PODER JUDICIÁRIO
JUSTIÇA DO TRABALHO
CONSELHO SUPERIOR DA JUSTIÇA DO TRABALHO

Processo Administrativo nº 501.215/2015-1
Memorando CSJT.CTENGSOFT nº 1

Brasília, 12 de junho de 2015.

Ao Supervisor da Seção de Serviços e Sistemas Nacionais
JOMAR PEREIRA DA SILVA

Assunto: Resposta ao Comitê Técnico Temático de Engenharia de Software (ctEngSoft) diante da solicitação do gtPessoalJT de parecer técnico.

Senhor Supervisor,

Em resposta à solicitação do Grupo de Trabalho gtPessoalJT, instituído pelo Ato CSJT.GP.SG.SEIT.CGGOV n. 314/2014 feita por meio do ofício n. GTPESSOALJT/001/2015, o Comitê Técnico Temático de Engenharia de Software do Conselho Superior da Justiça do Trabalho apresenta anexo o parecer solicitado.

Na oportunidade, renovo protestos de elevada estima e consideração.

CLAUDSON DOS SANTOS MELO:31633

Assinado de forma digital por CLAUDSON DOS SANTOS MELO:31633
DN: c=BR, o=ICP-Brasil, ou=Cert-JUS Institucional - A3, ou=Autoridade Certificadora da Justica - ACJUS v4, ou=TRIBUNAL SUPERIOR DO TRABALHO-TST, ou=SERVIDOR, cn=CLAUDSON DOS SANTOS MELO:31633
Dados: 2015.06.22 16:14:03 -03'00'

CLAUDSON DOS SANTOS MELO
Coordenador do Comitê Temático de Engenharia de Software/CSJT

ANEXO I

Parecer Técnico ctEngSoft nº 1/2015

Parecer ctEngSoft/CSJT nº 1/2015

Assunto: Consulta sobre a viabilidade da evolução da base de dados do Sistema SGRH como parte da solução do SIGEP; e sobre a análise da arquitetura que suporte o desenvolvimento e a sustentação de uma solução de gestão de pessoas que atenda a toda a Justiça do Trabalho.

Ementa: ENGENHARIA DE SOFTWARE – ANÁLISE DE MODELO DE BANCO DE DADOS – ANÁLISE DE ARQUITETURA DE REFERÊNCIA.

Análise da base de dados do sistema legado SGRH, desenvolvido pelo TSE; análise de arquiteturas para evolução (arquitetura mínima) e para substituição (arquitetura de referência).

Possibilidade de utilização do modelo de dados do atual sistema SGRH, porém, com ressalvas.

Interessado: Grupo de Trabalho gtPessoalJT/CSJT.

1. RELATÓRIO

O Grupo de Trabalho gtPessoalJT/CSJT, instituído por meio do Ato n.º CSJT.GP.SG.SEIT.CGGOV Nº 314, demanda parecer técnico acerca da viabilidade da evolução da base de dados do Sistema SGRH como parte da solução do SIGEP; e sobre a análise da arquitetura que suporte o desenvolvimento e a sustentação de uma solução de gestão de pessoas que atenda a toda a Justiça do Trabalho.

Segundo Estudo Técnico Preliminar, elaborado pelo gtPessoalJT, cujo objeto de estudo visa à integração da Gestão de Recursos Humanos e da Folha de Pagamento no âmbito da Justiça do Trabalho do 1º e 2º graus, o SGRH, sistema para gerenciamento de pessoas, desenvolvido pelo Tribunal Superior Eleitoral e adequado pelo TRT/2ª Região foi apontado pelo CSJT como solução alternativa a ser adotada, no âmbito do Judiciário Trabalhista, porém fora apontada a

restrições como demonstrado a seguir: *"na análise feita até o momento, inclusive com reuniões presenciais para avaliação do SGRH-SP, não obtivemos garantias de que haverá, com sua implantação, atendimento a todos os requisitos necessários."*

Vale ressaltar a necessidade de convergência de esforços apontada pelo gtPessoalJT, em reunião ocorrida em 4/2/2015, conforme transcrição *in verbis:*

> *"A coordenadora do grupo e o representante de TI do TRT3ª Região apresentaram suas preocupações com relação à necessidade de convergência de esforços e reaproveitamento máximo dos artefatos e produtos gerados a partir da implantação do Sistema SGRH com o objetivo final de se obter uma solução única. Destacaram que o projeto de implantação do SRGH deve ser feito de modo a possibilitar o aproveitamento do esforço no contexto do projeto de desenvolvimento da nova solução – SIGEP.Ponderaram que a partir dessa convergência seria possível aumentar as chances de sucesso de implantação de uma solução única, moderna e robusta para a toda Justiça do Trabalho, bem como seria possível obter um maior envolvimento dos Tribunais gerando maior colaboração e, por consequência, resultados mais rápidos e consistentes.*
>
> *Nesse sentido, os representantes do TRT3 apresentaram proposta de revisão do escopo da proposta de projeto do SIGEP no sentido de se ter uma solução que contemple a possibilidade de aproveitar o esforço de migração feito do sistema legado para o SRGH, no contexto do desenvolvimento da nova solução - SIGEP. A proposta também inclui como requisito para a nova solução (SIGEP), a necessidade de formulação de uma arquitetura de referência e uma outra que contenha os requisitos mínimos.*
>
> *O Coordenador de Gestão e Governança de TI ressaltou como requisito de viabilidade vislumbrado para a nova proposta de solução, a garantia de que o modelo de dados do SRGH seja consistente e capaz de suportar as evoluções funcionais e de arquitetura planejadas para o SIGEP. Nesse sentido, apontou a necessidade de que a revisão da proposta de projeto do SIGEP implique necessariamente em consignar como premissas: (p1) consistência e possibilidade de evolução do modelo de dados; (p2) existência de arquitetura mínima; (p3) existência de arquitetura de referência. Também consignou a importância da definição das seguintes restrições: (r1) manutenções no SRGH somente mandatórias; (r2) evoluções no SGRH somente se priorizadas pelo grupo e realizadas em