ser desenvolvidos, não como intuito de complementar o padrão, mas para dar suporte a serviços novos que não tenham sido previstos durante o projeto do *Framework*.

O **Quadro 5** separa os itens que podem ser customizados dentro de cada plataforma e os itens que devem ser padronizados de acordo com o *Framework*, para convenção geral em todas as plataformas orientadas por ele.

Quadro 5 - Convenções e Configurações do Framework

Convenções Padronizadas	**Configurações Customizáveis**
Ordem de passagem de parâmetros	Alguns tipos de serviços e funções podem ser desabilitadas ou não implementadas
Nomes próprios de categorias de serviços e suas ações	Bibliotecas de acesso à plataforma e protocolos de comunicação disponíveis
Nomes de tabelas de bancos de dados e colunas preexistentes	Especificações comerciais completas, mas que podem ser substituídas por especificação própria
Controle de usuários, perfis, procedures e sessões no servidor ou no banco	Independente de bancos de dados, linguagens de programação e sistemas operacionais escolhidos
Modelo de saída de procedures e microserviços em caso de erros	Limite de conexões aceitas para o mesmo IP ou mesma rede deve ser definida pelo gestor da plataforma
Listagens ENUM terão campos com rótulo no singular e no plural	Os serviços podem interagir entre si, com chamadas diretas estilo RPC ou através de Sequências

Fonte: O autor (2016)

3.2.2.2 Especificações do Banco de Dados

A modelagem relacional e corpo das tabelas estão no **Apêndice A**, e os padrões de modelagem e convenções estão no **Apêndice B**.

As especificações do banco de dados são poucas, porém abrangentes, para evitar inconsistência entre as plataformas desenvolvidas com base no *Framework*:

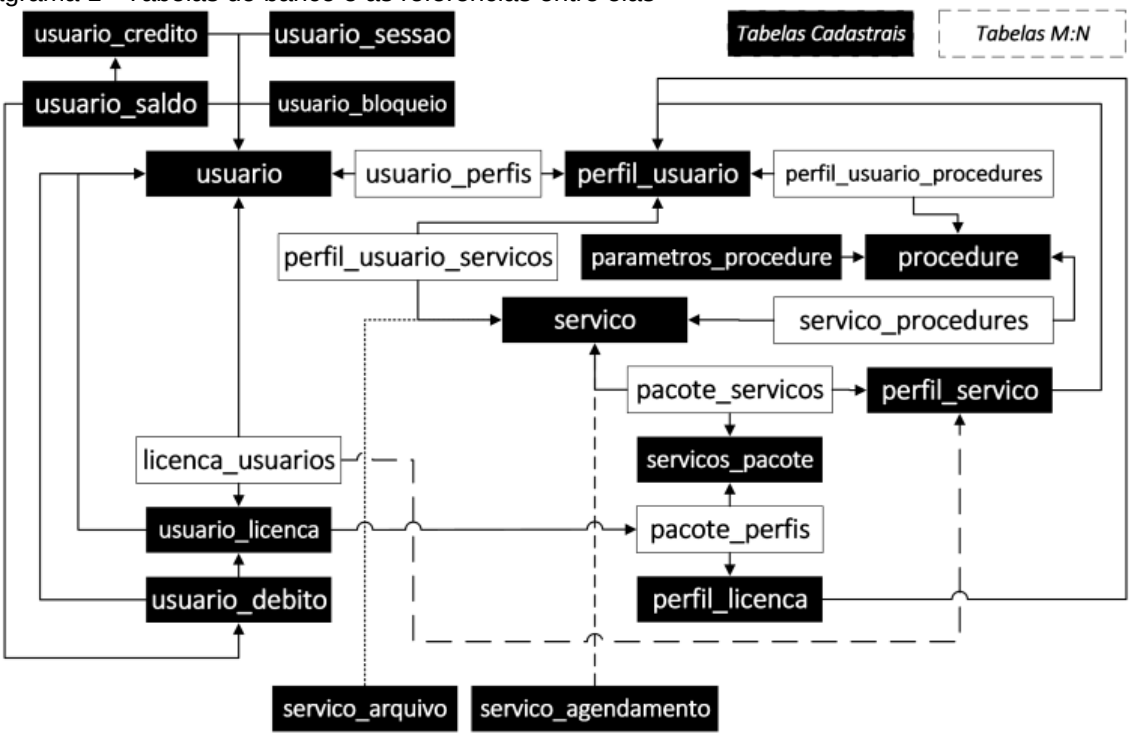
- O banco precisa ter tabelas no padrão do *Framework*, para controlar o **cadastro de usuários** da plataforma, seus **perfis de acesso** e suas **sessões duráveis** (utilizadas em aplicativos móveis que recebem informações do serviço em forma de *Notificações PUSH*);
- O banco precisa ter tabelas no padrão do *Framework*, para controlar os **microserviços disponíveis** e suas interações com os usuários e **funções disponíveis** dentro (procedures) e fora do banco (módulos);
- O banco pode ter tabelas no padrão do *Framework*, para controlar o **acessos** aos microserviços e funções **que necessitam de licença paga**;
- O banco só pode dar **permissão de execução de procedures**, com o comando SQL CALL. Dependendo do SGBD, as stored procedures são funções, que só podem ser acessadas por comandos SQL SELECT. Independente disso, o usuário do SGBD usado na Plataforma e em todos os microserviços, com o banco de dados padrão, **não pode ter acesso** a comandos **DELETE**, **UPDATE** e **INSERT**;
- Se um serviço precisar de acesso que viole as regras anteriores para acessar informação, esta deverá ser disponibilizada em um banco de dados separado do banco da plataforma.
- Os serviços podem acessar o as procedures via Plataforma com requisições HTTP e TCP, ou diretamente se puderem se conectar com o banco, mas obedecendo todas as regras anteriores;

3.2.2.3 Tabelas de banco de dados preexistentes no *Framework*

As tabelas do banco de dados padrão do *Framework* estão divididas em quatro categorias, para facilitar sua identificação, seu propósito e quais tabelas se relacionam:

- Tabelas Primárias: Usuários, Sessões e Procedures;
- Tabelas Secundárias: Configuração e Gestão de Serviços;
- Tabelas Terciárias: Controle de Licenças e Créditos;
- Tabelas de Aprimoramento da Plataforma.

Diagrama 1 - Tabelas do banco e as referências entre elas



Fonte: O autor (2017)

Em termos de design, estas tabelas são bastante dependentes umas com as outras, como mostrado no **Diagrama 1**, e por isso acabam contrariando as práticas de desacoplamento no desenvolvimento de sistemas. Mas a justificativa neste caso é que os serviços de sessões da plataforma utilizam as informações de usuário, perfis, serviços e licenças para manter a segurança nos acessos aos microserviços e às informações disponíveis nos bancos de dados.

As tabelas e suas referências foram desenhadas de forma a mimetizar o máximo possível o funcionamento de classes referenciadas em uma linguagem orientada a objetos, permitindo que sua estrutura também seja implementada em NoSQL ou bancos de dados orientados a objetos.

3.2.2.3.1 Tabelas de Usuários, Sessões e Procedures

A Plataforma pode funcionar com apenas 8 tabelas (listadas no **Quadro 6**) que possuem as informações mínimas permitidas pelo *Framework*, para que uma plataforma ofereça serviços de sessões e funções *serverless*.

Quadro 6 - Banco de dados básico do Framework

Tabela	Conteúdo
usuário	Usuários que acessam a plataforma para administrá-la ou usar serviço
usuário_bloqueio	Bloqueios e desbloqueios de usuários, por motivos diversos
usuário_sessão	Sessões ativas de usuários conectados a vários dispositivos
usuário_perfis	Intermediário **m:n** das tabelas *usuário* e *perfil_usuario* para permitir vários perfis diferentes para cada usuário
perfil_usuário	Perfis de acesso de usuários que definem o papel de cada um na plataforma, como administradores, licenciados etc.
perfil_usuário_procedures	Intermediário **m:n** das tabelas *perfil_usuario* e *procedure* para permitir acesso a várias procedures através dos perfis de cada usuário
procedure	Procedures do banco, que são acessíveis através da plataforma, como se fossem funções/ações de microserviços
parâmetros_procedure	Especificações dos parâmetros de cada procedure para facilitar a execução de uma procedure, principalmente em formulários html

Fonte: O autor (2016)

3.2.2.3.2 Tabelas de Configuração e Gestão de Serviços

Para um controle mais adequado de múltiplos serviços na Plataforma, é necessário implementar as tabelas listadas no **Quadro 7**.

Quadro 7 - Banco de dados para gestão e operação de serviços

Tabela	Conteúdo
serviço	Serviços disponíveis como: módulos, repositórios, comandos, etc.
serviço_procedures	Intermediário **m:n** das tabelas *serviço* e *procedure* para permitir acesso às procedures por determinados serviços na plataforma
perfil_serviço	Perfis de acesso disponíveis exclusivamente para um serviço, mas que não se encaixa nos demais perfis de acesso da plataforma
perfil_usuário_serviços	Intermediário **m:n** das tabelas *usuário_perfil* e *serviço* para permitir acesso a vários serviços dependendo do perfil de usuário
sequência	Programações para executar serviços de forma sequencial.
sequência_buffer	Configuração de tipo de exclusão de pasta temporária de buffer.

Fonte: O autor (2016)

3.2.2.3.3 Tabelas de Controle de Licenças e Créditos de Usuários

Muitos serviços e aplicativos na Web são gratuitos, enquanto outros precisam que seus usuários comprem uma licença de uso ou paguem uma mensalidade para continuarem a ter direito de uso. O *Framework* leva isto em consideração e já tem preparadas as especificações (listadas abaixo) para os desenvolvedores que prefiram usar uma solução pronta e compatível com a Plataforma:

- Serviços que impõe licenças precisam informações extras sobre usuários que possuem estas licenças, como data de aquisição, validade, modalidade...;
- Cada licenciado pode ter um número de usuários atrelados às suas licenças para operar os serviços em seu nome, com permissões diferenciadas e definidas pelo dono da licença;
- O número de usuários autorizados a utilizar uma licença depende do tipo de licença adquirida, assim como das permissões do serviço licenciado;
- O esquema de licenças impõe a implementação de tabelas do *Framework* (listadas no **Quadro 8**) que são específicas para gerenciar as complexas funcionalidades necessárias para gerenciar licenças.

Quadro 8 - Banco de dados para gestão de licenças

Tabela	Conteúdo
usuário_crédito	Registros de entrada de valores para o usuário adquirir licenças e pacotes
usuário_débito	Registros de saída de valores do usuário, por aquisição ou reembolso
usuário_saldo	O valor financeiro atual disponível para cada usuário adquirir licenças
usuário_licença	Licenças que cada usuário possui para acesso como assinante dos serviços
licença_usuários	Intermediário **m:n** das tabelas *usuário* e *usuário_licença* para permitir acesso de subusuários à uma licença alheia para operar seus serviços
pacote_serviços	Intermediário **m:n** das tabelas *serviço* e *serviços_pacote* para permitir preços diferentes no mesmo pacote, por cada perfil de acesso (nível) dos serviços
serviços_pacote	Pacotes com vários serviços que podem ser adquiridos através de licenças
pacote_perfis	Intermediário **m:n** das tabelas *serviços_pacote* e *perfil_licença* para permitir perfis diferentes para compra de pacotes escolhendo o perfil mais adequado
perfil_licença	Perfis de licenças disponíveis para usuários assinarem, em ciclos temporais

Fonte: O autor (2016)

3.2.2.3.4 Tabelas de Aprimoramento da Plataforma

Algumas tabelas podem ser implementadas para configurar e operar serviços adicionais à plataforma, com o objetivo de permitir que o desenvolvedor saia da convenção padronizada e possa customizar sua plataforma para funcionar como outros servidores disponíveis no mercado.

O **Quadro 9** apresenta algumas propostas de tabelas para serviços específicos, especialmente no caso da inclusão de novas categorias de serviços não padrões ainda do *Framework*, como o serviço de DNS dinâmico (DDNS) por exemplo.

Quadro 9 - Banco de dados para serviços extras

Tabela	Conteúdo
serviço_arquivo	Tipos de arquivos (extensões) que quando referenciados na URL da requisição, são passadas como parâmetro na execução do serviço correspondente, que foi referenciado neste registro. Ex.: arquivo *teste.php* executa o módulo *php.exe*
serviço_agendamento	Programações de execução agendada de serviços disponíveis na plataforma, feitos internamente por administradores ou por outros serviços
serviço_ddns	Possui endereços IP dinâmicos e identificadores únicos de clientes que por ventura precisem se comunicar diretamente com outros clientes através de uma rede remota roteada por serviços NAT
serviço_urlsimples	Possui pseudônimos curtos, autogerados ou não, para endereços extensos de páginas online, com o objetivo de facilitar sua digitação e pode ser aplicado ao um microserviço ou função de de redirecionamento de URL

Fonte: O autor (2016)

3.2.2.3.5 Tabelas para Enumeradores (Listas)

Um enumerador é um tipo de coluna (ENUM) em um banco de dados, cuja informação só pode ser uma dentre uma lista de itens específicos, onde a informação se mostra para o usuário como uma string (conjunto de caracteres), mas o que é gravado na coluna do banco de dados é um número que indica a posição do item selecionado na lista de itens deste enumerador.

Em alguns SGBDs um enumerador é componente exclusivos de uma tabela, criado no momento da criação da sua tabela. Mas em outros SGBDs, um

enumerador é um objeto separado da tabela, e torna-se um tipo de dado, assim como o *integer*, o *varchar* e outros tipos de dados padrões de bancos. Deste modo o enumerador pode ser apontado como o tipo escolhido para colunas em várias tabelas no esquema inteiro.

Para este trabalho, foi decidido utilizar tabelas para substituir a funcionalidade dos ENUMs, para manter a compatibilidade à maioria dos SGBDs. O **Quadro 10** mostra quais tabelas do tipo ENUM serão necessários para o *Framework*, pois serão utilizados para complementar as informações de algumas das das tabelas já mencionadas neste trabalho.

Quadro 10 - Listagens tipo ENUM para o banco de dados

Tabela ENUM	Tabela Usuária	Conteúdo
lista_bloqueio_tipo	usuários_bloqueio	Define o motivo pelo qual um bloqueio de usuário foi executado. Ex: confirmação de email, débito, judicial...
lista_bloqueio_fonte	usuários_bloqueio	Define num registro de bloqueio, qual o responsável pela solicitação de bloqueio. Ex: serviço, admin, judicial etc.
lista_perfil_tipo	perfil_usuário	Define o tipo de usuário numa lista de classes padrão de usuários da plataforma. Ex: suporte, admin, usuário etc.
lista_serviço_tipo	serviço	Define qual a categoria de serviço, num registro de um serviço. Ex: banco de dados, sessão, painel, etc.
lista_procedure_tipo	procedure	Define a função principal dentro da procedure. Opções: criar, listar, atualizar, apagar, verificar
lista_procedure_saída	procedure	Define o formato de saída de dados de uma procedure. Ex: tabela, registro, célula, variada, número, data etc.
lista_credito_status	usuário_crédito	Define a situação de um registro de crédito. Ex: em aberto, compensado, recusado, cancelado, reembolsado...

Fonte: O autor (2016)

- Em tabelas ENUM sempre haverá duas colunas para rótulos. Dependendo do uso de cada tabela, as colunas terão rótulos no singular e no plural (ex.: substantivos), ou no masculino e feminino (ex.: adjetivos como profissões).

  Observação: *Esta separação de tabelas não é física, nem obrigatória.*

3.3 O LADO DO CLIENTE

O desenvolvedor que optar por esta Plataforma, terá uma tarefa inicial de configurá-la, apontando seu banco de dados, inserindo as tabelas obrigatórias do *Framework*, e informando quais procedures estarão disponíveis para os usuários. Nenhuma linha de código será necessária para desenvolver uma API. Isso significa que o próximo passo é definir quem serão os clientes a utilizar os serviços da Plataforma. Até este ponto, os objetivos de separação de camadas de banco de dados e de interface com o usuário, por equipe de desenvolvimento, além da supressão da necessidade de desenvolvimento da camada de controle, já foram alcançados.

O lado do cliente se define como uma camada onde uma parte do processamento do serviço é executada, com foco na interface e experiência positivas do usuário, o qual pode fazer solicitações, inserir novos dados e receber informações e o resultado do processamento de destas solicitações.

Durante muitas décadas, a camada do cliente - também conhecida como camada visual - era toda montada no mesmo local onde se fazia o processamento e controle das informações do serviço: o servidor. Mas com a modernização dos navegadores e web e modernização das linguagens de script (JavaScript, TypeScript) e de layout (HTML5, CSS3) presentes nestes navegadores, veio a possibilidade de trazer uma boa parte deste processamento para o lado cliente, que agora podia acionar o servidor somente quando precisasse de novas informações oriundas de bancos de dados ou arquivos hospedados remotamente.

Com o advento da tecnologia AJAX (*Asynchronous JavaScript e XML*), não era mais necessário recarregar uma página inteira após cada consulta ao servidor. Ao invés disso, o script poderia fazer uma nova requisição ao servidor, que devolve a informação ao script para este decida como processá-la e mostrá-la para o usuário (SILVA, 2009).

Mais recentemente, a popularização dos dispositivos móveis, conectados continuamente à Internet, trouxe ainda mais importância à arquitetura de divisão de tarefas entre o cliente e servidor, sendo este último o responsável por manter o cliente atualizado de informações e por fazer o processamento mais pesado, deixado para o cliente a tarefa de guardar somente as informações mais essenciais e cuidar para que o usuário tenha a melhor experiência visual e utilize os serviços conectados de forma transparente, sem a necessidade de algum conhecimento técnico para isso.

Para o usuário de smartphones, não há como saber se uma informação em sua tela já estava em seu aparelho, ou se acabou de ser baixada via Internet, se num servidor central ou se foi de outro usuário que também está conectado. Todos os serviços parecem seguir as mesmas regras e funcionar da mesma forma. É o que se deseja que os usuários das plataformas deste *Framework* sintam, porém o que mais interessa é que os desenvolvedores também possam sentir-se da mesma forma ao projetarem seus serviços, microserviços e funções. O foco é o que seu código fará, pois toda a infraestrutura que ele usará para atingir seus usuários já estará disponível. Isto se chama Transparência de Interface.

3.3.1 **Especificações de Compatibilidade**

A lista de dispositivos, aplicações e navegadores (que deverão ser aptos a acessar a Plataforma) é extensa. Soma-se à necessidade de que as linguagens de programação mais populares possam ser usadas, tanto para desenvolver a Plataforma (servidor) quanto para desenvolver os aplicativos clientes. Por isso é necessário um equilíbrio entre funcionalidade e compatibilidade, para que um cliente possa manter uma comunicação completa e estável com seu servidor. Seguem as diretrizes para implementação de uma camada cliente compatível com o *Framework* da Plataforma.

- Deve ser capaz de trabalhar com **pelo menos um dos protocolos** de comunicação compatíveis com a Plataforma: HTTP, HTTPS, TCP e/ou UDP;
- Em comunicações HTTP e HTTPS, precisa ser capaz de trabalhar com ambos os métodos de transmissão **GET e POST**;
- Dependendo dos serviços utilizados, deve ser capaz de **encapsular os dados requisições** feitas à plataforma e mantê-los guardados em forma de objetos ou variável de estruturas de dados, para serem reaproveitados caso a resposta da requisição assim permita;
- Se for um cliente Web, deve ser capaz de **fazer requisições JSONP**, que são requisições seguras à servidores em domínios diferentes do domínio que carregou a página atualmente mostrada no navegador;
- Se utilizar protocolo **UDP**, é recomendável criar um mecanismo de controle do roteador para permitir recebimento de respostas remotas através de UDP, o que é geralmente conseguido através de uma **configuração do mecanismo NAT** em aparelhos de controlam a conexão doméstica com a Internet.

De acordo com Flanagan (2013) JSONP (*JSON with padding)* é um complemento ao formato de dados JSON. Ele provê um método para enviar

requisições de dados de um servidor para um domínio diferente, uma coisa proibida pelos navegadores convencionais por causa da *Política de mesma origem*, onde uma página carregada de X.com, não pode normalmente se comunicar com servidores diferentes, como Y.com ou Z.com.

Apesar de JSONP ser uma prática restrita na maioria dos servidores (CORS desabilitado ou incompatível), ela é bastante útil quando se identifica que o servidor original está fora do ar, mas o cliente sabe os endereços de servidores espelhados, fazendo com que a aplicação continue funcionando online mesmo que um servidor pare de funcionar. Para manter a segurança nestes casos, as **requisições JSONP precisam** obedecer as seguintes diretrizes:

- Incluir **código de registro do serviço** que gerou a página no servidor;
- Incluir o **código da sessão ativa do usuário**, que esteja registrada na tabela de sessões duráveis, no banco de dados;
- Incluir o nome de uma **função de *callback*** que deverá estar presente no código do cliente antes da requisição, para ser executada no fim e tendo como parâmetro os dados enviados como resposta pelo servidor.

  Observação: *só é possível fazer requisições JSONP com o **método GET**.*

3.3.2 **Bibliotecas de Auxilio ao Desenvolvimento**

Um desenvolvedor deve ter liberdade para criar, da forma que lhe for mais conveniente, as aplicações clientes que utilizarão os serviços da plataforma. Conhecer as especificações do *Framework* deve ser o suficiente para habilitar um desenvolvedor a cadastrar os serviços desenvolvidos e efetuar consultas à plataforma. Porém é interessante que exista uma forma padronizada do cliente se comunicar com a plataforma. Algo que se torne uma especificação e sirva de parâmetros para o desenvolvimento de novas ferramentas e novos componentes da Plataforma, ou até mesmo do *Framework*.

Uma forma de criar esta especificação é criando bibliotecas de funções para as linguagens, como uma opção mais prática para que desenvolvedores controlem a comunicação de suas camadas visuais com a plataforma de forma transparente, sem se preocupar com os requisitos técnicos, focando seu trabalho na experiência de interatividade do usuário.

3.3.2.1 Javascript para clientes Web ou Mobile

A ampla maioria dos sites modernos usa JavaScript e todos os navegadores - em computadores de mesa, consoles de jogos, tablets e smartphones - incluem interpretadores de JavaScript, tornando-a a linguagem de programação mais onipresente da história. JavaScript faz parte da tríade de tecnologia que todos os desenvolvedores devem conhecer, junto com HTML e CSS (FLANAGAN, 2013).

É com JavaScript que funcionam as requisições do tipo AJAX, seja para o servidor matriz da Plataforma, seja para servidores espelhados, usando o método JSONP. Por isso uma biblioteca para esta linguagem torna-se muito importante, pois poderá ser utilizada pela grande maioria dos desenvolvedores que utilizam o *Framework*.

Caso um desenvolvedor opte por não utilizar a biblioteca padrão do *Framework*, requisições à Plataforma podem ser feitas utilizando links diretamente atrelados às tags **<A>** de HTML, mas isto faz com que o resultado seja carregado em uma outra página. Isto faz sentido se a requisição conter uma chamada a um serviço de **Modelos** ou a uma **Sequência** que possua um Template ou download/visualização de arquivo, como último serviço da lista. Mas o ideal, para que se tenha uma experiência como aplicação, é utilizar o JavaScript para fazer estas requisições, seja em suas funções nativas como **XMLHTTPRequest()** ou com ajuda de bibliotecas como **jQuery** (SILVA, 2009) usando *$.ajax()* ou *$.getJSONP()*.

Desenvolvedores que estejam acostumados a utilizar *frameworks* de de estruturação e padronização de camadas para JavaScript, como *AngularJS* (SESHADRI; GREEN, 2014), *BackboneJS*, *EmberJS* ou *ReactJS*, podem continuar utilizando estas tecnologias, criando em seu código, um roteador que possa fazer requisições à Plataforma, de forma padronizada e transparente para a equipe, que não sentirá nenhuma diferença, caso já tenha trabalhado com APIs para trocar dados com sua interface de usuário.

Uma biblioteca de auxílio à plataforma é algo bem simples se comparada a um *framework* de estruturação. Os itens necessários são descritos abaixo, como sua especificação:

- **Objeto de Configuração e Controle de Informação**: trata-se de um objeto JavaScript padrão, nomeado como **$platformScope**, cujas propriedades são informações sobre a plataforma, como URL de acesso, dados do usuário a ser conectado, de sua sessão, e dos serviços remotos a serem utilizados pela aplicação cliente. Parte de sua estrutura é descrita no **Quadro 11** com o código básico do seu objeto mostrado no **Apêndice E**.

Quadro 11 - Estrutura principal do objeto $platformScope

Propriedade	Subpropriedades	Descrição
target	platformURL, wsdl, backupURL[], pID	Indica o caminho para acessar a plataforma, que pode ser uma URL ou IP, e caminhos alternativos
user	id, $key, info, access	Dados do usuário logado na plataforma com este cliente
security	$encryptMode, $key	Modo de segurança ativado para esta sessão de uso
application	ID, type, profiles	Identifica aplicação ou serviço relacionado a este cliente
requests[]	URN, $objectSent, $objectReceived	Vetor de histórico de requisições feitas à plataforma com este cliente, na memória volátil, com objetos trocados
session	$id, $token, type, lastCheck, expiration	Dados da sessão de usuário registrada na plataforma

Fonte: O autor (2017)

Em *requests*[] serão guardadas informações sobre cada acesso à plataforma, criando um histórico com dados recebidos como objetos, que poderão ser reutilizados pelo cliente sem necessidade de novo acesso, a não ser para atualizar tais dados. Os métodos deste objeto são as funções para acessar a plataforma, tratar suas respostas e manter a segurança e privacidade do usuário durante sua utilização. Ao chamar a função *$platformScope.initialize*() o objeto guarda o endereço da Plataforma e imediatamente consulta o serviço de listagem de diretório para descobrir quais serviços estão disponíveis para o cliente.

- **Funções de Acesso à Plataforma**: são métodos utilizados para enviar requisições remotas, do cliente para o servidor da Plataforma. É possível que qualquer requisição possa ser feita através de um método genérico, com todas as opções que a plataforma possa oferecer. Porém uma função assim acaba tendo muitos parâmetros que poderiam ser ignorados dependendo do que estiver sendo requisitado ao servidor. Por isso, admite-se a função geral, chamada de **request()** (parte de sua estrutura é detalhada no **Apêndice E**), juntamente com outras funções mais específicas para requisições como envio de formulários (listadas no **Quadro 12**), controle de acesso e sessões de usuários, envio e recebimento de arquivos, verificações cíclicas, entre outras. Todas estas funções devem utilizar a função principal *request*() no seu código interno, para evitar código repetido.

Quadro 12 - Funções JavaScript para acesso à Plataforma

Nome	Função
request()	Faz as requisições à Plataforma através de AJAX, ou usa JSONP se a URL padrão for diferente da URL alvo. Todas as outras funções abaixo a utilizam
getWSDL()	Recebe a estrutura de serviços e funções disponíveis na plataforma, para evitar que o cliente faça uma requisição a um serviço que não existe
getSession()	Inicializa ou recebe informações da sessão aberta na Plataforma para o cliente
getAccess()	Inicializa o Login ou recebe informações sobre o Usuário logado na Plataforma
getObject()	Recebe um objeto JSON ou XML de um serviço, para tratá-lo no cliente
putObject()	Devolve um objeto JSON ou XML para um serviço, para que este atualize no banco de dados ou na configuração da Plataforma, os dados alterados pelo cliente
getForm()	Recebe os valores dos campos de um formulário específico, através de função
putForm()	Envia os valores dos campos de um formulário no cliente para um serviço remoto
response	Guarda as informações recebidas da última requisição feita
fail	Guarda as informações da última falha de requisição, ou nulo se houve sucesso

Fonte: O autor (2017)

- **Funções de Criptografia**: são métodos de assistência às funções de acesso ao usuário e sessões na plataforma, cujas requisições são criptografadas com as opções disponíveis de SSL e RC4 que não são gerenciadas pelo *browser* ou smartphone, quando se utiliza HTTP. Estas estão listadas no **Quadro 13**.

Quadro 13 - Funções JavaScript para criptografia de dados

Nome	Função
getPublicKey()	solicita uma chave pública à Plataforma, exclusiva para a sessão
decodeWithPublicKey()	decodifica conteúdo recebido pela plataforma com SSL simples
encodeWithPublicKey()	codifica conteúdo com SSL para envio à Plataforma
generateDoubleKey()	cria a chave RC4 e envia para a Plataforma usar na sessão
doubleEncode()	codifica conteúdo em RC4 e SLL com as chaves cliente/plataforma
doubleDecode()	decodifica conteúdo recebido e codificado com as duas chaves
refreshKeys()	atualiza as chaves do cliente e da plataforma para redobrar a segurança
encodeRequest()	criptografa todos os parâmetros da requisição para aumentar segurança

Fonte: O autor (2017)

- **Funções Utilitárias**: são métodos que não estão relacionados com o acesso à Plataforma, nem com a segurança e encriptação, mas são funções gerais que podem facilitar o dia-a-dia do desenvolvimento relacionado a ela, como mensagens de erro, formatação de parâmetros, conversão de dados e validação de formulários.

3.3.2.2 Outras linguagens para clientes no Modo Desktop

Muitas das grandes empresas e instituições públicas ainda podem estar utilizando grandes sistemas monolíticos feitos há décadas atrás. Sistemas podem ter uma grande influência nas culturas das organizações e no rendimento de seus funcionários. Neste cenário, mudanças radicais não são bem vindas, e por isso qualquer restruturação tecnológica geralmente segue um cronograma bem extenso, que permita a adaptação de todos os envolvidos (LOBO, 2008).

Segundo a IBM (2010), grande parte dos sistemas monolitos pelo mundo ainda se encontram na forma de aplicações desktop que acessam mainframes para sincronizar seus dados. Uma forma de desconstruir esse paradigma é permitir que as aplicações desktop passem a acessar serviços e microserviços de forma transparente e distribuída (RICHARDSON, 2016). Utilizar a Plataforma baseada no Framework proposto é o primeiro passo para atingir este objetivo, pois permite que aplicações desktop façam requisições sob demanda, de dados e serviços remotos, da mesma forma que uma aplicação Web faria junto à Plataforma. Assim a transição dentre o desktop e a Web começaria de dentro para fora.

A compatibilidade de várias linguagens de programação e sua capacidade de acessar a plataforma não é útil somente para aplicativos desktop, mas também para a própria Plataforma (que pode ser implementada em diversas linguagens) a fazer consultas a outras plataformas do mesmo *Framework*, mesmo que feitas em linguagens diferentes, e tanto em HTTP como TCP ou UDP.

A sua utilização é facultada ao desenvolvedor interessado em implementar uma versão da Plataforma em uma das linguagens mais recomendadas (JavaScript, Java, PHP, Pascal, Python, Ruby entre outras), cujas vantagens e desvantagens são citadas no **Apêndice F**.

3.4 SEGURANÇA

De acordo com Forristal (2002), a questão da segurança é uma das mais importantes ao se projetar uma aplicação na Internet. Se alguma brecha for deixada aberta, por erro ou omissão, todo o conteúdo do proprietário e de seus clientes estará potencialmente comprometido, podendo ser copiado, apagado, violado, vendido ou publicado sem autorização, causando muito prejuízo a seus donos.

Em um servidor Web comum, dois protocolos podem ser utilizados para trocar informações: o HTTP, onde as informações não são criptografadas e podem ser interceptadas por terceiros, na saída, chegada, ou até no meio do caminho. Mas também existe o HTTPS (S de secure), que possui uma camada adicional que codifica os dados transmitidos com criptografia SSL/TLS, que além de permitir a privacidade da comunicação entre servidor, também possibilita que se verifique a autenticidade do servidor e do cliente por meio de certificados digitais.

3.4.1 **Especificações de Segurança para Comunicação**

As diretrizes de segurança entre a Plataforma, clientes e serviços permitem que o administrador escolha entre as seguintes opções de segurança:

- Sem encriptação: as informações transferidas não sofrem nenhum mudança direta pelos pontos de transmissão e recebimento, podendo facilmente ser interceptadas por pessoas com ferramentas e conhecimento suficiente, se a rede utilizada para a comunicação não for segura. Em contrapartida, a comunicação ganha em performance e compatibilidade com dispositivos e aplicativos que não conseguem implementar criptografia;
- Encriptação SSL com camada simples: o servidor enviar uma chave pública exclusiva para o cliente, que a usa para encriptar suas requisições e para decriptar as respostas da Plataforma. A criptografia neste caso não será decodificada pelo *browser*, mas pelo aplicativo cliente ou código Javascript. Também não é necessário certificado de autenticidade;
- Encriptação de dupla camada, com SSL e RC4: este modelo de segurança foi pensado ao mesmo tempo que as idéias principais deste *Framework*, e implementa dois métodos de encriptação juntos, onde o servidor envia uma chave pública para que o cliente encripte e devolva outra chave intermediária que será usada para enviar mensagens encriptadas duas vezes, a primeira camada em SSL e a segunda em RC4. Diferentemente das chaves públicas e privadas do SSL, cada conexão recebe uma chave RC4 própria.

3.4.2 **Especificações de Segurança para Configuração**

As diretrizes de segurança da Plataforma para gestão da configuração são muito importantes, pois espera-se que muitos usuários obtenham direitos administrativos, que eventualmente possuem permissões cruzadas.

- As configurações são divididas em Básicas, de Usuário e de Serviço;
- Os tipos de usuário com perfil autorizado a modificar configurações são Administrador, Suporte, Desenvolvedor, Diretor e Sistema;
- As configurações Básicas podem ser alteradas com perfil de Administrador;
- Configurações de Usuários podem ser alteradas pelo perfil de Suporte;
- Os Serviços podem ser criados e configurados pelo perfil Desenvolvedor;
- Usuários com perfil de Sistema são representações de serviços, para que eles tenham acesso disciplinado à Plataforma. Estes usuários podem alterar configurações relativas aos usuários destes serviços, adicionando características e perfis que eles usarão somente com estes serviços;
- Diretores são usuários com permissão para criar novos usuários Administradores e revogar suas permissões. Também podem escolher quais configurações podem ser alteradas livremente e quais requerem seu aceite para passarem a vigorar na Plataforma;
- Usuários com permissões cruzadas devem receber avisos automáticos por e-mail informando, formulados pela Plataforma, os informando sobre o que foi modificado, qual o conteúdo anterior, e qual o conteúdo substituto.

Também faz parte da implementação de segurança da plataforma, a função de **Balanceamento de Carga**, pois esta trabalha para que os servidores que compõe a Plataforma não fiquem sobrecarregados, seja por excesso de requisições autênticas, seja por tentativa de ataques coordenados através de técnicas maliciosas como DDoS ou BOT-NET.

3.5 SERVIÇOS, MICROSERVIÇOS E FUNÇÕES SERVERLESS

Neste tópico serão detalhados os serviços que foram projetados para o *Framework*, de natureza obrigatória. Muitos serviços opcionais também foram especificados, e serão detalhados no **Apêndice G**.

Para este *Framework*, a noção de **serviço** refere-se na verdade a uma **categoria** da qual vários serviços fazem parte e aparentam ser indivisíveis, cujo funcionamento é transparente entre suas partes. Já os **microserviços** são aqueles

aplicativos com **objetivos únicos**, que possuem poucas funções separadas em seu código, sendo estas chamadas de **ações**, quando podem ser requisitadas separadamente pelo cliente que acionou o microserviço. Quando um microserviço é tão simples que possui **uma única função indivisível** em seu código, este é chamado de **função serverless**, caso não seja necessário desenvolvimento de uma API específica para sua chamada remota. Na falta de uma tradução amigável para o termo *serverless functions*, estas serão chamadas de **comandos**.

3.5.1 **Serviços de Sessões e Acesso de Usuários**

Sessões são uma característica muito importante para servidores que precisam prestar serviços na forma de aplicações remotas, pois somente desta forma é possível manter um controle de fluxo entre as requisições anteriores e futuras de um usuário remoto, para manter a continuidade da aplicação.

O controle de acesso aos usuários cadastrados no servidor é uma parte complementar ao conceito de sessões, pois torna o ambiente mais seguro e garante que o usuário receberá somente aquelas informações pertinente a ele, e que seu conteúdo pessoal não será enviado a outro usuário por engano.

Também é necessária a abertura de uma sessão para que as requisições entre cliente e servidor possam ser criptografas, pois o servidor trabalha com chaves de segurança exclusivas para cliente com quem se comunica.

Por essa razão, estes três tipos de serviços estão na mesma categoria e são referenciados e solicitados apenas como ***/*sessão**.

3.5.1.1 Especificações de Serviço

As diretrizes para implementação de sessões, autenticação e criptografia são:

- Requisições podem ser feitas sem a necessidade de uma sessão;
- Requisições podem ser feitas sem a necessidade de um usuário autenticado, desde que a função requisitada não dependa de um tipo de usuário ativo;
- Se a requisição depender de um usuário *logado*, ela também dependerá de uma sessão aberta, exceto em caso de Sequências, onde ser pode efetuar um *login* de um usuário e logo depois chamar outras funções na requisição;
- As sessões não são obrigatoriamente criptografadas;
- As sessões podem ser criptografas, usando HTTPS com suporte do navegador, ou usando as duas opções de criptografia da Plataforma com HTTP e sendo controlada pelo código do cliente e da Plataforma;

- Uma sessão pode vir a se tornar criptografada depois de criada, assim como pode deixar de ser criptografa, por solicitação do cliente ou do serviço;
- Existem dois tipos de sessões: as temporárias e as duráveis;
- A sessão temporária é criada e administrada no Gerente de Requisições, por solicitação do cliente ou do serviço requisitado, e pode ser finalizada pelo cliente, pelo serviço ou por limite de tempo de ociosidade esgotado. É própria para o uso em navegadores Web;
- A sessão durável é criada no banco de dados (tabela *usuário_sessão*) e administrada por *stored procedures* como funções serverless, por solicitação do cliente, e não é finalizada por limite de tempo esgotado nem por fechamento do aplicativo cliente, mas somente através de uma requisição de cliente ou por data de vencimento. É própria para o uso em aplicativos desktop ou móveis, como os de *tablets* e *smartphones*, pois permitem que o usuário entre e saia constantemente do aplicativo, autenticando-se uma vez;
- O usuário tem um cadastro único na Plataforma, mas que pode conter vários cadastros de perfil de acesso atrelado a ele;
- Um perfil de usuário define o papel deste na Plataforma, como administrador, usuário simples, usuário licenciado etc. e por isso o perfil está relacionado às permissões de acesso aos serviços disponíveis;
- Um usuário pode ser bloqueado para acesso à Plataforma, ou ter apenas um ou mais de seus perfis bloqueados, por motivos diferentes;
- O cadastro de um microserviço ou função referente a sessões, autenticação ou encriptação deve ser configurado de modo a adicionar uma opção a uma função padrão já existente ou substituir esta função.

3.5.1.2 Ações de Serviço e seus Parâmetros

As ações dos serviços de sessão, autenticação e criptografia (referenciadas na URL como **/sessão/ação/**[**parâmetros**]) são comandos com parâmetros específicos, e se relacionam com o início, meio ou fim de uma sessão de usuário.

- **(ação padrão)** - se não for especificado um comando, o serviço de sessão ativa a ação padrão, que criará uma sessão temporária sem criptografia;
- **/criar** - cria uma sessão durável, mas sem criptografia, e retorna um *token*, que é um identificador único da sessão, usado para acessá-la novamente sempre que o cliente for reaberto;
- **/proteção** - protege uma sessão existente, com criptografia, ou cria uma sessão durável que já começa com criptografia do tipo **simples** ou **dupla**;

- **/recuperar** - recupera uma sessão durável existente, indicando um **token** como parâmetro para encontrá-la. Caso a sessão tenha sido criada com criptografia, ela continuará criptografa, mas com chaves diferentes;
- **/autenticar** - autentica um usuário na sessão através de **login** e **senha** como parâmetros através do método POST;
- **/sair** - finaliza uma sessão, mesmo que não tenha um usuário ativo. Caso a sessão seja durável, esta será bloqueada e não poderá mais ser usada;
- **/info** - recebe os dados da sessão, ou do usuário autenticado como nome, email e foto, ou seus perfis de acesso, ou licenças para acesso aos serviços. Depende do que for solicitado no parâmetro. Estas informações já são todas enviadas ao cliente assim que ele utiliza o comando */autenticar*;
- **/verificar** - verifica se existe uma sessão avisa se ela está ativa ou já expirou;
- **/manter** - envia uma requisição que mantem a sessão ativa, como um *ping*;

3.5.2 **Serviços de Armazenamento e Uso de Arquivos**

Servidores Web começaram servindo página estáticas, imagens e documentos para download. Apesar de hoje usarmos aplicações cuja principal característica é gerar conteúdo dinâmico na Web, a disponibilidade de arquivos ainda é uma parte crucial dos servidores Web.

3.5.2.1 Especificações de Serviço

Na Plataforma, o armazenamento e a gestão de arquivos também são muito importantes, e apresentam algumas diferenças do comportamento de um servidor comum. Seguem as diretrizes para este tipo de serviço:

- A Plataforma não dá suporte ao protocolo FTP;
- Todas as transferências de arquivos são feitas no protocolo HTTP e HTTPS;
- A manipulação de arquivos pode ser realizada pelo cliente ou por serviços;
- A Plataforma deve disponibilizar comandos para manipulação e recebimento de arquivos via URL com método GET e para envio por variáveis numa requisição tipo POST;
- Deve haver dois métodos de download: um que force o navegador a solicitar um caminho para salvar o arquivo, e outro que permita receber o arquivo silenciosamente (via AJAX ou para clientes não navegadores);
- Para referenciar um caminho (nomes de diretórios e subdiretórios) e nome de arquivo, deve-se adicionar uma arroba (@) entre ele e o restante da URL;

- A biblioteca de funções para clientes da plataforma deve conter métodos para auxiliar o desenvolvedor a informar ao usuário a estimativa de tempo e conteúdo restantes para a transferência;
- Apagar, sobrescrever, pesquisar ou adicionar conteúdo de um arquivo são ações que podem ser realizar pelo usuário ou serviço que criou este arquivo;
- Outros usuários, que não o criados de um arquivo, só podem adicionar ou pesquisar conteúdo deste arquivo, caso o repositório onde ele está, esteja configurado para permitir estas ações em comunidade.
- O cadastro de um microserviço referente a gerenciamento de arquivos deve especificar um entre dois objetivos: a manipulação de arquivos, como a conversão de um tipo de arquivo para outro, pesquisa de conteúdo etc.; ou informar um repositório, com seu local e as permissões de uso dos arquivos contidos dele, para usuários e outros serviços;

Existe uma tabela específica (serviço_arquivo) para determinar que serviço executar caso algum arquivo deste tipo tenha sido apontado na URL sem especificar nenhuma ação existente para ele. Isso permite que serviços acoplados (detalhados no Tópico 3.5.4) possam ser utilizados para executar os arquivos solicitados, inclusive usando-se interpretadores de script como PHP, NodeJS e outros sistemas que podem ser acoplados à Plataforma, assim como em diversos servidores Web.

3.5.2.2 Ações de Serviço e seus Parâmetros

Ações do Serviço de Arquivos são referenciadas na URN da requisição como **/arquivo/ação/repositório/**[**parâmetros**]**@caminho/nomeArquivo1**[**@caminho2**] e funcionam, em sua maioria, como comandos de um cliente FTP, que manipulam o arquivo como um todo. Outros comandos funcionam como se fossem funções de linha de comando, para trabalhar o seu conteúdo.

Todas as ações envolvendo arquivos logicamente precisam obrigatoriamente do nome de **no mínimo:** um **repositório** e **um arquivo** como **parâmetros**.

Já que nomes e caminhos de arquivos utilizam a barra de data (/) assim como os outros parâmetros de uma URL, deve-se separar cada nome de arquivo (incluindo seu caminho) com uma arroba, assim como também são admitidos coringas para nomes de arquivos.

- **(ação padrão)** - se não for especificado nenhum comando além do nome do arquivo, a Plataforma verificará se há algum serviço padrão para executar com este arquivo como parâmetro. Caso contrário, o conteúdo do arquivo será transferido para o cliente, que deverá estar preparado para ele;

- **/download** - força a Plataforma a enviar o arquivo para o cliente, que deve ser um navegador Web ou outro aplicativo que saiba lhe dar com download de arquivos, recebendo no cabeçalho o valor *Content-Disposition:attachment*;
- **/verificar** - recebe informações da plataforma sobre a existência do repositório ou do arquivo arquivo informados como parâmetro, assim como número de arquivos no repositório;
- **/criar** - cria um arquivo em branco ou com conteúdo enviado por POST, no caminho e nome de arquivo especificados, e por padrão não o sobrescreve caso já exista no diretório, exceto se especificado o parâmetro **sobrescrever**;
- **/enviar** - envia para o repositório um arquivo já existente no cliente, utilizando a mesma regra de sobrescrita do comando /criar;
- **/apagar** - envia uma requisição para apagar um arquivo presente no repositório;
- **/copiar** - solicita a cópia de um arquivo ou diretório, para outro diretório ou para o mesmo local, mas com outro nome de arquivo;
- **/mover** - solicita que a Plataforma mova o arquivo ou diretório para outro local ou para um nome de arquivo diferente (renomear);
- **/acrescentar** - solicita à Plataforma que adicione, o conteúdo enviado, num arquivo já existente, em seu **início**, **fim** e **antes** ou **depois** de uma **expressão** ou **linha** específica, de acordo com o parâmetro de **pesquisa**, que indica uma posição ou texto a ser localizado para o critério de inserção;
- **/alterar** - comando complexo que utiliza os mesmos parâmetros de pesquisa do comando /acrescentar, mas que possui um parâmetro que permite especificar um **tamanho** de dados a serem atualizados. Este pode ter um tamanho especificado por um número ou por uma **expressão** informada como parâmetro, tanto de início como de fim, como último parâmetro, solicita **excluir, substituir** ou **retornar** o conteúdo, por dados enviados na requisição. Estes parâmetros podem ser simplificados ou até omitidos caso o repositório já esteja configurado com expressões de pesquisa e substituição padrões;

3.5.3 **Serviços de Bancos de Dados**

Este tipo de serviço é o mais exclusivo deste *Framework*, pois os servidores de aplicação convencionais esperam que as aplicações acopladas a eles acessem o banco de dados e formulem a resposta ao usuário com base em seus controladores.

Em contrapartida, este *Framework* define que a própria Plataforma fará a ponte de comunicação entre o cliente e a execução de *stored procedures* no banco de dados para coletar e converter a resposta no formato solicitado, isentando o desenvolvedor da necessidade de criar APIs, web-services e outros tipos de aplicativos para este fim.

Este modelo de funcionamento permite cortar uma etapa complexa no desenvolvimento de sistemas, além de economizar recursos de processamento e memória dos servidores, inclusive no SGBD, que fica com apenas uma conexão aberta pela Plataforma para servir às requisições internas e outra para servir às requisições do cliente, para cada SGBD configurado na Plataforma.

Na necessidade de uma nova funcionalidade para aplicações, relacionada a informações, basta desenvolver a *stored procedure* que acessará esta informação nas tabelas do banco, registrá-la na Plataforma, e ela ela estará pronta para ser acessada pela camada do cliente.

3.5.3.1 Especificações de Serviço

As diretrizes para implementação de serviços de bancos de dados são:

- Cada banco de dados a ser usado na plataforma deverá ser registrado como um serviço, na categoria de banco, cujas configurações incluem seus dados de localização e autenticação, incluindo formas de backup, espelhamento e rotas alternativas;
- As permissões de autenticação nos bancos de dados para a plataforma só podem permitir a consulta SELECT e a chamada CALL para procedures;
- Cada *stored procedure* faz o papel de uma ação (função) e também deverá ser configurada individualmente, registrando seus parâmetros e seu comando interno (CRUD) em relação aos dados: inserir, alterar, apagar, consultar etc.;
- Poderão existir funções com nomes repetidos, desde que estejam em bancos de dados diferentes;
- Os nomes de *stored procedures* serão os mesmos nomes de seu comando na Plataforma, suprimindo-se o prefixo *sp*, usado por alguns DBA;
- Parâmetros de *stored procedures* que podem ser nulos ou podem adotar valores padrão deverão ser os últimos parâmetros destas procedures;
- Funções que retornem listas potencialmente grandes devem ter seu código de paginação e limitação de linhas na própria *stored procedure*, e definidas por parâmetros, para evitar a sobrecarga de memória e processamento na Plataforma;

- O modelo de saída de procedures e microserviços em caso de erros é o mesmo por convenção da Plataforma. O resultado de uma consulta ou tentativa de inclusão ou alteração de dados malsucedida deve ser programada na *stored procedure* para retornar apenas uma linha com duas colunas: o código de erro e sua descrição;
- Os parâmetros sem nome passados por URL serão passados à *stored procedure* na mesma ordem em que estiverem dispostos na URL;
- Os parâmetros nomeados passados pelo método POST (como por exemplo os valores de um formulário em uma página HTML) que tiverem o mesmo nome dos parâmetros da procedure, terão seus valores usados na ordem em que estes parâmetros estão definidos no registro de parâmetros da procedure, sem a necessidade de preparar a URL.

3.5.3.2 Ações de Serviço e seus Parâmetros

As ações do Serviço de Arquivos são referenciadas na URL como **/dados/**[**nomeBanco**]**/nomeProcedure/parâmetros** onde cada ação é uma *stored procedure* registrada na Plataforma, sem exceção.

Os **comandos padrões** da Plataforma que estão disponíveis através *stored procedures*, como aqueles responsáveis pelo gerenciamento de usuários, não precisam ser referenciados com o nome do banco na URL, exceto se tiverem sido implementados em um banco de dados diferente do banco de dados de configuração e operação da Plataforma.

Vale lembrar que os parâmetros podem ser passados de quatro maneiras diferentes, em qualquer combinação:

- Através **parâmetros nominados** omitidos no método POST;
- Através de parâmetros convencionais de URL, usando &n1=v1&n2=v2;
- Através da URL, no formato **/nome:valor**; ou
- Através da URL, somente com os **valores ordenados** (v1/v3/v3/...) de acordo com os parâmetros da *stored procedure*, o que pode ser consultado pelos desenvolvedores na configuração ou através do comando */dados/[nomeBanco]/nomeProcedure/** para obter a descrição do comando;
- No caso de serviços de bancos de dados, os **parâmetros padrões da plataforma** (como os formatos de **entrada** e **saída** da requisição) deverão ser passados somente de uma das duas primeiras formas, ou seja, **com o nome e valor explícitos**, para evitar que sejam confundidos com parâmetros da *stored procedure* no momento do processamento da requisição;

- A requisição de execução de uma procedure só será aceita se todos os parâmetros definidos como obrigatórios na configuração desta procedure forem entregues na requisição, inclusive considerando a formatação correta dos tipos de dado de cada parâmetro;
- Para passar um parâmetros com valor em branco, mais adicionar uma barra sem conteúdo (/p1/p2//p4 ou /n1:v1/n2:v2/n3:/n4:v4 ou &n1=v1&n2=&n3=v3);
- Para passar um parâmetro de valor nulo, usar o termo *NULL* como valor.

3.5.4 **Serviços Customizados Acoplados na Plataforma**

Como já citado no Tópico 3.5.3 a Plataforma poderá obter informações do banco de dados sem a necessidade de desenvolvimento de APIs. Porém nem todas as necessidades informacionais podem estar disponíveis com este processo.

Nos casos em que os SGBDs suportados pela Plataforma não disponham de meios ou comandos para tratar os dados exclusivamente dentro de seu próprio ambiente de execução, é sim necessária a intervenção do desenvolvedor, que precisará implementar mecanismos complementares aqueles oferecidos pela SQL.

Alguns SGBDs até dispõe de tecnologias para se criar *stored procedures* em linguagens alheias ao SQL, como Java ou C++. Porém, se isto não for uma realidade no SGBD escolhido pelo desenvolvedor, o *Framework* oferece uma alternativa, permitindo o desenvolvimento de novos serviços para a Plataforma.

3.5.4.1 Especificações de Serviço

Um serviço customizado pode ter um tamanho variável, e pode ser desde uma simples função, ou um microserviço com vários comandos, até mesmo uma aplicação grande o suficiente para ser considerada monolítica, o que permite que a Plataforma seja compatível com aplicações já existentes, e que podem ser acopladas a ela para funcionar de forma transparente ao desenvolvedor, à camada cliente e ao usuário final. Dependendo da linguagem de programação utilizada para a implementação da Plataforma, cada serviço pode ser acoplado de 4 formas:

- **Módulos executáveis externos**: são aplicações do tipo **console** (que podem ser inclusive arquivos de lote) totalmente independentes da plataforma, mas que são compatíveis com o SO em que a plataforma está sendo executada, caso estejam alocados na mesma máquina. Devem seguir as convenções gerias do *Framework* para entrada e saída de dados, bem como convenções específicas para módulos externos;

- **Módulos de serviços remotos**: são serviços alheios à rede da Plataforma, que estão espalhados pela Web, mas que podem ser utilizados pelo cliente ou por outros serviços, como por exemplo APIs de pagamentos (PayPal), de rastreamento de objetos (Correios) e de processamento de tarefas, como a smallpdf.com, que permite compressão e conversão de documentos via URL;
- **Módulos em classes na mesma linguagem da Plataforma**: em linguagens como PHP, Python, Pearl, Ruby, Java, JavaScript e Scala, cada classe fica disponível em um arquivo separado, sem serem compilados para linguagem de máquina. Sua natureza de execução interpretada permite que classes previamente conhecidas sejam usadas para instanciar objetos e executar tarefas (ALURA, 2016). Desta forma, é possível que novas classes possam ser desenvolvidas, para funcionarem como **microserviços**, cujos métodos públicos sejam ações acessíveis ao cliente, ou **comandos**, com apenas um método público que possa ser executado na chamada da requisição. Cada linguagem tem seu próprio método de acesso a esse tipo de classe;
- **Módulos em classes internas da Plataforma**: em linguagens como C, C++, Pascal e outras cujo código fonte é compilado em código de máquina, este fica encapsulado num sistema fechado, que se comunica com o SO através de mensagens. Seu código não consegue interagir com classes feitas na mesma linguagem, caso não tenham sido compiladas juntas. A cada módulo interno criado, o núcleo executável da Plataforma precisa ser recompilado, o que fere o princípio de desacoplamento.

Independentemente da forma de acoplamento de cada serviço customizado, todos eles devem seguir as mesmas diretrizes listadas abaixo:

- Todo tipo de módulo customizado deve ser registrado na configuração da plataforma para poder ser "enxergado" pelo serviço de diretório e ficar acessível ao cliente;
- Além dos parâmetros necessários para que os módulos realizem suas funções, eles devem ter como primeiro parâmetro obrigatório o ID da sessão de usuário, para que possam abrir o arquivo (gerado pelo GRP na Etapa 6 e 7) contendo os dados da sessão e da requisição que os executou, evitando uma enorme quantidade de dados passados através de parâmetros;
- No caso de módulos internos, compilados ou não, o nome de um módulo deve ser configurado da forma como estará disponíveis ao cliente, juntamente com o nome real da classe corresponde ao comando. Não é necessário anexar prefixos ou sufixos na classe, mas é recomendado que se escolha uma forma de diferenciá-las das demais classes da Plataforma.

3.5.4.2 Ações de Serviço e seus Parâmetros

As ações dos serviços customizados na Plataforma são os próprios serviços, o que significa que cada módulo acoplado deve ter apenas uma ação padrão, e seu comportamento depende dos parâmetros recebidos, o que os qualificam como microserviços ou comandos, sendo a URL como **/modulo/nome/**[**parâmetros**].

- As regras para parametrização das requisições aos serviços customizados são as mesmas regras aplicadas aos serviços de bando de dados, descritos no Tópico 3.5.3 anterior a este, exceto:
- Módulos externos acoplados são similares a comandos de linha de consoles, e por isso não aceitam pulos de parâmetros (em branco). Seu funcionamento interno deve prever a sua própria necessidade de permitir que alguns parâmetros sejam ou não emitidos, interpretando sua linha de comando como um todo, o que eventualmente exigirá o uso de formatações customizadas.

3.5.5 **Serviços de Gestão da Plataforma**

Alguns dos servidores Web mais utilizados no mundo (W3TECHS, 2017) como o Microsoft IIS (C# e ASP) e JBOSS (Java e Scala) possuem um painel visual para que o administrador o configure, o que é uma característica de quase tudo em seus SO Windows. Mas outros que estão na mesma lista, como o Apache e NGINX não possuem nem um painel de administração visual nem sequer um painel de console (WIKIPÉDIA, 2017), sendo que todas as suas configurações devem ser feitas editando seus arquivos de configuração com um editor comum.

Muitos profissionais experientes preferem não depender de interfaces gráficas para gerenciar servidores, pois a maioria os administra de forma remota através de terminais console com o protocolo SSH. Mas muitos profissionais iniciantes se beneficiariam de uma interface gráfica para administrar serviços remotos.

3.5.5.1 Especificações de Serviço

Para manter baixa a curva de aprendizado, o *Framework* exige que as duas opções de acesso sejam implementadas:

- **Acesso via Console**: todas as configurações da Plataforma podem ser consultadas, criadas ou alteradas através de uma aplicação que dá acesso a

uma linha de comando para executar funções que permitem administrar usuários, serviços e preferências;

- **Acesso via Painel Visual**: na forma de uma aplicação Web modular, o Painel Administrativo é acessado por HTTP via URL e deve permitir as mesmas funcionalidades que a aplicação de linha de comando;

As seguintes diretrizes valem para os dois métodos de administração:

- Somente usuários com Perfís elevados tem acesso às ferramentas de administração da Plataforma. As configurações não devem ser acessíveis por módulos externos. Cada usuário com perfil administrativo tem acesso seletivo às configurações por tipo de serviço. Somente o super-usuário tem acesso a todas as configurações;
- Depois que qualquer alteração de configuração é feita, o painel ou console responsável pela alteração envia um requisição ao Gestor de Requisições avisando desta alteração, que por sua vez recarrega as configurações e solicita que todos os servidores integrados que façam o mesmo.

3.5.5.2 Ações de Serviço e Seus Parâmetros

As ações do Serviço de Gestão da Plataforma são referenciadas na URN como **/painel/**[**módulo**]**/**, onde cada ação é um módulo administrativo que surge como ferramenta visual para configurar um tipo de serviço ou dados da Plataforma, sem a necessidade de parâmetros, exceto login de usuário via POST, quando desejado pelo cliente, ou através de uma sessão ativa no navegador. Os módulos implementáveis até o momento são:

- **(ação padrão)** - carrega um painel de configurações primárias da plataforma, como o cadastro de usuários administrativos e opções customizáveis;
- **/rede** - carrega o painel com configurações de rede, como o registro de servidores de backup, de espelhamento e rotas alternativas para o Gestor de Balanceamento de Carga;
- **/bancos** - carrega o painel que administra os bancos de dados que serão usados na plataforma e nos serviços;
- **/módulos** - carrega o painel que administra todos os módulos customizáveis;
- **/repositórios** - carrega o painel que gerencia os repositórios de arquivos, modelos de saída (templates) e serviços de transmissão de multimídia, determinando limite de espaço em disco, localização física, métodos de utilização, perfis permitidos, rotas alternativas, alocação de banda, limites etc.;

- **/licenças** - carrega um painel que funciona como um aplicativo CRM para administração de licenças, comissões e outras funções relacionadas a aquisições de usuários e permissão de uso de serviços;
- **/mensageria** - administra o cadastro de canais, filas e tópicos do serviço de mensageria, bem como configurações como limites de usuários, tempo de guarda de informações etc;
- **/sequências** - painel onde se cria e administra sequências de serviços, especificando sua ordem, comunicação, entradas e saídas.

3.5.6 **Serviços do Núcleo da Plataforma**

Como visto no **Tópico 3.2.1**, o Gerente de Requisições é responsável pelo fluxo de cada requisição, que pode ou não sair momentaneamente de sua gestão ativa para executar um módulo externo (acoplado). O seu fluxo tem uma considerável complexidade de 17 etapas onde a requisição passa por diversos processos em sequência, como se estivesse em uma linha de montagem industrial. Cada processo é realizado por uma função interna específica do GRP.

Isso permite não só a modularização e organização do projeto, como também permite que tais funções possam ser substituídas por versões melhoradas na forma de módulos externos, de acordo com a necessidade de cada desenvolvedor.

Um exemplo disso é o módulo de Bibliotecas de Codificação, que possui por padrão a capacidade de converter conteúdo de um formato para outro, como XML para JSON e vice-versa. Um módulo externo poderia ser desenvolvido com capacidade para trabalhar com mais formatos do que o módulo interno original.

Outra grane vantagem da modularização, é que algumas funções podem sair do escopo de trabalho de processamento do fluxo de requisições e entrar na lista de funções disponíveis para serviços da plataforma, evitando a necessidade de implementar nos serviços uma função que a Plataforma já tenha em seu núcleo.

3.5.6.1 Especificações do Serviço

O **Quadro 14** lista todos módulos de funções internas do Gestor de Requisições, indicando quais deles são acessíveis a outros serviços e quais podem ser substituídos por outros serviços disponíveis na Plataforma, lembrando que o GRP não possui apenas funções relacionadas ao fluxo, mas também carrega a implementação das categorias de serviço básicas da Plataforma, como sessões, segurança, arquivos, usuários e bancos de dados.

A substituição é feita quando na configuração de um serviço acoplado é inserida a propriedade **ServiçoDeNúcleo**, indicando em seu valor o **nome do módulo** do GRP a ser substituído.

Quadro 14 - Módulos do Gestor de Requisições

Módulo	Função	Acesso	Substituível?
Monitor de Entrada	Monitorar as requisições através das portas HTTP, TCP e UDP	Nenhum	Não
Motor de Criptografia	Des/criptografar dados em trânsito, armazenados ou em bancos de dados	Total	Sim
Decodificador de Entrada	Identificar e organizar serviços e parâmetros	Leitura	Não
Leitor de Configuração	Consultar todas as configurações da plataforma	Leitura	Não
Gestor de Sessões	Controlar as sessões de acesso à plataforma	Leitura	Sim
Balanceador de Carga	Monitorar os limites de *hardware* e da rede e re-encaminha requisições	Leitura	Sim
Biblioteca de Formatos	Converter dados de um formato para outros	Total	Sim*
Gestor de Execução	Executar funções internas ou externas solicitadas	Nenhum	Não
Codificador de Saída	Formatar e compactar os dados que serão retornados ao cliente ou a outro serviço	Total	Sim*
Gerador de LOGs	Registrar informações técnicas sobre os processos executados e seus resultados	Total	Sim
Gestor de Arquivos	Controlar o acesso aos arquivos dos repositórios	Total	Sim*
Gestor de Banco de Dados	Controlar o acesso às funções dos bancos de dados	Chamada	Não*
Gestor de Usuários	Controlar as informações de usuários e seus perfis	Total	Não

Fonte: O autor (2017)

* Alguns módulos, substituíveis ou não, podem ser expandidos para se adicionar novas funções, sem que se deixe de utilizar as funções existentes, adicionando à configuração do serviço acoplado a propriedade **ModoDeNúcleo** com o valor **Expansão** ao invés de Substituição.

3.5.6.2 Ações de Serviço e Seus Parâmetros

As Funções de Núcleo da Plataforma não são acessíveis diretamente à camada cliente nem ao usuário final. Seu objetivo é reduzir a necessidade de implementação destas funções dentro de serviços acoplados à Plataforma, e somente estes poderão ter acesso a eles, dentro da rede da Plataforma. A URN usada para acesso é referenciada como **/núcleo/útil/módulo/ação/parâmetros**. A quantidade e obrigatoriedade dos parâmetros depende de qual ação será solicitada, e os comandos e parâmetros mais comuns estão listados abaixo:

- **/cripto** - é o comando para ações de **/criptografar e** /**descriptografar** um parâmetro na requisição ou um @**arquivo** indicado;
- **/entrada** e **/saída** - são dois comandos que podem ser utilizados por serviços para identificar e modificar dados da requisição. Mesmo que o GRP entregue os parâmetros para o serviço executado e recolha seus resultados para transformar em saída, o serviço pode também recorrer diretamente às funções de núcleo para controlar estes dados. Na /entrada estão os parâmetros, arquivos enviados ao servidor (uploads) etc. Na /saída é possível preparar objetos, arquivos e mensagens que serão entregues ao cliente;
- **/config** - recupera alguma informação de configuração disponível, utilizando como parâmetro a **/categoria** de configuração, e nome do **/objeto** envolvido. Também é usado para informar uma **/atualização** de configuração feita por um administrador da Plataforma, forçando o GRP a recarregá-las;
- **/carga** - recupera informações de ***/hardware***, ***/software*** e **/rede** relacionadas à capacidade, utilização e disponibilidade atual de recursos à Plataforma. Também possui o subcomando **/redirecionar** que permite ao serviço indicar um site externo como alvo, ou mesmo um arquivo html interno;
- **/converter** - executa um dos **/formatador**es disponíveis para converter um parâmetro da requisição ou um @**arquivo** indicado, como por exemplo, de tabela de banco de dados para Excel, ou de RichText para Word;
- **/log** - Permite **/gravar** uma mensagem ou **/pesquisar** a ocorrência de destas nos arquivos de LOG da Plataforma;
- **/usuário** - permite **/adição**, **/mudança**, **/remoção** e **/bloqueio** de prerrogativas relacionadas aos usuários da Plataforma e seus perfis, desde que sejam pertinentes ao **/serviço:id** que requisitou a ação;

Observação: o*s serviços de Sessões, Arquivos e Banco de Dados já foram detalhados nos* ***Tópicos 3.5.1*** *até* ***3.5.3****.*

3.5.7 **Serviços em Sequência**

O conceito de Sequência foi introduzido no **Tópico 3.2.1.5** e trata-se de um método para permitir a Orquestração e Coreografia de Serviços como Processos que seguem um fluxo pré-determinado e coordenado pela Plataforma, a fim de obter-se um resultado mais complexo do que o permitido com a execução de apenas um microserviço, visto que suas etapas não são exclusivas de daquele processo e por isso não devem ser reimplementadas com esse único objetivo.

3.5.7.1 Especificações do Serviço

Conforme especificado no **Tópico 3.5.7**, os serviços de Modelo de Saída para o Cliente possuem uma capacidade de executar outros serviços de qualquer categoria, na ordem desejada, assim como podem decidir quais serviços serão executados antes e depois de sua própria execução.

Já a Sequências é o método padrão de execução orquestrada de serviços, mesmo que entre eles não haja chamada a algum serviço de modelo de saída. Suas especificações estão listadas abaixo:

- Uma Sequência é um serviço registrado na categoria de Sequencias;
- Uma Sequência se define pela chamada a várias requisições que seguem uma ordem imutável, transferindo os dados de saída de o primeiro serviço executado como entrada de dados do próximo serviço, até que o último serviço seja executado, quando a saída deste será enviada para o Cliente, juntamente como um relatório de execução de todos os serviços executados;
- O tamanho de uma sequência é medida em número de arestas e vértices, onde as arestas são direcionadas e se referem às requisições nela contida, já os vértices são numerados e referem-se aos serviços envolvidos nesta sequência, que podem ser executados mais de uma vez nela;
- Na configuração de uma sequência, as requisições devem ser listadas na sua ordem de execução, incluindo os nomes dos parâmetros a serem recebidos, nos formatos /requisição/$parâmetro1/$parâmetro2/@$caminhoEarquivo3;
- Os parâmetros enviados na requisição que chama a Sequência podem ser ser usados em mais de uma requisição dentro da Sequência, desde que seu nome seja citado na configuração do comando desta requisição;
- Ao se deparar com um erro avisado por um dos serviços em execução numa Sequência, a mesma é interrompida e a Plataforma envia ao cliente somente o resultado obtido até aquele ponto, seguido do relatório de serviços e erros.

3.5.7.2 Ações de Serviço e Seus Parâmetros

Os Serviços de Sequência são requisitados através da URN **/sequência/nome/parâmetros** e mesmo que suas ações já tenham sido coreografas de modo a dependerem umas das outras, também é possível ter flexibilidade, conforme as opções abaixo:

- Os **/parâmetros** passados pelo cliente são utilizados nas requisições que a Sequência executa, de forma coreografada pelo desenvolvedor que a configurou, e é possível inseri-los pela sequência em que são registrados na nos comandos na configuração da Sequência;
- É possível **/iniciar** uma Sequência especificando o nó pelo seu número sequencial (aresta), desde que entregue todos os parâmetros requisitados por este nó e pelo restante da Sequência, usando seus nomes;
- Também é possível **/terminar** uma Sequência num nó específico que não seja o último e não esteja antes do nó selecionado como inicial.

3.6 CONSIDERAÇÕES FINAIS DESTE CAPÍTULO

O desenvolvimento do projeto foi iniciado pelo detalhamento dos objetivos e arquitetura do *Framework*, passando pela divisão das camadas cliente e servidor, e finalizando pelos serviços da Plataforma, descrevendo cada item com detalhes de suas especificações e comportamento esperado, para permitir que um desenvolvedor qualificado e experiente possa implementar uma versão da plataforma que seja totalmente compatível com o *Framework* proposto.

Foram abordadas questões de segurança, consumo de recursos, balanceamento de carga, compatibilidade com tecnologias já existentes, convenções e aprendizagem, tando na questão do desenvolvimento e manutenção, quando na questão de uso e configuração.

Todas as funcionalidades estão dentro do que é considerado implementável e que é possível alcançar os objetivos almejados, individualmente em termos de serviços e, globalmente, a respeito da Plataforma como um todo.

4 **APLICAÇÕES E RESULTADOS**

Neste capítulo são indicadas as aplicações pertinentes do produto no mercado atual usando situações existentes e comparando-as com as soluções adotadas atualmente. Também são demonstrados os resultados dos testes já realizados com protótipos que utilizam parte das especificações levantadas neste projeto, comparando os resultados estatísticos com software de mesmas funções que foram desenvolvidos com as tecnologias tradicionais.

4.1 APLICAÇÕES TEÓRICAS

É somente implementando a Plataforma em sua totalidade e testando-a em ambiente comercial que será possível acompanhar sua capacidade total e real alinhamento com os padrões de mercado e expectativa de seus projetistas e desenvolvedores. Mesmo assim é possível predefinir padrões e expectativas de utilização, comportamento e resultados de acordo com escalas padrões de mercado.

4.1.1 **Públicos Alvo**

É interesse que o *Framework* possa permitir o desenvolvimento de Plataformas que caibam às necessidades da maioria dos proprietários de soluções que precisem dos serviços oferecidos pelo *Framework*.

Abaixo estão listados os público-alvos desejáveis para utilizar a plataforma, indicando os benefícios nestes segmentos:

- **Desenvolvedores Autônomos, *Startups* e Microempresas**: para o desenvolvimento de pequenas soluções, os serviços básicos da Plataforma, como Sessões, Banco de Dados e Modelos de Saída para o Cliente são um pacote que permite automatizar a maioria dos processos e agilizar a criação das funções customizadas de um ou mais sistemas;
- **Empresas de Pequeno e Médio Porte**: a primeira coisa que empresas que cresceram precisam é de automação comercial para manter os negócios sob controle. Empresas precisam de módulos de vendas, compras, estoque, recursos humanos, fornecedores e contabilidade, além da possibilidade de venderem seus produtos e serviços pela Internet. Para estas demandas, a plataforma conta com capacidade para módulos acoplados que podem ser desenvolvidos sob medida para seu proprietário ou obtidos externamente;

- **Corporações e Aplicações de Acesso Massivo**: grandes empresas e multinacionais dependem de uma rede distribuída e segura, com alta disponibilidade e capacidade de roteamento e opções de emergência, serviços que são incorporados por padrão na Plataforma;
- **Governo e Organizações Internacionais**: um dos maiores problemas na informatização de instituições e do governo em geral, especialmente no Brasil e outros países em desenvolvimento, é o isolamento dos sistemas de informação em seus departamentos e órgãos. A maioria deles não conversa entre si e por isso são incapazes de obter informação remotamente. Isso é tão comum que até as secretarias de uma mesma prefeitura não são capazes de obter dados necessários a seus próprios processos, o que as torna ineficientes. Se todas as instituições governamentais utilizassem o *Framework* da Plataforma para desenvolver seus sistemas e serviços, todos teriam o potencial de se comunicar, bastando a inserção de uma instituição interessada como usuária dos serviços de outra. Até mesmo o cadastro de cidadãos poderia ser centralizado ou distribuído com opção de checagem cruzada e atualização recorrente entre servidores.

O *Framework* pode adotar o Programa Brasileiro de Qualidade do Software (PBQP-Sw) e o Modelo de Melhoria de Processos de Software (MPS.BR) para a Plataforma, mesmo que estes métodos não sejam usados por mais da metade das instituições governamentais segundo uma pesquisa oficial (TCU, 2008).

4.1.2 **Escalabilidade**

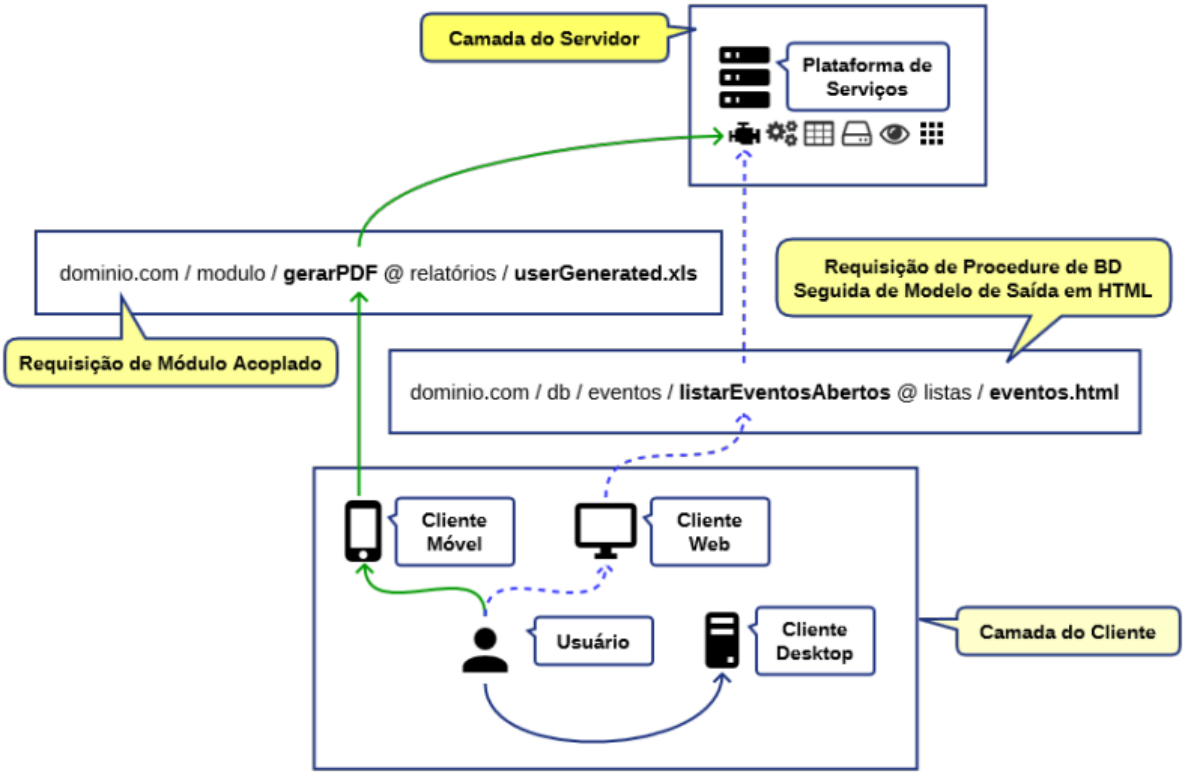
O *Framework* foi projetado com a premissa de que as plataformas desenvolvidas sejam capazes de atender um amplo espectro de necessidades, que podem ser catalogadas em, dentre outras variáveis, as seguintes:

- Quanto ao tamanho e distribuição da infraestrutura;
- Quanto ao número de serviços e recursos disponíveis;
- Quanto ao número e distribuição de usuários atendidos;

Um dos fatores mais importantes que definem se a Plataforma irá trabalhar em alto rendimento com qualquer tamanho de variáveis é sua capacidade de ser escalável a todas elas.

Os **Diagrama 2, 3** e **4** mostram exemplos de níveis de arquitetura no *Framework* projetado, de centralizada a distribuída, seguindo uma prática de escalabilidade de infraestrutura distribuída por serviços.

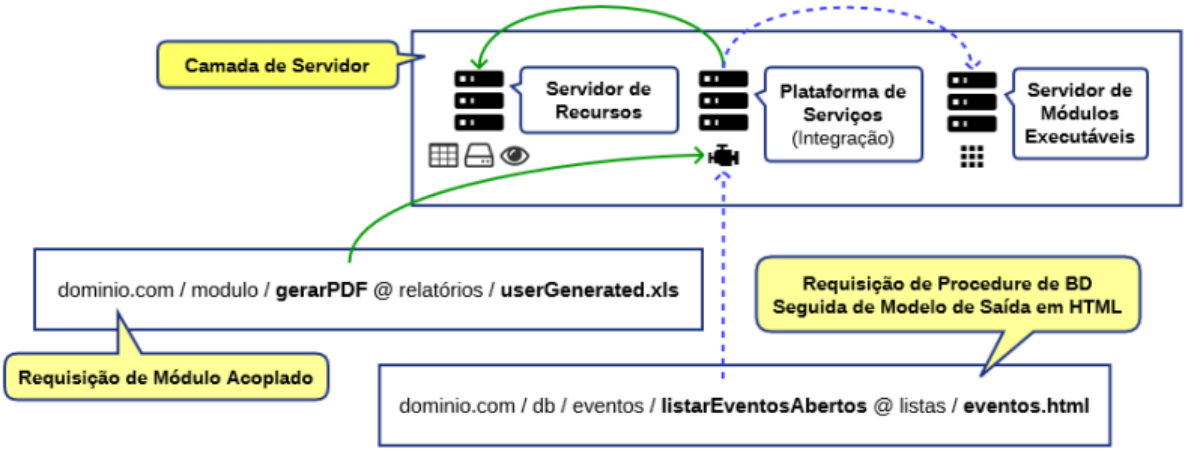
Diagrama 2 - A Plataforma em uma arquitetura não-distribuída



Fonte: O autor (2017)

O **Diagrama 2** demonstra os clientes de diversos tipos de plataformas, acessando recursos diferentes que estão disponíveis no mesmo servidor.

Diagrama 3 - A Plataforma em uma arquitetura parcialmente distribuída



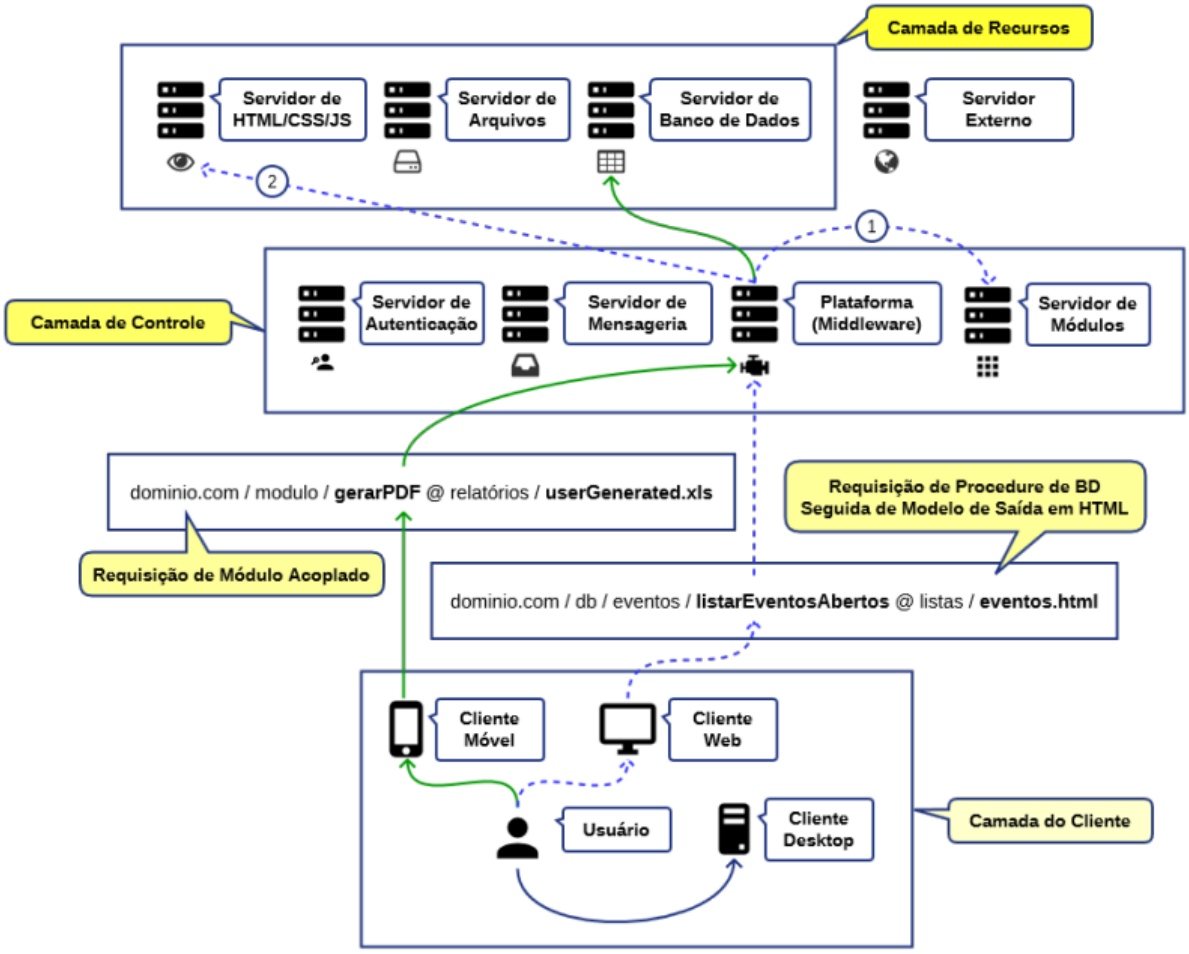
Fonte: O autor (2017)

O **Diagrama 3** demonstra os mesmos serviços de antes, porém agora num modelo de Plataforma distribuída entre três servidores, a própria plataforma, atuando como um middleware de integração, mais outros dois servidores, um para módulos acoplados e outro para os demais serviços.

As principais características do *Framework* que trabalham estes mecanismos são sua capacidade de **mediar a comunicação** entre seus componentes (servidores, serviços, usuários) sem perda de informação e sem bloqueio de processamento, e também sua capacidade de **administrar os recursos** disponíveis em todos os seus componentes de infraestrutura (*hardware*, *software* e rede).

Se estas duas especificações do *Framework* estiverem bem implementadas, a Plataforma poderá crescer tanto verticalmente quanto horizontalmente (servidores e clientes) sem que isso afete sua eficiência e eficácia.

Diagrama 4 - A Plataforma em uma arquitetura totalmente distribuída



Fonte: O autor (2017)

O **Diagrama 4** demonstra os mesmos clientes, acessando os mesmos serviços de antes, porém agora num modelo de plataforma totalmente distribuída, onde cada servidor é responsável por uma única categoria de serviços. Tanto que na requisição composta por chamada à uma procedure e a um modelo de saída, o a Plataforma acessar dois servidores, em sequência, para completar a requisição.

4.2 APLICAÇÕES PRÁTICAS

Este projeto é uma iniciativa pensada desde 2012, com o objetivo de criar uma Plataforma com toda a infraestrutura necessária para inserir novas aplicações, que seriam desenvolvidas mais rapidamente por terem o foco de desenvolvimento em suas funções principais, deixando os requisitos não funcionais como uma das responsabilidades da Plataforma, que inclusive compartilharia o cadastro de usuários e pessoas com todas os serviços, usando esses dados seletivamente.

Desde então um longo trabalho de pesquisa vem sendo feito, e a ideia evoluiu para criar vários conceitos e integrá-los com conceitos já existentes. Uma destas ideias é a utilização de procedures bancos de dados como comandos acessados pelo cliente com mediação da Plataforma, mas sem a necessidade de criar novas APIs para cada procedure. Porém, depois de alguns anos procurando material sobre este assunto, nada foi encontrado. O que poderia significar duas coisas: ou os termos pesquisados não estavam de acordo com o que estava sendo proposto, ou o conceito nunca fora cogitado por algum tipo de impossibilidade de seu funcionamento, ou ainda por um desconforto da comunidade de desenvolvimento em perder o controle e responsabilidade por esta camada de desenvolvimento.

4.2.1 **Testes de Laboratório**

Para responder a pergunta de se esta forma de utilização do banco de dados é possível e viável, em 2014 foi desenvolvido um protótipo deste conceito, utilizando MySQL 5 como mecanismo de banco de dados, PHP 4 como linguagem de programação para um gestor de requisições de procedures, e Apache Server 2 para receber as requisições HTTP. O resultado foi promissor, pois as procedures estavam sendo executadas e o usuário final recebe o resultado enviado pelo banco de dados, mas na formato especificado como objeto JSON.

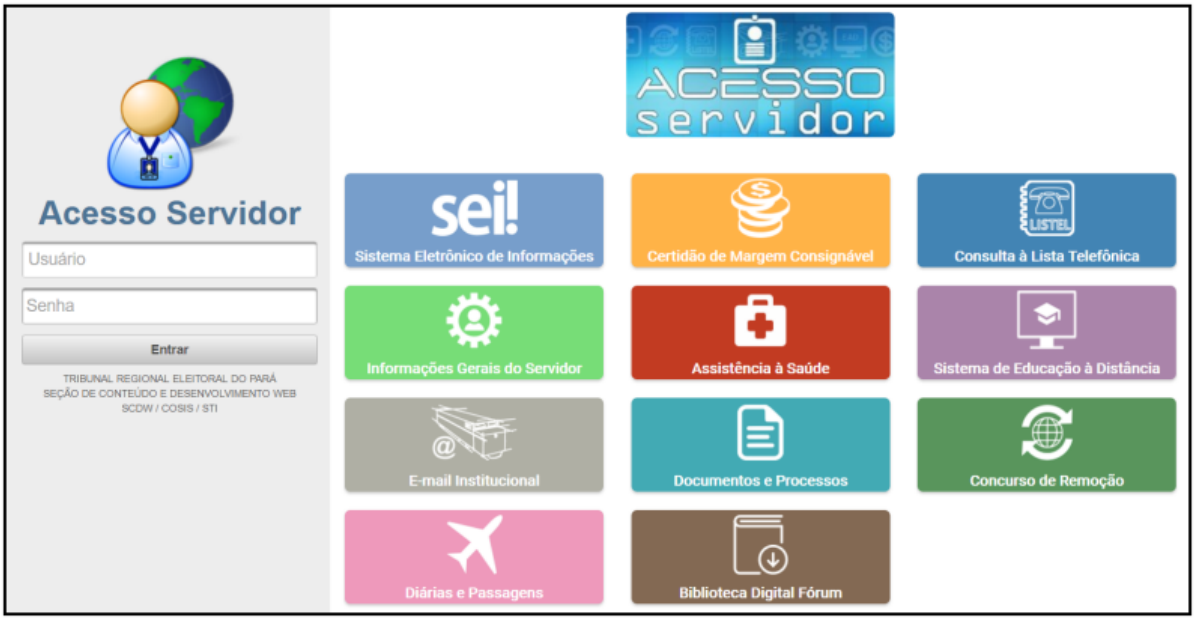
4.2.2 **Implementação Parcial**

Em 2016 surgiram duas oportunidades de implementar parte do *Framework*. Na condição de técnico judiciário alocado na Coordenadoria de Sistemas do Tribunal Regional Eleitoral do Pará, este autor foi incumbida a tarefa de projetar e implementar dois novos sistemas.

4.2.2.1 Sistema de Acesso do Servidor

O objetivo desta demanda era desenvolver uma nova versão do Sistema Acesso Servidor, cuja função é disponibilizar acesso a sistemas internos da instituição para que seus funcionários pudessem utilizá-los em ambiente seguro, mesmo estando fora da rede interna do Tribunal.

Figura 2 - Telas de login e operação do Sistema Acesso Servidor



Fonte: O autor (2017)

O sistema deveria ser desenvolvido em Java, a mesma linguagem de sua versão anterior. A nova versão deveria modernizar a interface com o usuário, apresentando um design mais modelo e limpo, conforme a **Figura 3**, além de ser responsivo, que é a capacidade de modificar o layout para adaptar-se a diferentes tamanhos de tela, incluindo telas de *smartphones* e *tablets*.

Esta foi uma boa oportunidade de comparar a eficácia e performance do *Framework* de Plataforma em contrapartida aos *frameworks* Demoiselle, JavaServer Faces e JavaBeans utilizados na primeira versão do sistema.

Mesmo que ainda utilizando Java, a nova versão do sistema não utiliza mais os *frameworks* citados anteriormente. Em relação ao *Framework* da Plataforma, foram implementados **serviços de sessão**: os comandos **/autenticar** e **/sair**. Também foi criado um comando de **/redirecionamento** através do servidor, pois a os aplicativos contidos no Sistema Acesso Servidor não podem ser acessados diretamente do navegador do cliente. Eles só aceitam requisições vindas de um servidor da rede interna do Tribunal, cujo IP esteja registrado na aplicação.

Conforme especificação do *Framework*, os três comandos desenvolvidos como microserviços desta mini-plataforma agora estão disponíveis para uso de quaisquer outras aplicações futuras, seja na forma serviço interno para sistemas acoplados no mesmo servidor, seja na forma de APIs para sistemas operando em outro servidor fora da plataforma.

O sistema foi desenvolvido e testado em apenas duas semanas.

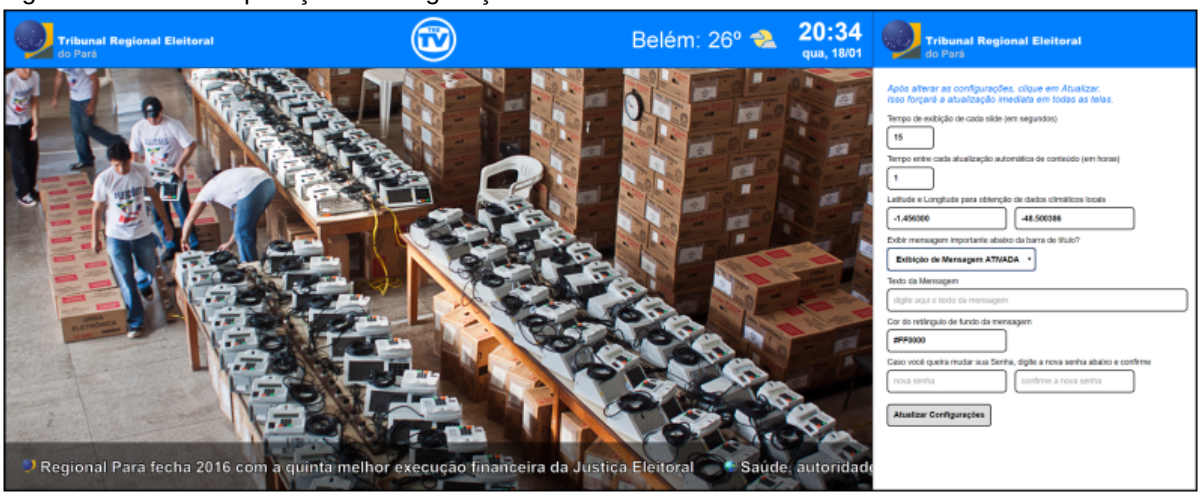
4.2.2.2 Sistema de TV Institucional

O objetivo desta demanda é desenvolver um novo sistema, para atender à solicitação da Assessoria de Comunicação, de criar uma TV institucional com divulgação de notícias internas e informações úteis aos funcionários, sem a necessidade de contratar serviços terceirizados.

O sistema poderia ser desenvolvido em Java ou PHP, já que não havia versão anterior estabelecida, e as duas linguagens são padrões de uso no Tribunal. Então a linguagem PHP foi selecionada, para permitir um teste prático com outra linguagem além do Java.

A aplicação era composta por uma interface preparada para monitores de TV, seja como saída para um computador, seja através de navegador de SmartTV, e também por uma área administrativa acessível também por navegadores, para que o operador pudesse configurar e atualizar o conteúdo da TV, conforme a **Figura 4**.

Figura 3 - Telas de operação e configuração do sistema TRE-TV



Fonte: O autor (2017)

Este novo sistema, apesar de ser desenvolvido em PHP, **utiliza os serviços de autenticação** que foram criados em Java com o Sistema de Acesso Servidor, nas diretrizes do *Framework*, para permitir que usuários com elevação possam

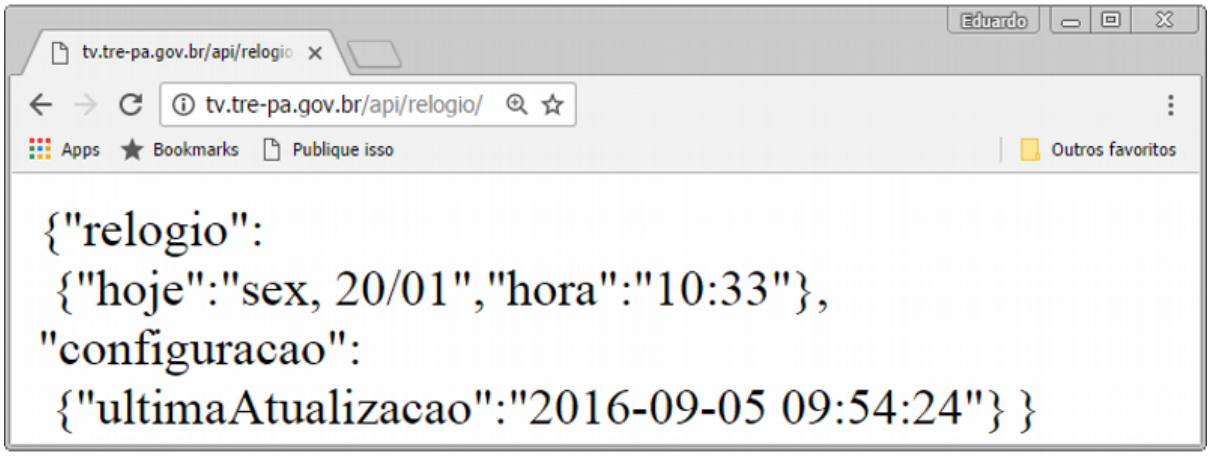
acessar a tela de configuração do sistema que, como mostrado na parte direita na Figura 4, é bem simples, tendo apenas 8 informações modificáveis. Isso demonstra a capacidade do *Framework* de agilizar o desenvolvimento de serviços, aproveitando funções já implementadas nos serviços anteriores.

As funções implementadas exclusivamente para este sistema foram **/slides**, **/carregarConfiguração** e **/salvarConfiguração** (que requer autenticação prévia), todas estas dependendo de dados exclusivos da aplicação.

Também foram desenvolvidos microserviços independentes e compartilháveis nos moldes do *Framework*, que são utilizados na montagem da tela de apresentação da TV, mas que também podem aproveitados em qualquer outro sistema do Tribunal e não requerem autenticação. São estes:

- **/relógio** - a tela de apresentação mostra o dia e hora atuais, porém não extrai esta informação localmente (pois foi desenvolvida primariamente para uso em SmartTVs) já que foi percebido que a informação sempre vinha errada, porque as TVs não tinham bateria interna como os PCs. A solução foi atualizadar pelo servidor de aplicação, conforme a **Figura 4** mostra;
- **/notícias** - usa os parâmetros **/locais**, **/externas** ou **/todas** para receber uma lista de de notícias oriundas do Portal da Intranet e/ou do site UOL Notícias, para deslizarem no rodapé da tela de notícias, alternadamente;
- **/clima** - usa parâmetros como **/cidade**, **/coordenadas** ou **/detectar** (via IP) para receber dados climáticos do **/dia** ou **/semana** para informar na tela. O módulo desenvolvido extrai estes dados de um web-service externo gratuito;
- **/loterias** - usa parâmetros como **/tipo** e **/sorteios** para receber os últimos N sorteios da Mega Sena ou Loto-fácil, para informar junto com as notícias.

Figura 4 - Exemplo de saída do comando /relógio do TRE-TV



Fonte: O autor (2017)

4.3 RESULTADOS OBTIDOS

Os protótipos criados - não com a Plataforma completa, mas com componentes e métodos especificados pelo *Framework* - mesmo que modestos em suas funcionalidades, já podem dar uma ideia das vantagens de se desenvolver uma Plataforma como a oferecida neste projeto, em termos de velocidade de desenvolvimento, velocidade de execução e economia de recursos de rede, memória e processamento.

4.3.1 **Resultados das Implementações Parciais**

Depois de executar cada uma das duas versões do aplicativo **Acesso Servidor** cem vezes, cada uma com limpeza de cache local para que todos os recursos fossem carregados em todas as cem vezes, foram obtidos valores com pouquíssima variação, gerando uma média consistente para as comparações.

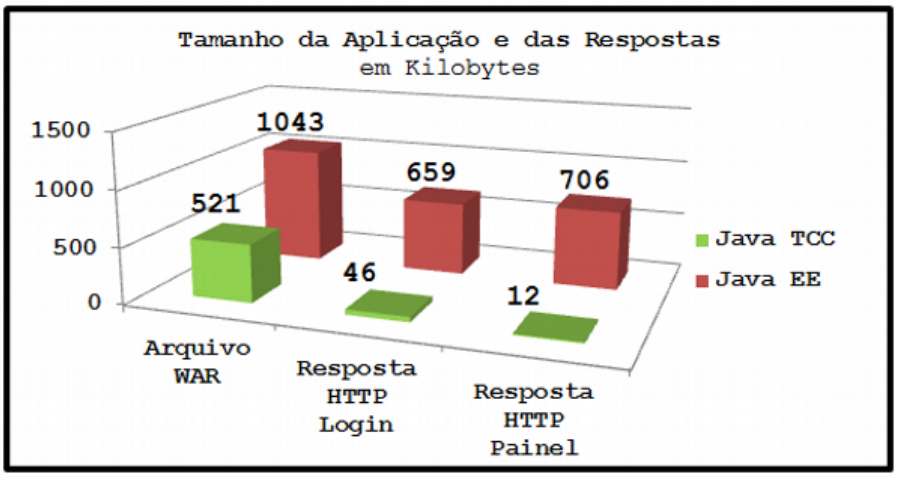
Para obter os resultados, é necessário executar as requisições em um navegador que possua ferramentas para desenvolvedores, bastando pressionar a tecla F12 para ativá-las, e assim utilizar as ferramentas Network e Timeline para acompanhar o processamento das requisições e suas respostas.

A primeira variável de comparação descreve o tamanho total do arquivo de pacote WAR onde fica contida a aplicação Java que deve ser executada no *container* Web. Foi utilizado o *Container* JBoss 7.1 Wild Fly com Java EE 1.6 num servidor Ubuntu 14.4. O JBoss já vem com os *frameworks* Java EE embutidos, o que significa que a versão do Acesso Servidor que utiliza este padrão não foi empacotada com os arquivos JAR destes *frameworks*, caso contrário teriam um tamanho final bem maior.

A segunda variável descreve o tamanho das respostas das requisições entre as duas versões existentes, excluindo desta métrica os arquivos de imagens que fazem parte das páginas carregadas, pois nem todas são idênticas entre as versões.

Os resultado da primeira e segunda variável está demonstrado no **Gráfico 1**, onde é possível identificar que a versão da aplicação que foi baseada no *Framework* (TCC) tem seu tamanho diminuído pela metade (**50%**) em relação a primeira versão que é inteiramente baseada nos *frameworks* Java EE. Identifica-se também que a diferença é ainda maior na quantidade de dados enviada ao cliente pelo servidor, entre uma versão e outra, onde a página de login da nova versão é apenas **7%** do tamanho da antiga, e a página principal é menos de **2%** o tamanho da predecessora.

Gráfico 1 - Tamanho do aplicativo e das respostas HTTP



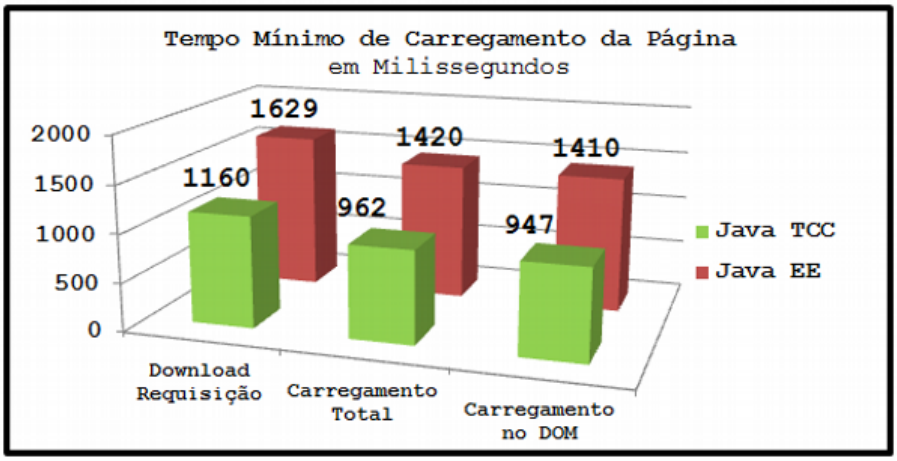
Fonte: O autor (2017)

A terceira variável descreve o tempo mínimo necessário entre o início do recebimento da resposta à uma requisição e o total carregamento de seu conteúdo no navegador do cliente.

O mecanismo dos navegadores funciona de forma a receber o conteúdo do servidor e começar a moldá-lo no Modelo de Objeto de Documentos (DOM) assim que os primeiro sub-objetos estão completos, como imagens, tabelas, scripts, mesmo que ainda não tenha recebido o conteúdo total - ou seja, as duas ações ocorrem paralelamente.

No **Gráfico 2** é possível identificar que o tamanho das requisições foram proporcionais à velocidade com que uma requisição é completada. A diferença de tempo entre as versões é de quase meio segundo nos três parâmetros: *download* da resposta, carregamento na tela e no DOM, sendo melhor a versão do *Framework*.

Gráfico 2 - Tempo mínimo de carregamento da página



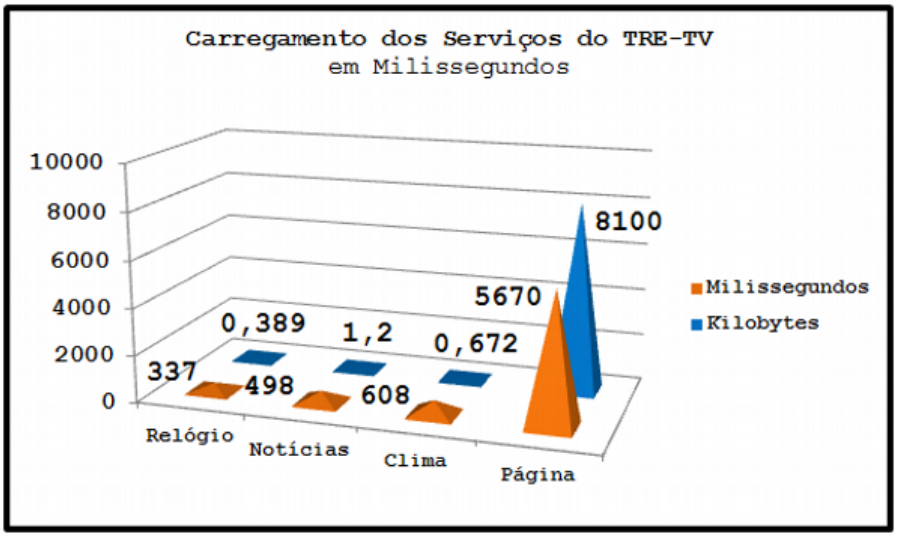
Fonte: O autor (2017)

A aplicação **TRE-TV** não tem referencial de comparação, porem seus resultados não deixam de ser surpreendentes em termos de tamanho e velocidade. Seu código-fonte possui apenas 148 kilobytes e levou apenas duas semanas para ser desenvolvido: uma para os itens da página principal e outra para a configuração.

No **Gráfico 3** é possível identificar o tamanho do resultado e o tempo de resposta da requisição de cada um dos principais comandos desenvolvidos, juntamente com o resultado do carregamento completo da página, que utiliza todos os comandos e ainda carrega scripts e imagens.

Quanto ao tamanho dos dados recebidos e seu tempo de carregamento da página principal, em relação aos serviços carregados através de comandos, estes representam apenas **0,02%** do tamanho e **25%** do tempo totais de carregamento.

Gráfico 3 - Tempo de carregamento dos serviços do TRE-TV



Fonte: O autor (2017)

É interessante enumerar também os resultados dos testes realizados em 2014, onde foram utilizados dois sistemas operacionais em um mesmo provedor remoto de máquias virtuais para o estabelecimento de websites.

Como dito anteriormente, o teste foi realizado com uma estrutura de gerenciamento de requisições em PHP como linguagem, MySQL como banco de dados e Apache para servidor HTTP. Os dois sistemas operacionais utilizados foram o Ubuntu 12 e o Windows Server 2008, ambos recém instalados com uma configuração padrão em máquina virtual de 1 core de 1GHz, 1GB de memória RAM e 20GB de disco Solid State Drive (SSD).

Em todos os testes realizados, as respostas às requisições eram em média **35%** mais rápidas quando executadas no sistema operacional Linux, não importando o tamanho do resultado, que era definido por qual stored procedure era executada.

4.3.2 **Resultados do Desenvolvimento do Projeto do *Framework***

Foram criadas **mais de 400 especificações** para o *Framework*, o suficiente para dar forma e personalidade que lhe confere diferenciais entre as opções de plataformas existentes, que são a baixa curvatura de aprendizagem, a agilidade no desenvolvimento de microserviços imutáveis, que não precisam conter código para lhe dar com interação com usuário, além da real separação em camadas MVC, não somente no código dos sistemas desenvolvidos, mas também nas suas equipes de desenvolvimento.

Só a capacidade de disponibilizar as *stored procedures* de bancos de dados como comandos acessíveis via web-service já fazem deste *Framework* um modelo pioneiro no desenvolvimento de funções *serverless* de desenvolvimento rápido e que podem ser utilizadas por um número indefinido de serviços clientes das plataformas desenvolvidas com estas especificações.

Mas é obvio que para o sucesso desta arquitetura, as outras capacidades da Plataforma são essenciais, como o controle de usuários, sessões, arquivos e serviços, isso sem contar com aqueles serviços de infraestrutura que mantém todo este ecossistema seguro, disponível e equilibrado, como as funções de criptografia, balanceamento de carga e gerência tríade de *hardware*, *software* e rede.

Na **Tabela 1** temos o total de especificações deste *Framework*, divididas em suas áreas de trabalho, que foram citadas no **Capítulo 3**. Não são consideradas as especificações resultantes de configurações salvas em tabelas dos bancos de dados de serviços, mas apenas aquelas diretamente ligadas à Plataforma.

Tabela 1 - Total de especificações do projeto de framework

Tipo de Especificação	Quantidade
Características de Arquitetura e Funções Principais	12
Características de Convenção de Funcionamento	6
Categorias de Serviço Desenhadas como Padrão	8
Categorias de Itens de Configuração	14
Comandos de de Microserviço Desenhados	100
Especificações das Categorias de Serviços	120
Especificações Gerais da Camadas de Servidor, Cliente e Segurança	116
Tabelas de Banco de Dados Desenhadas	34
Total Geral	**410**

Fonte: O autor (2017)

4.4 CONSIDERAÇÕES FINAIS DESTE CAPÍTULO

Não houve tempo para desenvolver um protótipo mais completo e específico de plataforma que atendesse a todas as especificações gerais e das categorias de serviços desenhados nas 54 páginas de desenvolvimento do Projeto. Porém os pequenos resultados obtidos, tanto teóricos quanto práticos já revelam em seus números que as idéias propostas neste trabalho são promissoras, e que a implementação deste projeto pode trazer muitos benefícios para seus usuários como: empresas de qualquer tamanho que utilizarem a Plataforma para sua automação; instituições públicas que quiserem baratear seus custos e acelerar o desenvolvimento além de aumentar a integração entre instituições e esferas governamentais; para os próprios serviços de *backend* na nuvem, que podem ter milhares de usuários criando e utilizando microserviços em conjunto para servir a aplicativos móveis e desktop no mundo todo; e por fim até mesmo os usuários domésticos terão facilidade em desenvolver serviços pessoais ou comunitários, além de sites pessoais, grupos de discussão e blogs.

Um dos objetivos específicos deste projeto é compartilhar este projeto para a comunidade internacional de desenvolvedores, o que foi feito através de um serviços popular de compartilhamento de projetos, o GitHub, que também é um Sistema de Controle de Versionamento (SCV) e permite que projeto seja evoluído sem que sua raiz seja descaracterizada. O endereço web para acesso ao projeto é:

- **https://github.com/eduardoalcantara/ybatinga**

5 **CONCLUSÃO**

Quando se pensa em *frameworks* relacionados à tecnologia da informação, é mais comum imaginar ferramentas para linguagens de programação. Mas também existem *frameworks* para facilitar e padronizar o desenvolvimento de projetos, homologar camadas de segurança e infraestrutura, gerenciar serviços de tecnologia e muitos outros fins indispensáveis para a conjuntura atual.

Ao buscar as fundamentações e conceitos que nortearam este trabalho, verificou-se que as linguagens de programação e seus utilizadores dependem cada vez mais de ferramentas de automação e *frameworks* de padronização que façam a maior parte do trabalho complexo e repetitivo no ato de desenvolver sistemas. Neste contexto, o *framework* age como um mero remédio para o déficit funcional de uma linguagem de programação, que por natureza não possui previsão ou suporte para determinado tipo de utilização ou capacidade de desenvolvimento de um software para um determinado fim que não era contemplado no momento da criação destas linguagens – o que fica claro se for considerado, por exemplo, que nem os protocolos HTTP/HTTPS e nem a linguagem HTML originais - que são o contêiner básico da Web - foram desenvolvidas com o objetivo de suportar aplicações distribuídas complexas e dinâmicas. Seu papel era meramente transportar e exibir informações estáticas navegáveis através de *links*, em servidores remotos.

Neste trabalho desenvolveu-se um *framework* como uma ferramenta de modelagem de uma plataforma de microserviços distribuídos, cuja estrutura pode ser construída em qualquer linguagem de programação moderna. Ou seja, um *framework* não para uma linguagem, mas para o resultado final obtido ao utilizar, também, uma linguagem de programação.

Identifica-se então que este tem a pretensão de ser um *framework*, para a camada de serviço, que tem como missão tornar desnecessário o uso de múltiplos *frameworks* na camada de codificação dos sistemas, tornando-os mais rápidos e simples de serem desenvolvidos, sem interferir em sua eficácia e objetivos únicos.

De acordo com a contextualização teórica deste projeto, pode-se concluir que mesmo sendo um trabalho restrito - no que diz respeito à verificação de sua eficácia, mas conhecendo as tecnologias aplicadas a este projeto - é possível alcançar seus objetivos caso proceda-se sua implementação.

É dado como cumprido o objetivo geral de desenvolver o Projeto de *Framework* para Plataformas, produzindo mais de quatrocentas especificações para sua implementação, assim como os objetivos específicos, nos quais foram projetadas todas as camadas de funcionamento e serviços necessários ao seu pleno funcionamento, atendendo as especificações de segurança, flexibilidade e facilidade

de aprendizado, e assim como foram apresentados cenários de aplicação e utilização no mercado, também foi totalmente disponibilizado em repositório público de projetos para apreciação e contribuição pela comunidade interessada.

Através do desenvolvimento de aplicativos que seguiram padrões do *Framework* como: desacoplamento de APIs em comandos, unificação de controle de acesso de usuários e substituição do uso *frameworks* padrões pela divisão real de camadas de aplicação - foi possível determinar a aplicabilidade dos resultados esperados para as plataformas desenvolvidas nas bases do *Framework* projetado, com expectativas e projeções realistas para comparação com os paradigmas atuais.

Além de elaborar uma nova proposta que sirva de alternativa para a administração e desenvolvimento de serviços remotos e redes distribuídas, com este trabalho também foi possível:

- Compreender os mecanismos necessários ao tratamento de requisições remotas, seu processamento e resposta ao cliente;
- Elucidar sobre a arquitetura de microserviços distribuídos e suas vantagens tanto para desenvolvedores quanto para consumidores de serviços na rede;
- Desenvolver o conceito e a implementação de funções *serverless,* além de entender seu futuro como principal ativo dos serviços de *back-end* remotos;
- Ratificar a importância da camada de segurança e do controle de usuários e suas permissões para a estrutura de serviços através da Internet;
- Envolver os vários temas chave do curso de Sistemas de Informação para o desenvolvimento deste trabalho, como Redes, Programação, Banco de Dados, Segurança, Administração da Informática e Engenharia de Software;
- Abrir as portas para a implementação do projeto por quaisquer interessados nos benefícios deste *Framework*, assim como a oportunidade de melhora contínua e exponencial de seu potencial através da colaboração paralela e universal possibilitada pelo compartilhamento em código aberto.

A palavra *ybatinga* significa nuvem (a conotação atual da Internet) em tupi-guarani e não foi a escolha inicial, nem a mais aceita para nomear este *framework*, porém é uma homenagem ao povo da região norte do Brasil e ao potencial que tem para contribuir com ideias sustentáveis.

A necessidade de revisão dos paradigmas do desenvolvimento de software na nuvem é uma realidade, seja por fatores humanos, sociais, técnicos, ambientais ou financeiros. Este trabalho oferece uma nova interpretação para o funcionamento de bases da estrutura atual de serviços remotos, com a confiança de que seu conteúdo se tornará uma ferramenta promissora e geradora de novas tendências em sua área de atuação, seja no âmbito acadêmico, seja no âmbito comercial.

REFERÊNCIAS

ALURA. Curso de Java e Orientação a Objetos. **Caelum**. São Paulo, 2016. Disponível em <https://goo.gl/co4SnM>. Acesso em: 09 Set 2016.

_____. Java para Desenvolvimento Web. **Caelum**. São Paulo, 2016. Disponível em <https://goo.gl/0swlst>. Acesso em: 10 Set 2016.

_____. Laboratório Java com Testes, JSF e Design Patterns. **Caelum**. São Paulo, 2016. Disponível em <https://goo.gl/k0Rf19>. Acesso em: 10 Set 2016.

_____. **SOA na prática**: Integração com Web Services e Mensageria. Curso Intensivo em Belém sob demanda do TRE-PA: Caelum, v. FJ-36, 2015.

AQUINO JR, Gibeon S de. **Desenvolvimento de Sistemas Web em Java**: Frameworks, Padrões de Projeto e Diretrizes para a Camada de Apresentação. Recife, 2002. 129 p. Disertação (Pós Ciência da Computação)-UFPE

BORKAR, Shekhar; CHIEN, Andrew A. **The Future of Microprocessors**. Hillsboro: ACM, 2011. Disponível em <https://goo.gl/QMSF0S>. Acesso em: 10 Set 2016.

BRUCE A, Tate. **Beyond Java**. Sebastopol: O'Reilly, 2005. 185 p.

BYRNE, David. Is the Information Technology Revolution Over?. **Federal Reserve**. Washington, 2013. Disponível em <https://goo.gl/pYj6bf>. Acesso em: 06 Set 2016.

DE ROSA, Aurélio. Learning Curve: What it is and How it Applies to Information Technology. **Web Developer Blog**. London, 2013. Disponível em <https://goo.gl/GiRVA6>. Acesso em: 10 Set 2016.

DEITEL, Paul e Harvey. **JAVA**: Como Programar. Tradução Carlos Arthur Lang Lisboa. 8. ed. Porto Alegre: Prentice Hall, 2010. 1152 p.

DMITRUK, Hilda Beatriz(Org). **Cadernos metodológicos**: diretrizes da metodologia científica. 5. ed. Chapecó: Argos, 2001. 123 p.

FLANAGAN, David. **JavaScript**. 6. ed. Porto Alegre: Bookman, 2013. 1062 p.

FORRISTAL, Jeff. **Site Seguro e Aplicações Web**. Rio de Janeiro: Alta Books, 2002. 490 p.

GONÇALVEZ, Edson. **Desenvolvendo Aplicações Java com JSP, Servlets, JSF, Hibernate, EJB 3 Persistence e Ajax**. São Paulo: Ciência Moderna, 2007. 776 p.

GUPTA, Arun. Monolithic to Microservices: Refactoring for Java EE Applications. **Miles to go 3.0**. São Francisco, 2015. Disponível em <https://goo.gl/TQXq46>. Acesso em: 20 Nov 2016.

IBM. **Who uses mainframes and why do they do it?**. 2010. Disponível em <https://goo.gl/jg5uqq>. Acesso em: 08 Jan 2017.

KELLY, Martin CAMPBELL. **From Airline Reservations to Sonic the Hedgehog**: A History of the Software Industry. Cambridge: The MIT Press, 2003. 372 p.

KUROSE, Jim; ROSS, Keith. **Computer Network**: A Top Down Approach. 6. ed. Boston: Addison-Wesley, v. Slides, 2012.

LOBO, Edson. **Curso de Engenharia de Software**. São Paulo: Digerati, 2008.

LUCKOW, Décio Heinzelmann; MELO, Alexandre Altair de. **Programação Java para a Web**. 2. ed. São Paulo: Novatec, 2015. 640 p.

MACVITTIE, Lori. Braceyourselves. Serverless is coming. **F5 Networks**. Seattle, 06/2016. Disponível em <https://goo.gl/foZ6ZA>. Acesso em: 09 Set 2016.

_____. Cookies, Sessions, and Persistence. **F5 Networks**. Seattle, 07/2008. Disponível em <https://goo.gl/bsO6Rr>. Acesso em: 09 Set 2016.

MALAVASI, Eike. Orquestração e Coreografia em SOA. **Sensedia**. Campinas, 2008. Disponível em <https://goo.gl/rE5Cm8>. Acesso em: 12 Jan 2017.

MASSOL, Vincent; ZYL, Jason van. **Better Builds with Maven**: Um guia para Maven 2.0. South California: Mergere Library Press, 2007. 301 p.

MELO, Ana Cristina Vieira de; SILVA, Flávio Soares Corrêa da. **Princípios de**

Linguagens de Programação. São Paulo: EdgardBlücher Ltda, 2003. 211 p.

NORTON, Peter; AITKEN, Peter; WILTON, Richard. **Peter Norton: A Bíblia do Programador**. Tradução Geraldo Costa Filho. 3. ed. Rio de Janeiro: Campus, 1993. 640 p. Tradução de: PC Programmer's Bible.

NOYES, Katherine. 10 Reasons Open Source Is Good for Business. **PC World**. San Francisco, 2010. Disponível em <https://goo.gl/900PeS>. Acesso em: 09 Set 2016.

O'BRIEN, James A. **Sistemas de Informação**. São Paulo: Saraiva, 2011. 430 p.

OLIVEIRA, Valéria. **Desmistificando a pesquisa científica**. Belém: EDUFPA, 2008.

PRESSMAN, Roger. **Engenharia de Software**. São Paulo: Makron, 1995. 1056 p.

RICHARDSON, Chris. Eventuate Platform: Solving DDM problems in a microservice architecture. **Eventuate.IO**. 2016. Disponível em <http://eventuate.io/>. Acesso em: 09 Jan 2017.

_____. Pattern: Microservices Architecture. **Microservices.IO**. 2014. Disponível em <https://goo.gl/idyDAG>. Acesso em: 09 Jan 2017.

RIEHLE, Dirk. **Framework Design**: A Role Modeling Approach. Zürich, 2000. 230 p. Tese (Doutorado em Ciências Técnicas)-Universität Hamburg

SARKUNI, Sehrope. How I Write SQL: Naming Conventions. **Launch by Lunch**. Cranbury, 2014. Disponível em <https://goo.gl/4oAI5K>. Acesso em: 02 Jun 2016.

SESHADRI, Shyam; GREEN, Brad. **Desenvolvendo com AngularJS**: Aumento de produtividade com aplicações Web estruturadas. São Paulo: Novatec, 2014. 349 p.

SILVA, Maurício Samy. **Ajax com jQuery**: Requisições Ajax com a simplicidade de jQuery. São Paulo: Novatec, 2009. 327 p.

SMITH, Roger. Why Java is the nº 1 programming language. **The Server Side**. Newton, 2016. Disponível em <https://goo.gl/0XAgNS>. Acesso em: 20 Nov 2016.

SOARES, Sérgio. **Des. Progressivo de Programas Concorrentes Orientados a Objetos**. Recife, 2001. 121 p. Disertação (Ciência da Computação)-UFPE

SOMMERVILLE, Ian. **Software Engineering**. Harlow: Addison-W, 2011. 792 p.

TANENBAUM, Andrew S.; STEEN, Maarten van. **Sistemas Distribuídos**: Princípios e Paradigmas. 2. ed. São Paulo: Pearson, 2007. 416 p.

TANENBAUM, Andrew S.; WETHERALL, David. **Redes de Computadores**. 5. ed. São Paulo: Pearson, 2011. 600 p.

TANENBAUM, Andrew S; WOODHULL, Albert S. **Operating Systems Design and Implementation**. 3. ed. Londres: Pearson, 2008. 1080 p.

TCU. **Levantamento acerca da Governança de Tecnologia da Informação na Administração Pública Federal**. Brasília, 2008. 48 p.

TERRA. Multas de trânsito serão integradas nos 27 Estados. **Terra Notícias**. 2007. Disponível em <https://goo.gl/q6Tg3c>. Acesso em: 18 Jan 2017.

W3TECHS. Usage of web servers for websites. **W3Techs**. Canadá, 2017. Disponível em <https://goo.gl/g7pNmD>. Acesso em: 11 Jan 2017.

WIKIPÉDIA. **Apache Maven**. 2016. Disponível em <https://goo.gl/9mqGjj>. Acesso em: 11 Out 2016.

_____. **Comparison of web server software**. 2017. Disponível em <https://goo.gl/dFOiu4>. Acesso em: 10 Jan 2017.

_____. **Java EE Version History**. 2016. Disponível em <https://goo.gl/iyOOxN>. Acesso em: 06 Out 2016.

_____. **Microservices**. 2016. Disponível em <https://goo.gl/E3puJk>. Acesso em: 09 Jan 2017.

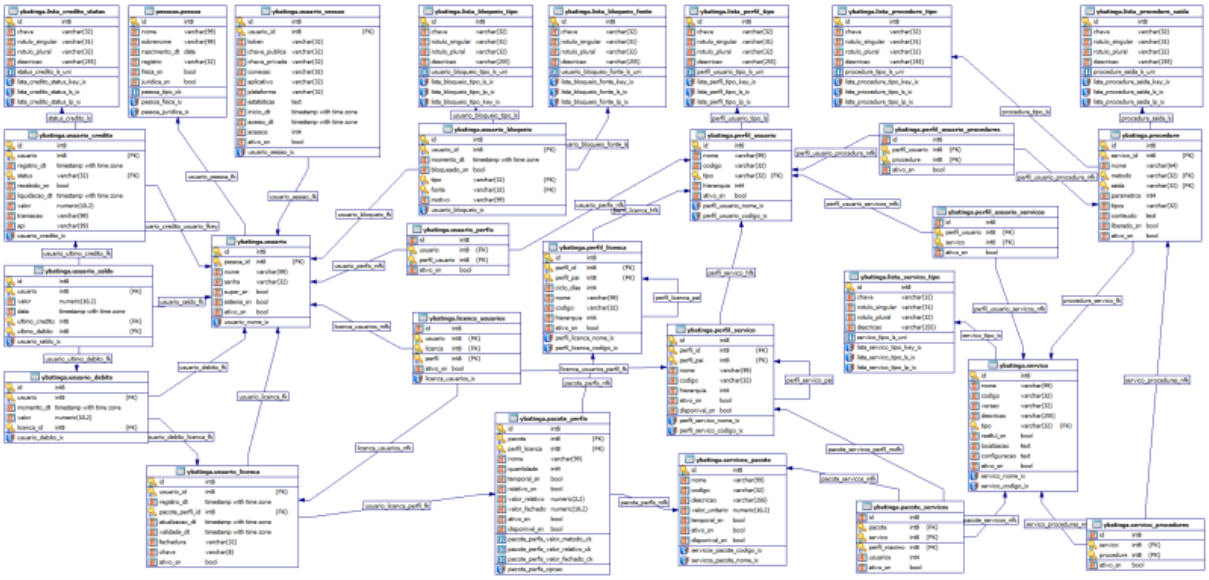
_____. **URL**. 2016. Disponível em <https://goo.gl/atdtka>. Acesso em: 14 Dez 2016.

APÊNDICE A — Diagrama Visual do Banco de Dados da Plataforma

A Figura neste apêndice apresenta um diagrama contendo as principais tabelas do banco da dados da Plataforma, com suas colunas, índices e conexões de chaves estrangeiras.

O sistema de banco de dados utilizado foi o PostgreSQL 9.5. Já a modelagem visual foi feita usando o MicroOLAP Database Designer for PostgreSQL v1.2.3, em sua versão de avaliação.

Este diagrama é melhor visualizado numa área impressa de 30 centímetros de altura por 65 centímetros de largura, e por isso foi reimpresso numa folha de papel especial e anexado na próxima página. Caso ele não esteja presente, é possível visualizar o arquivo original (assim como as demais figuras neste trabalho) através do endereço eletrônico **https://goo.gl/X80f2H**



APÊNDICE B — Padrões de Modelagem e Convenções para Bancos de Dados

Padrões de Modelagem de Estrutura

Regras e convenções de modelagem devem ser seguidas para evitar inconsistência e facilitar o entendimento do modelo por outros profissionais. Algumas das convenções utilizadas aqui foram baseadas no trabalho de Sarkuni (2014).

Quadro - Padrões de modelagem e convenções no bancos de dados

Regra ou Convenção	Exemplo Correto	Exemplo Incorreto
Usar caixa alta para identificadores reservados da SQL	CREATE TABLE	select * from
Usar caixa baixa com *underline* para identificadores do banco de dados (*snake_case*)	nome_completo	PESSOA_ID, SobreNome
Não colocar tipo de dado como parte do nome da coluna	telefone_fixo	idade_int, endereço_texto
Não colocar o tipo de objeto como parte do nome do objeto, exceto em índices e chaves	idx_pessoas, fk_parametros	fn_verifica_cpf
Usar o singular para nomes de objetos, exceto em numeral, blob, json ou tabelas do tipo N:M	pessoa (tabela), filhos (coluna)	telefones (coluna)
Se uma tabela precisar de nome no plural (tabela N:M), usar um alias no comando select	FROM docs AS documento	
Não usar abreviações	data_nascimento	data_nasc
Chave primária de uma tabela deve ter _id no final	usuario_id	usuario_inc
Chaves estrangeiras devem ser a combinação do nome das tabelas e sua id	team_member_fk (team_id, person_id)	members_from_team
Colocar *underline* antes dos nomes de parâmetros de *functions* e procedures para não confundir com nomes de colunas	_telefone	telefone

Fonte: O autor (2017)

Padrões de Nomenclatura de Dados Comuns no Banco

Devemos seguir um padrão consistente em todos os esquemas relacionados ao banco de dados da plataforma, para facilitar a identificação da vocação de cada objeto. Muitos objetos aparecem mais de uma vez no banco, como datas, documentos, apontadores, descritores… e em todos os casos eles devem ter a mesma estrutura de nomenclatura, para facilitar a manutenção de código existente e criação de novo código ou novos objetos seguindo a mesma base.

Quadro - Padrões de nomenclatura da dados comuns no banco

Objeto	**Padrão**
id	chave primária de todas as tabelas: para evitar ambiguidade, todas as menções a esta coluna devem conter o nome de sua tabela
rotulo	uma descrição pessoal acerca da informação daquele registro na tabela, que a separa dos demais registros do mesmo dono
tipo	poderia ser um ENUM categorizando uma informação, ou uma lista de opções em uma tabela
***_id**	todas as chaves estrangeiras são [nome da tabela estrangeira + “_id”]
***_dt**	datas são exceção à regra de abreviação e tipo de dado sem prejuízo do entendimento
***_ls**	uma coluna que aponta um dado que foi selecionado de uma tabela de listagem (enum), sem chave estrangeira
***_sn**	uma coluna com valor binário (booleano) de true/false ou 0/1 ou sim/não
***_js**	uma coluna com valor em notação (json) de dados javascript
***_url**	uma url, quando a própria é apenas uma característica (não principal) do registro
***_pai**	referencia um id de outro registro na mesma tabela, fazendo papel de pai do registro atual, e usa o [nome da tabela + “_pai”]

Fonte: O autor (2017)

APÊNDICE C — *Frameworks* da Plataforma Java EE para a Web

***Frameworks* de conectividade**

Pelo fato da Web ser basicamente um repositório de informações e serviços remotos, a capacidade de conectividade é um fator crucial no desenvolvimento de sistemas para este ambiente.

Existem *frameworks* que tratam das conexões, do seu nível mais básico (protocolos de conexão e monitoramento de requisições) até seu nível mais sofisticado (sessões, autorizações, redirecionamento, atribuição e agendamento de tarefas, balanceamento de carga). Cada *framework* que trabalha na camada mais alta foi construído como tradutor e facilitador de camada imediatamente inferior, o que os torna muito acoplados e dependentes entre si.

Para este tipo de *framework*, pode-se citar: **Servlets**, que é uma especificação Java EE que trabalha recebendo requisições HTTP de clientes remotos e enviando estas requisições para o código customizado do programa - que irá tratar esta requisição e posteriormente enviar uma resposta ao cliente requisitante (AQUINO JR, 2002) - e **Java Database Connectivity** (JDBC), uma interface que serve tanto ao Java EE como a sua versão padrão. Ela define como um cliente Java pode acessar um banco de dados, que antes era apenas compatível com bancos de dados relacionais, mas hoje já suporta tecnologias como NoSQL.

***Frameworks* de persistência**

Com a popularização do Java em ambientes corporativos, logo se percebeu que grande parte do tempo do desenvolvedor é gasto na codificação de queries SQL e no respectivo código JDBC responsável por trabalhar com elas (CAELUM, 2016).

Para este tipo de *framework*, pode-se citar: **Java Persistence API** (JPA), que é um conjunto de bibliotecas e padrões de Java para permitir o mapeamento relacional entre classes Java e objetos de dados a serem persistidos. Ou seja, uma forma de manter suas as informações, como por exemplo, num banco de dados. Esta persistência também pode ser temporária, como por exemplo, numa sessão de usuário ao utilizar uma aplicação Web (GONÇALVEZ, 2007, com adaptações).

Também cita-se o **Hibernate**, que é uma ferramenta de mapeamento objeto-relacional (ORM) que abstrai o banco de dados (suas tabelas e seu código SQL) em forma de comandos inseridos no *framework*, com orientação a objetos, e o **Enterprise JavaBeans** (EJB), que é uma especificação bastante complexa de Java, pois sua principal função é receber, trocar e entregar informações reunidas em um só objeto, acerca de um só assunto, que pode ser uma tabela de banco de dados, ou uma página Web sendo mostrada para o cliente naquele momento.

***Frameworks* de interface**

O uso da arquitetura Web como tecnologia para levar aplicações remotamente aos clientes exigiu uma abordagem diferente da qual a Web foi criada. Ao invés de conteúdo estático, que era depois formatado pelo navegador que interpreta o HTML, se fez necessário que essas aplicações entreguem conteúdo dinâmico. Isso significa que, ao digitar um endereço Web, ou apertar um botão ou clicar em um link, o servidor Web poderá mandar uma informação diferente de acordo com uma infinidade de variáveis que podem interferir no sistema, desde o conteúdo de um banco de dados e o usuário que está acessando o sistema, até coisas simples como o mês, a hora atual ou preferências pessoais de cor e layout.

Alguns *frameworks* de interface têm caminhado para atingir o objetivo de facilitar a entrega de conteúdo dinâmico, bem como outros têm tentado mimetizar o comportamento de aplicações feitas para desktop, na medida em que inserem componentes na tela, os quais interagem com o servidor, independentemente da requisição ativa do usuário.

Para este tipo de *framework*, pode-se citar: **JavaServer Pages** (JSP), uma especificação Java EE capaz de gerar conteúdo dinâmico como HTML, DHTML, XHTML e XML e funciona com um documento com extensão JSP cujo conteúdo é um misto de uma das linguagens de marcação citadas acima, adicionada de scripts de comando em Java, que são executados no lado do servidor, e geram o conteúdo totalmente na linguagem de marcação antes de enviar este para o cliente (AQUINO JR, 2002). Também cita-se, devido sua atual importância, o **JavaServer Faces** (JSF), que é uma especificação Java EE capaz de gerar interfaces de usuário através do uso de componentes, que podem ser adicionados na página e ter uma funcionalidade e programação similar aos aplicativos desktop desenvolvidos com componentes e orientação a objetos (ALURA, 2016). Ao invés de utilizar HTML com código Java, este framework é capaz de separar as duas camadas, utilizando

apenas referências aos objetos Java, dentro dos arquivos JSF, que são codificados em XML próprio deste framework, chamado de Facelets. O resultado deste modelo é convertido em Servlets em tempo de execução, que depois é convertido em código HTML e enviado ao cliente, que remonta toda a página, ou pode atualizar apenas parte desta, através da tecnologia AJAX, em componentes compatíveis.

As bibliotecas de componentes padrões do JavaServer Faces podem ser substituídas ou complementadas com componentes de terceiros, dos quais se destacam os pacotes **RichFaces** e **PrimeFaces** pois, além de trazer muitos outros componentes mais modernos e complexos, compatíveis com os layouts da Web 2.0, oferecem “temáticas” completas para criação de aplicativos mais elegantes, do ponto de vista da interface com o usuário. Sua utilização pode ser em conjunta, e não impede a utilização de componentes padrões JSF e nem impede a criação de componentes customizados pelo próprio programador.

Frameworks multitarefa

Muitas vezes um *framework* não se restringe a oferecer serviços para apenas uma camada da aplicação, ou mesmo nem é desenvolvido com este propósito. Como um *framework* é também uma proposta de padronização de projeto, desenvolvimento e arquitetura, ele também pode se estender e influenciar toda a aplicação. Neste contexto existem os *frameworks* multicamada, de arquitetura e outras ferramentas que podem englobar o projeto inteiro.

Para este tipo de *framework*, pode-se citar: **Spring MVC**, que se propõe a separar o desenvolvimento de um sistema em três camadas que dão nome a esta arquitetura: Modelo, Visão e Controle. O modelo é responsável pela lógica de negócio, representada pela forma como as informações são organizadas num banco de dados. A visão é responsável pela interface com o usuário, exibindo de forma orientada as informações contidas no modelo. O controle é responsável pela lógica computacional do sistema, respondendo às requisições do usuário, coletando as informações solicitadas dentro da camada de modelo, e organizando-as dentro da camada de visão (LUCKOW; MELO, 2015, com adaptações).

É de se esperar que, ao utilizar tantos *frameworks*, APIs e bibliotecas num projeto de sistema, as coisas comecem a ficar confusas o projetista e seus programadores, tendo que gerenciar tantas tecnologias aplicadas simultaneamente em cada classe Java criada. Estas classes acabam ficando extremamente dependentes de códigos,

frameworks, bibliotecas e empresas de terceiros. Isto se chama forte acoplamento e tem sido problema para programadores por muitas décadas. O *framework* **Spring IoC** oferece duas propostas de arquitetura que resolvem este problema e ainda permitem que as classes utilizem serviços de outros frameworks, mas de forma genérica, sem depender especificamente de nenhum deles: são a Inversão de Controle e Injeção de Dependência (LUCKOW; MELO, 2015). Na **Inversão de Controle** (IoC), a classe não é mais responsável por criar objetos dentro dela, ao invés disso, ele é passado a ela como parâmetro pela função que a chama e o framework é responsável por saber qual objeto deve entregar toda vez que um parâmetros com inversão de controle for solicitado sem que o mesmo tenha sido criado, e o cria. Isto é a **Injeção de Dependência** (DI).

Frameworks e ferramentas de apoio ao desenvolvedor

Para ajudar o desenvolvedor a ter seu código sempre organizado e gerenciável, aumentar sua produtividade e garantir que suas ferramentas e *frameworks* estejam sempre atualizados, são necessárias medidas automatizadas dependendo do tamanho do projeto, que podem levar à necessidade de mais ferramentas e *frameworks* especialistas.

O **Eclipse**, que é um Ambiente de Desenvolvimento Integrado (IDE) e reúne características e ferramentas de apoio ao desenvolvimento de software com o objetivo de agilizar o processo. Foi desenvolvido em Java, mas é preparado para qualquer linguagem de programação e sua característica mais expressiva é a arquitetura de plug-ins, que permite sua aprimoramento por milhares contribuintes no mundo todo. Esta ferramenta é categorizada como RAD, que significa em Desenvolvimento Rápido de Aplicativos, em português (LUCKOW e MELO, 2015).

Seu principal *plugin* de apoio ao desenvolvimento Java é o **Maven**, uma ferramenta que proporciona automação de compilação e obtenção de recursos de terceiros. O Maven utiliza um arquivo XML para descrever o projeto de software sendo construído, suas dependências sobre módulos e componentes externos, a ordem de compilação, diretórios e *plugins* necessários. Ele vem com objetivos pré-definidos para realizar certas tarefas bem definidas como compilação de código e seu empacotamento. Seu principal benefício é permitir que todos os *frameworks* necessários a um projeto sejam baixados automaticamente e integrados de forma automática no código/projeto na máquina local do desenvolvedor, assim como atualizar aqueles que tenham versões mais atuais que as instaladas localmente.

Demoiselle é um *framework* desenvolvido pelo SERPRO (Serviço Federal de Processamento de Dados) em 2008, com o objetivo de ser uma solução de padronização de desenvolvimento que pode, e é usada, por muitas instituições governamentais. De acordo com o site oficial (frameworkdemoiselle.gov.br), este *framework* tem o objetivo de facilitar e normatizar o desenvolvimento de aplicativos de grande porte, dando uma estrutura básica, indicando quais tecnologias devem ser utilizadas, indicando padrões de implementação em camadas e ajudando nas decisões de projeto. Sua filosofia é de que a aplicação deve ser totalmente independente das camadas de interface e de banco de dados. Apesar de ser uma excelente ferramenta que reúne muitas vantagens de outros grandes *frameworks* (o que dermite o descarte destes no projeto), sua manutenção é muito lenta, pois a comunidade de desenvolvimento é reduzida, e por isso ainda não foi disponibilizada uma versão compatível com as duas últimas versões da plataforma Java EE.

Sistema de Controle de Versionamento

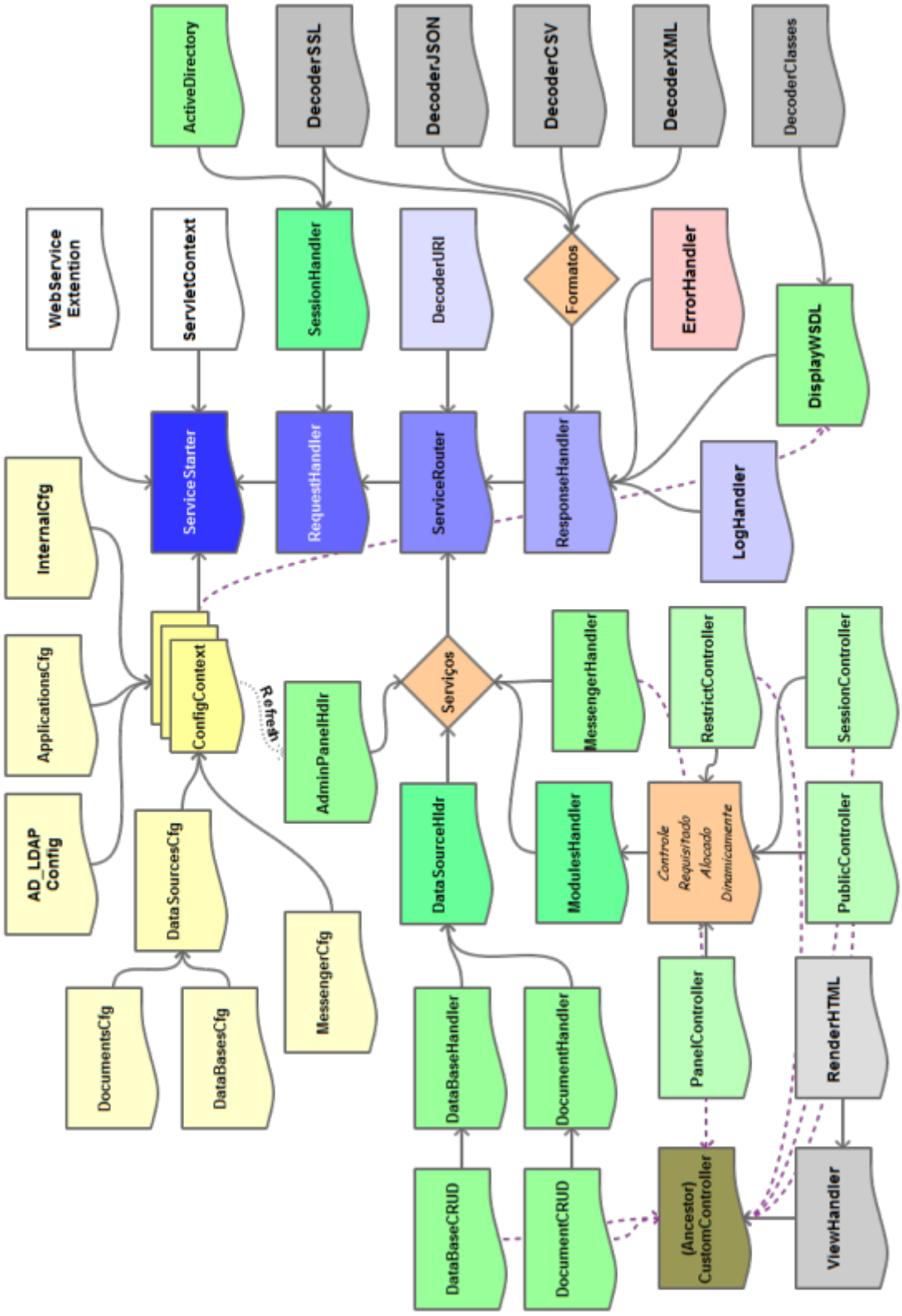
SCV é uma ferramenta de apoio ao desenvolvedor que permite gerenciar as versões de todos os documentos relacionados a um projeto de sistema, como código fonte, multimídia, estrutura de banco de dados e guarda informações que permitem restaurar versões anteriores ou apenas consultar quais alterações foram feitas de uma versão para outra.

Este tipo de ferramenta permite que vários desenvolvedores possam atuar em um mesmo projeto com o menor risco de sobreposição de código e injeção de código malicioso, pois permite um controle conciso de alterações e responsabilidades, inclusive com esquema de supervisão e liberação de alterações para usuários com este tipo de permissão.

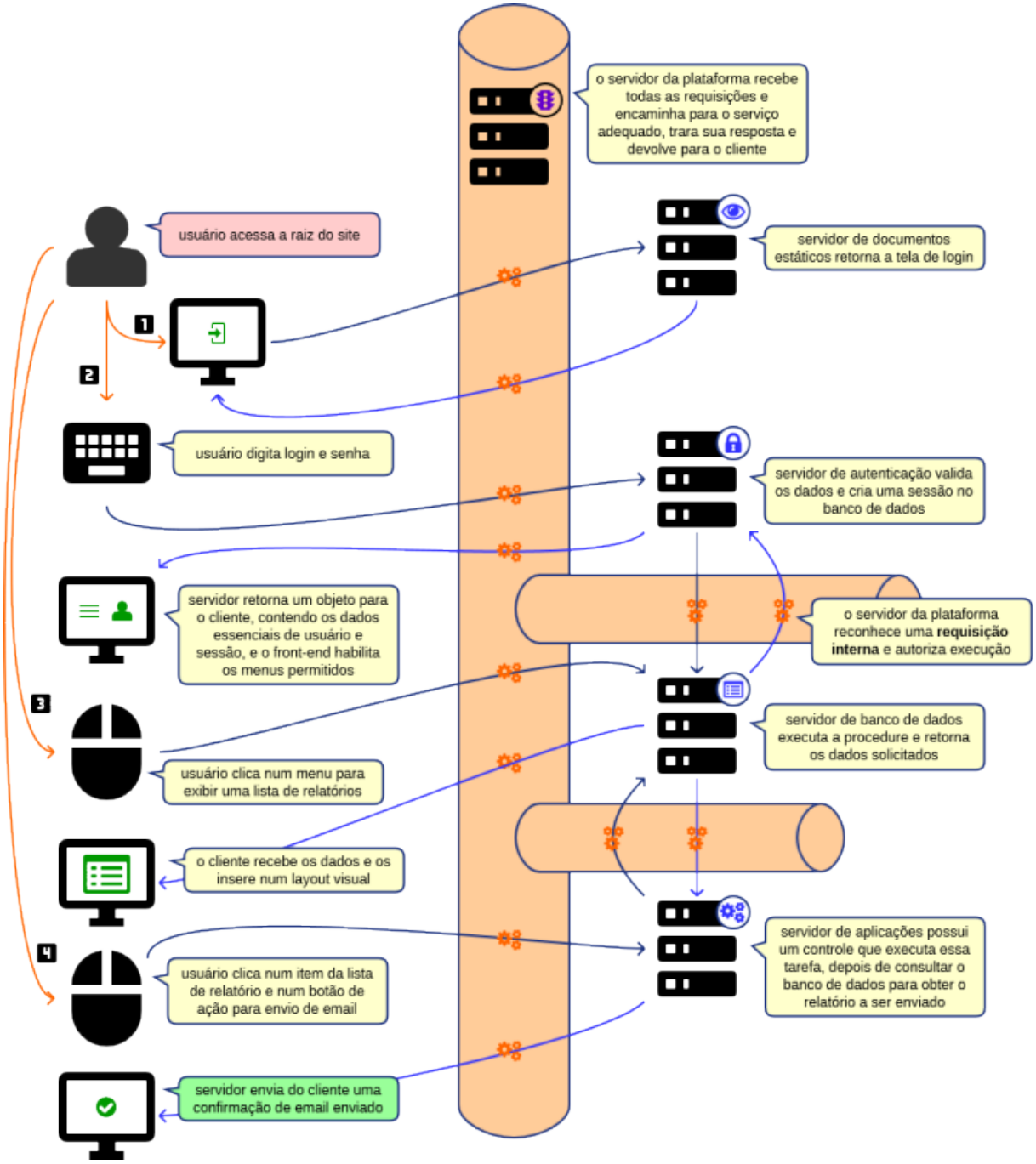
Existem no mercado várias soluções de código aberto e gratuitas, tanto para instalação no computador de trabalho como para funcionamento online em servidores centrais ou distribuídos, assim como também existem *plugins* que permitem usar estas ferramentas diretamente na IDE Eclipse.

APÊNDICE D — Diagramas do Gestor de Requisições (Fluxos e Classes)

O diagrama abaixo ilustra a modelagem de classes de um Gerente de Requisições implementado em Java. Disponível em **https://goo.gl/X80f2H**



O diagrama abaixo ilustra visualmente um exemplo interação entre usuário e servidores da Plataforma, mostrando o fluxo de requisições.



APÊNDICE E — Códigos em JavaScript para a Camada Cliente

Funções de Acesso ao Servidor da Plataforma para Requisições JS:

```
var action = function($scope){
    var nextRequest: {};
    // gatilhos para sucesso ou falha
    $.ajaxSetup({
        "error": function(result){
            this.fail = true;
            $scope.$callbackError(result);
        },
        "success": function(result){
            response = result;
            fail = false;
            $scope.requests[].add(
                response,
                now(),
                fail
            );
        }
    });
    // função geral de requisições http
    request: function($urn, $post, $dataType, $func){
        if ($scope.target.backupMode){
            requestJSONP($urn+'&'+serialize($post));
        }else{
            this.nextRequest.urn = $urn;
            this.nextRequest.post = $post;
            this.nextRequest.xhr = new XMLHttpRequest();
            this.nextRequest.xhr.open('POST',
                $scope.target.platformURL, true);
            this.nextRequest.xhr
            .setRequestHeader('Content-type', $dataType);
            this.nextRequest.xhr.onload = $func;
        }
    },
    // função para adicionar parâmetros à requisição
    param: function($p) {
        this.nextRequest.urn += $p;
    },
    // função para enviar a requisição à Plataforma
    send: function(){
        var data = new FormData();
        data.append('post',this.nextRequest.post);
        data.append('urn',this.nextRequest.urn);
        this.nextRequest.xhr.send(data);
    },
    requestJSONP: function($urn){
        script = document.createElement('script');
        script.type = 'text/javascript';
        script.src = $scope.target.backupURL + $urn
                    + '&callback=responseJSONP';
    },
    responseJSONP: function(result){
        this.response = result;
    },
    response, fail, getWSDL,
    getForm, putForm,
    getAccess, getSession,
    getObject, putObject},
```

O cabeçalho do objeto **$platformScope,** que serve para guardar o estado das requisições e outras informações dentro da camada do cliente em JS:

```
var $platformScope = {
    application = {ID, type, profiles[]},
    requests[] = {URN, $objectSent, $objectReceived, _addRequest},
    security = {$encryptMode, $key: {platformPublicKey, clientKey}},
    session = {$id, $token, type, lastCheck, expiration},
    target = {pID, platformURL, backupURL[], wsdl: {}},
    user = {id, $key, info: {name, email, type, profiles[]:{}}},
    action = {request, response, fail, getWSDL, putForm,
              getAccess, getSession, getObject, putObject},
    initialize = function(pURL) {
        this.target.platformURL = pURL + '/';
        this.target.wsdl = this.action.getWSDL();
    },
};
```

Observação: *não foi considerado o uso de conversões de tipos de dados, nem o uso de criptografia nestes exemplos de código.*

APÊNDICE F — Vantagens e Desvantagens das Linguagens na Camada Cliente

As linguagens de programação que interessam ao projeto do Framework são aquelas que já possuem formas nativas para acesso remoto através destes três protocolos, sendo que algumas linguagens possuem ainda bibliotecas criadas pelas suas comunidades de usuários, que facilitam ainda mais o trabalho com estes protocolos, mesmo sem a necessidade de frameworks mais complexos para este fim. O quadro abaixo cita as vantagens e desvantagens nas linguagens mais usadas:

Quadro - Linguagens de programação para a camada cliente

Linguagem	Vantagens	Desvantagens
JavaScript	Funciona em qualquer OS com um interpretador, até em smartphones está sendo utilizada para desenvolver aplicativos	Linguagem interpretada; fracamente tipada; sem segurança; sem um bom *debugger* para erros
Java	*Bitcode* multiplataforma; já é muito difundida no desenvolvimento Web e Desktop;	Precisa da plataforma (*frameworks*) Java EE para a Web; o código gerado é muito extenso sem uso de *frameworks*
PHP	A mais popular em servidores Web; muito dinâmica por ser interpretada em tempo de execução	Muitas falhas de segurança documentadas; código fica exposto no servidor
Pascal	Bem regrada; possui IDEs muito poderosas e completas, como Delphi e Lazarus; tem a execução mais rápida e menor consumo de memória	Está caindo em desuso, pois novos desenvolvedores procuram Java, JavaScript, Python e Ruby
Python	Bem regrada; orientada à produtividade; extensas bibliotecas para serviços; comunidade crescente	Recursos de interface são menores que os das outras linguagens; faltam IDEs boas
Ruby	Suporta programação funcional, oo, imperativa e reflexiva; tipagem dinâmica	Não é bem regrada, dificultando interpretação do código; só atinge melhor potencial com framework Rails

Fonte: O autor (2017)

Observação: *as vantagens e desvantagens são apenas referências comparativas. O que pesa na escolha é a preferência da equipe de desenvolvimento ou instituição, já que todas são compatíveis com o projeto do Framework.*

APÊNDICE G — Categorias de Serviços Opcionais da Plataforma

SERVIÇOS DE MENSAGERIA

Com a popularização da Internet nos dispositivos móveis e a melhora de hardware e sistemas operacionais destes, a comunicação instantânea e ininterrupta já é uma realidade há muito anos nos grandes centros urbanos e nas organizações.

O conceito de Mensageria integra a capacidade de pessoas e sistemas se comunicarem de maneira complexa e indireta, de forma assíncrona ou em tempo real (ALURA, 2015), através de um controlador MOM (*Middleware Oriented Message*). É de grande interesse que o Projeto do Framework disponibilize a previsão para implementação de serviços para permitir que seus serviços e usuários possam se beneficiar deste tipo de tecnologia.

No mercado existem muitos *frameworks* de mensageria, como por exemplo InfoQ, JMS, AMQP e HornetQ, que foram desenvolvidos para funcionar com servidores específicos, linguagens de programação específicas, e alguns um pouco mais generalistas. Mas o objetivo do *Framework* é permitir que sejam oferecidas todas as funcionalidades de mensageria através da Plataforma, o que as tornarão genéricas e capazes de funcionar com qualquer linguagem e serviço atrelado à Plataforma. Suas vantagens são:

- Mantem o baixo grau de acoplamento entre serviços;
- Reduz o gargalo de comunicação entre sistemas;
- Tem alta capacidade de escalabilidade;
- Sua arquitetura é flexível e ágil;
- Tem se tornado um padrão mundial amplamente adotado.

Especificações de Serviço

Para os Serviços de Mensageria, as diretrizes adotadas são as mesmas do padrão comercial para manter a compatibilidade entre outros servidores e frameworks, e estão listadas abaixo:

- Todas as mensagens passam da origem ao destino através de um mediador automatizado, capas de manter um controle de entrega, fila de espera e *feedback* da situação de cada mensagem;

- As mensagens possuem as seguintes características: ID, momento, canal, proprietário, criador, conteúdo, mediador, receptor, duração e prioridade;
- Há dois tipos de canais: fila (queue) e grupo (topic);
- Há dois modelos de mensagem: ponto-a-ponto e publicar-e-assinar;
- As mensagens do tipo ponto-a-ponto (p2p) são enviadas através de uma fila que só a envia a um receptor identificado, de forma síncrona ou assíncrona, sem espera de retorno ou confirmação (modelo *fire and forget*);
- As mensagens do tipo publicar-e-assinar (pub/sub) são enviadas através de canais do tipo grupo, e podem ser recebidas por mais de um receptor, que são os assinantes deste grupo (também chamado de tópico), usando o modelo de difusão (*broadcasting*);
- Mensagens entre serviços e aplicativos não executam comandos como as RPC, mas apenas transportam informação que será utilizada para processamento por essas entidades;
- Mensagens term hierarquia de prioridade, sendo a maior prioridade a comunicação entre a Plataforma e serviços, seguido de serviços para serviços, usuários de/para serviços e por último usuários para usuários;
- A duração de uma mensagem pode ser volátil (onde a mensagem só é válida se o receptor estiver ativo) ou persistente (onde a mensagem pode ser entregue assim que receptor se conecte);
- O momento é um objeto composto, que guarda quatro datas e horas: quando ela foi enviada por um serviço, quando o serviço mediador recebeu a mensagem, e quando o serviço receptor a recebeu, e quando o usuário do serviço acessou a mensagem. Só é obrigatório aos serviços implementar o momento em que o mediador recebeu a mensagem, que é o padrão;
- O proprietário é um serviço ou aplicativo que utiliza a Plataforma como mediadora de suas tarefas de Mensageria, como por exemplo o WhatsApp.

Ações de Serviço e Seus Parâmetros

As ações do Serviço de Mensageria são referenciadas na URN como **/msg/ação/parâmetros**, e devem especificar um **serviço proprietário**, juntamente com um criador, um receptor, um canal ou uma mensagem existente.

A quantidade de comandos disponíveis deve suprir às necessidades do serviço para quaisquer tipos de cliente, confirme a lista abaixo:

- **/cadastrar** e **/descadastrar** - efetua/desabilita o registro como agente criador-receptor de mensagens, em um serviço proprietário, utilizando como parâmetros a **ID do proprietário** e a **ID do usuário** da Plataforma. O cadastramento de serviços como usuários só poderá ser feito no Painel;
- **/grupo** - cria um grupo/tópico disponível em um serviço proprietário que será gerenciado pelo usuário ou serviço que o criou. Será especificado o **/nome**, a **/descrição**, a **/visibilidade** (público ou privado) e **/modo** (escrita ou leitura);
- **/info** - retorna as informações de um usuário ou grupo com /tipo/**ID**;
- **/convite** - envia uma mensagem-convite para um **/usuário** ou **/serviço** inscrever-se em um **/grupo** existente. Este receberá um código de acesso para que o cliente do proprietário possa fazer a requisição de inscrição e ser aceito, caso o grupo seja privado. O serviço proprietário pode ter uma configuração que permite aceitar convites automaticamente;
- **/apagar** - desabilita (mas não apaga) um **/grupo**, que não mais poderá receber mensagens nem inscrições. Somente o gestor do grupo pode fazê-lo;
- **/transferir** - permite uma transferência de titularidade (permissão de administração) de um **/grupo** para outro **/usuário** ou **/serviço**;
- **/enviar** - envia uma **/mensagem** padrão para um **/usuário** ou **/serviço** ou para um **/grupo** existente no qual o criador da mensagem esteja inscrito e tenha permissão de envio (gestor ou modo escrita). Também é possível usar o parâmetro **/resposta**/idMensagem para notificar o receptor de que esta mensagem é uma resposta a uma mensagem anterior específica, o que é especialmente útil em grupos de discussão com muitos assinantes ativos;
- **/difundir** - é o ato de enviar uma **/mensagem** copiada para vários receptores inclusos num **/recipiente** definido na requisição. Este recipiente só pode conter um tipo de canal e pode ter um número indeterminado de receptores, caso o criador seja a Plataforma ou um serviço permitido no Painel (que envia em modo persistente), porém limitado para usuários padrões de um serviço proprietário, cujo limite é estabelecido também no Painel e cuja mensagem é enviada em modo volátil para não acumular no canal;
- **/inscrever** - comando para se cadastrar num **/grupo** e começar a receber suas atualizações (modo leitura) e/ou também enviar mensagens (modo escrita). Se o grupo for privado, é necessário o parâmetro **/convite**/código;
- **/sair** - cancela a assinatura de um usuário em um **/grupo**;
- **/bloquear** e **/desbloquear** - bloqueia/desbloqueia as mensagens recebidas de um **/usuário** assim como seus convites para assinar tópicos;
- **/recebida** - enviar confirmação de recebimento de uma **/mensagem**/id;
- **/lida** - enviar confirmação de recebimento de uma **/mensagem**/id;

- **/conectar** e **/desconectar** - informa ao mediador que um **/usuário** ou **/serviço** pode ou não receber mensagens **/síncronas** e/ou **/persistentes**;
- **/erro** - informa ao mediador que não foi possível ler uma **/mensagem**/id;
- **/listar** - retorna ao cliente uma lista com os grupos assinados pelo usuário.

SERVIÇOS DE MODELOS DE SAÍDA PARA O CLIENTE

A especificação do *Framework* recomenda (e facilita) que todas as camadas sejam desenvolvidas em seu devido lugar, como no caso da camada visual, que é melhor estabelecida no ambiente do cliente. Mas esta regra também pode ter suas exceções, desde que seja para tornar o desenvolvimento mais rápido e prático.

Esta modalidade foi a última a ser pensada para este *Framework* como uma solução para concluir um ciclo de requisições em uma Sequência, sem a necessidade de tratar a resposta desta Sequência no cliente e sem ferir os princípios de separação de camadas como na arquitetura MVC.

Considere-se o exemplo onde: um formulário em HTML, que possui todos os campos com o mesmo nome dos parâmetros de uma função que chama uma *stored procedure*, não precisa de um código em JavaScript para preparar a requisição e pode declará-la diretamente como *action* da *tag* <form>. Se esta requisição possuir uma solicitação de retorno de um modelo também em HTML, cujo script interno contenha todos os campos iguais aos campos de saída da *stored procedure* executada, então a Plataforma irá preencher as lacunas e retornar o HTML pronto diretamente no navegador do cliente. Isso tudo sem nenhuma linha de código adicional. É uma forma eficaz para desenvolvedores da Plataforma disponibilizarem serviços de menor complexidade em tempo hábil.

Trata-se de uma extensão do Serviço de Arquivos, que permite aplicar os dados que foram resultados do processamento de um Serviço, como entrada para substituir trechos marcados dentro de um Arquivo de Modelo. Para que isso ocorra, deve ser usada uma linguagem simples de script de marcação: um conjunto de regras de interpretação de coringas que permite um comportamento de montagem padrão no *Template*, criando sempre um conteúdo dinâmico, mas previsível, que pode ser enviado diretamente ao cliente, no seu formato inicial ou convertido em outro formato compatível, como por exemplo, conversão de HTML para PDF ou RTF.

Especificações de Serviço

Para os Serviços de Modelo de Saída para o Cliente, as diretrizes adotadas são baseadas nos padrões utilizados por muitos frameworks de camada visual como JSP, JSF e PHP, para que desenvolvedores não precisem aprender mais uma nova linguagem. As especificações estão listadas abaixo:

- Cada expressão dinâmica em um modelo deve ser envolta em chaves duplas: **{{** expressão **}}**;
- Em modelos HTML, XML ou outra linguagem de marcação, uma expressão pode ser escrita na forma de comentário **<!--** {{expressão}} **-->** permitindo a pré-visualização do conteúdo por seus designers sem que as expressões interfiram na formatação visual do modelo;
- Para que a expressão não seja interpretada pelo serviço e apareça no resultado apresentado ao cliente (como por exemplo num documento que ensine a usar as expressões), deve-ser usar chaves triplas **{{{** e **}}}**. Elas serão convertidas para chaves duplas literais no final do processamento;
- Existem sete tipos de comandos aceitos numa expressão: **$**objeto, **=**valor, #ação, @modelo, **(**condição**)**, **[**repetição**]** e **<**evento**>**;
- Mais de um comando é aceito numa mesma requisição, seguindo a prioridade da esquerda para a direita, separados pelo operador de passagem **>** que defina a relação de entra e saída de dados entre os comandos;
- O comando **$objeto** é uma variável composta por um ou mais valores, assim como um objeto JSON, acessados por por um ponto seguido do nome; Exemplo: {{$cliente}} onde cliente = {nome:'Edu'; email:'edu@ufpa.br'};
- O comando **=valor** insere, naquele ponto do modelo, o valor que está na propriedade de um objeto. Pode ser usado dentro ou fora de uma TAG. Se for usado dentro de uma outra expressão, o mesmo deve usar chaves simples **{}**; Exemplo: Seu nome é {{=cliente.nome}} onde o resultado é: Seu nome é Edu;
- O comando **#ação** irá criar uma requisição a um serviço da Plataforma. Ele deve sempre estar acompanhado de um ou dois comandos: um para entrada de dados e outro para saída. A entrada pode ser um $objeto e a saída pode ser tanto $objeto como um @modelo ou até outra #ação, desde que a última saída da expressão seja atribuída a um $objeto ou @modelo.
  Exemplo1: {{$cliente > #/banco/novoCadCli > @modelo/saidas/cadEfetuado}} onde $cliente serve como parâmetros de entrada nome e email para a função #novoCadCli, cuja saída de dados serve como parâmetro de entrada para o modelo @cadEfetuado, que será inserido dentro do conteúdo do modelo onde

esta expressão estava declarada.
Exemplo2: {{#/banco/listarProdutos/celulares > $produtos}} onde o resultado da função listarProdutos, que é uma tabela de celulares é inserida em $produtos, sendo transformada em um vetor de dados de celulares;

- O comando **@modelo** aponta para um arquivo de modelo na Plataforma, cujo conteúdo será processado e depois inserido no modelo principal, naquele ponto, e pode ou não receber um $objeto ou #ação como dados de entrada;
- O comando **(condição)** faz o mesmo papel que uma estrutura if/else em linguagens de programação e permite a tomada de decisões com base em expressões lógicas envolvendo valores de $objetos ou saídas de #ações.
  Exemplo1: {{ ($usuário.existe) #ação }}
  Exemplo2: {{ ($usuário.sexo = 'M') @enfeiteAzul | @enfeiteRosa }}
  Exemplo3: <span class="{{ ($usuário.saldo > 0) 'txtVerde' | 'txtVermelho' }}">
  Exemplo4: {{ ($resultado.erro > 0) $resultado > @/modelo/msgErroJanela }};
- O comando **[repetição]** tem dois formatos: executa outro comando repetidamente (#ação ou @modelo), ou repete uma parte do modelo, que começa depois desta expressão e termina com a expressão **{{] }}**.
  Exemplo1: {{ [$produtos] > @modelo/listas/cartãoProduto }} onde o resultado é a inserção do modelo cartãoProduto com os valores de uma iteração do vetor $produtos repetidamente até chegar ao fim do vetor.
  Exemplo2: Meus amigos são: {{ [$amigos] > $amigo }} {{=amigo.nome}} / {{] }} onde o resultado seria: Meus amigos são João / Maria / José /;
- Quando se usa comandos de [repetição] num modelo, o processador de modelos é ativado de forma recursiva para cada ocorrência deste tipo, da ocorrência mais interna, para a ocorrência mais externa na estrutura;
- O comando **<evento>** é específico para modelos HTML pois configura um evento dinâmico que cria a possibilidade de modelos mais completos, que possam interagir com serviços da Plataforma mesmo depois de enviados ao cliente. Pode ser inserido em uma linha única, ou englobar uma parte do modelo, assim como em [repetições]. Têm de 4 a 6 parâmetros que devem ser inseridos como propriedades, assim como as TAGs padrões do HTML e definem o comportamento da página no momento que o evento é disparado. Devem ser especificados o **nome** do evento, o **tipo de TAG** (div ou span) usada para englobar a parte monitorada do modelo, um arquivo de **modelo** opcional a ser inserido em volta da TAG monitorada, o **tipo de evento** (clique ou mouse-over) que será monitorado, o **tipo de ação** (abrir, baixar, info ou carregar) junto com o **alvo** desta ação, que pode ser um serviço, arquivo ou modelo. Caso o tipo de ação seja info ou carregar, o resultado do alvo será

obtido via AJAX e será inserido na página dentro do **destino** designado como o próprio local da TAG do evento ou em outro local indicado por sua #id. Exemplo1: {{ < nome='ex1' tag='span' modelo='btns/botão.html' evento='clique' ação='carregar' alvo='/db/getNome' destino='#nome' /> }} Exemplo2: {{ < tag='div' evento='over' ação='info' alvo='db/getPreço' > }} Passe o mouse sobre o produto para ver o preço**{{ />}}**;

- Ao usar um comando <evento> em um modelo, o processador insere ao final deste uma TAG <script> carregando uma biblioteca JavaScript que controla os eventos contidos no modelo, identificando cada um pelo parâmetro Nome;
- Um modelo pode ser chamado através de uma requisição de cliente, de outro serviço ou através de outro arquivo de modelo;
- Se um modelo pode ser invocado várias formas, então os valores de $objetos podem estar vazios no momento da montagem do modelo. Para evitar este comportamento falho, **a primeira expressão em todos os modelos** deve conter um $objeto e uma #ação. Assim, caso não haja parâmetros na requisição para preencher o $objeto, o resultado da #ação deverá fazê-lo. Exemplo: {{ > $produtos < #/banco/listaProdutosHome}} onde o operador de passagem não tem nenhum comando antes dele, indicando que a entrada virá de uma entidade externa para preencher $produtos, mas que na falta desta entidade, $produtos será preenchido pela função listaProdutosHome, que está usando um operador de passagem invertido;
- Num arquivo modelo também ser definida uma ação a ser executada depois que estiver totalmente montada. Para issto basta que a última expressão contenha um operador de passagem invertido seguido de um comando de #ação. Assim a plataforma criará um arquivo como resultado do processamento do modelo e o passará como parâmetro para a ação. Exemplo: {{ < #/modulo/converterHTMLparaPDF }}.

Ações de Serviço e Seus Parâmetros

As ações do Serviço de Modelos de Saída de Dados são referenciadas na URL de duas formas: requisitando somente o modelo, ou requisitando-o junto com outro serviço de saída de dados:

No primeiro caso, a URL é **/modelo@repositório/caminho/arquivo.**

No segundo caso, é incluindo o modelo como ação secundária de uma requisição de banco de dados ou serviço acoplado, a qual fornecerá os dados necessários para

preencher as expressões dentro do modelo antes de entregar o resultado ao cliente, e a URN deste tipo de requisição possui o formato **/tipo/nome/[parâmetros]@/repositório/caminho/arquivo**, sendo que serviço é substituído por **banco** ou **módulo** (acoplável), seguido do nome do serviço.

SERVIÇOS DE TRANSMISSÃO MULTIMÍDIA

Esta modalidade de serviço é interessante para administradores que precisem disponibilizar conteúdo multimídia com capacidade para serem transmitidos em tempo real para o cliente. Conteúdos como música, rádio, conversas e vídeos entram nesta categoria.

Os serviços de streaming estão cada vez mais populares entre os usuários de Internet, graças a serviços como YouTube e Nextflix, além da popularização de smartphones com Internet rápida e das *SmartTVs* capazes de reproduzir este tipo de conteúdo, facilitando o acesso para quem não tem intimidade com computadores.

Especificações de Serviço

Os serviços devem implementar as especificações RFC 3550, RFC 3711 e RFC 2326 que dizem respeito ao Protocolo de Transporte em Tempo-Real (RTP), Protocolo de Controle RTP (RTCP), Protocolo de Transferência em Tempo-Real Seguro (SRTP) e o Protocolo de Transmissão em Tempo-Real (RTSP) que são os protocolos mais utilizados atualmente nas camadas de aplicação e transporte de dados (KUROSE; ROSS, 2012) e permitem controles complexos para transmissão de mídia guardada e mídia gerada em tempo-real.

Esta escolha também permite que as capacidades de transmissão da Plataforma sejam compatíveis com as rede de distribuição de conteúdo (CDN) já existentes. Os serviços devem ser implementados seguindo especificações abaixo:

- Deve aceitar várias conexões para o mesmo cliente, separando o conteúdo transmitido em canais;
- Deve abrir um canal exclusivo para envio de comandos;
- Os comandos também poderão ser transmitidos via HTTP com o mesmo formato de URL aceitos nos outros tipos de serviços;
- Deve permitir a transmissão com ou sem encriptação;
- Só pode aceitar encriptação padrão da Plataforma nos comandos HTTP;

- Deverá utilizar somente a encriptação padrão do protocolo SRTP para dados;
- Deve permitir que o cliente escolha a resolução (qualidade) do conteúdo transmitido (seja áudio ou vídeo), quando disponíveis;
- Deve permitir que o cliente solicite à Plataforma calcule a qualidade mais adequada do conteúdo de acordo com a qualidade da conexão e disponibilidade de qualidades alternativas do conteúdo, de forma adaptativa;
- Deve admitir múltipla camada de balanceamento de carga, utilizando o gestor de balanceamento da plataforma para dividir os canais entre os servidores mais aptos a transmitir e receber multimídia, por proximidade regional ou largura de banda disponível para enviar e receber dados;
- A Plataforma deve prover relatório de status de qualidade de serviço (QoS) como uma função para cada requisição de transmissão, para fins de manutenção, estatística e/ou controle de Acordo de Nível de Serviço (SLA);
- No caso de implementação de Voz sob IP (VoIP), o desenvolvedor deverá seguir os padrões e protocolos comerciais obrigatórios;
- O conteúdo multimídia protegido por direitos autorais deve ter uma base de dados com dados do proprietário e endereço de uma API de controle de uso;
- O conteúdo tarifado deverá ter uma base de dados que especifique os valores e unidades de utilização como minutos, dados transferidos, requisições etc.;
- Deverá ser implementado uma função de aviso que acesse a API de controle de uso do proprietário do conteúdo, avisando-o sobre cada transmissão;
- A tentativa de transmissão de dados contínuos de multimídia via HTTP ou TCP deverá resultar em uma mensagem de Erro da Plataforma, com bloqueio temporário para evitar ataques do tipo DoS.

Ações de Serviço e Seus Parâmetros

As ações e parâmetros do Serviço de Transmissão de Multimídia são divididos entre **requisição** de transmissão e **controle** de transmissão de conteúdo.

As requisições de transmissão são chamadas de busca quando a intenção é carregar um conteúdo multimídia **unidirecional**, como músicas, rádio, vídeos gravados, e são chamadas de **convite**, quando o objetivo é conectar com outra fonte de transmissão de dados em tempo real, criando uma streaming **bidirecional** ou **multidirecional**, como no caso de conversas por voz ou videoconferência.

URN: **/streaming/tipo(busca,convite)/ação/parâmetros@caminho/Mídia**

Ações e Parâmetros Gerais:

- **/status** - informa os dados de qualidade de uma transmissão usando o parâmetro **/id** da transmissão, e só fornecesse essa info se o IP for o mesmo;
- **/medir** - solicita à Plataforma que calcule a melhor resolução de vídeo e/ou taxa de bits de áudio para transmitir de acordo com a conexão dos pontos;
- **/info** - envia informações gerais sobre a mídia selecionada e conexão, usando ou o **/id** da transmissão ou **@mídia**, caso ainda não esteja em transmissão;
- **/modificar** - solicita a modificação da taxa de bits (qualidade) da mídia atual;
- **/camadas** - identifica quantas camadas disponíveis em uma mídia, como por exemplo, vídeo, áudio (vários canais), legenda (várias línguas);
- **/trocar** - solicita a mudança de um dos canais de mídia, como por exemplo, o áudio de português para inglês. Também é possível adicionar canais, como por exemplo **/legenda**, que não estava transmitindo antes. No caso de chamadas, a ação pode trocar um protocolo ou adicionar vídeo ao áudio;
- **/tarifa** - verifica o valor da tarifa de uma mídia, que pode ser fixa ou variante de acordo com o tipo de mídia, localização, participantes, horário etc.

Ações e Parâmetros de Busca:

- **/tocar** - inicia a transmissão de uma @mídia se todos os requisitos forem atendidos, como banda disponível, permissão de usuário, tarifa etc. Deve-se especifica quais as **/camadas** solicitadas e **/qualidade** específica ou gerida;
- **/baixar** - permite baixar uma mídia ao invés de transmitir, o que pode ser permitido em aplicativos com permissão de acesso a conteúdo offline;
- **/congelar** - permite pausar a transmissão de uma mídia;
- **/capítulos** - recebe **/informações** sobre posições e descrição de capítulos de uma mídia ou ***/previews*** de cenas próximas à posição atual;
- **/parar** - interrompe definitivamente a transmissão, retornando a posição;
- **/rolar** - **/volta ou /avança** a transmissão para uma **/posição** diferente em relação à posição atual que está sendo transmitida, **/segundos** ou **/quadros**.

Ações e Parâmetros de Convite:

- **/listar** - exibe os endereços conhecidos do servidor, retirados de algum repositório de contatos, para ser utilizado num convite;
- **/buscar** - procura nas listas de contatos permitidas ao cliente, algum destinatário que combine com o /filtro, como nome, telefone, IP etc.;
- **/convidar** - envia um convite para conversa com o(s) destinatário(s);

- **/resposta** - resposta que pode ser enviada para um /convite recebido, informando a aceitação ou não, e os formatos de mídia disponíveis;
- **/segurar** - interrompe temporariamente a transmissão, enviando um aviso aos destinatários, que não poderão ver ou ouvir o cliente até que ele /retome;
- **/transferir** - transfere uma das pontas da transmissão do cliente atual para outro **/destino**, que recebe o convite e pode aceitar ou não, fazendo com que a transmissão retorne ao cliente anterior;
- **/desconectar** - interrompe a transmissão permanentemente. Caso o cliente seja um convidado de muitos, a ligação segue. Caso ele seja o criador da chamada, a conexão finaliza para todos os participantes;
- **/adicionar** - envia um convite para um /participante adicional, criando uma conferência.

Observação: *outro tipo de serviço de transmissão multimídia é o Acesso a Dispositivos Remoto, que é do tipo convite bidirecional, sendo que em uma direção (convidado) é enviada a mídia, e na outra (controlador) são enviados comandos, como no modo busca. Sua especificação será desconsiderada neste trabalho, por ser de interesse comercial muito específico*.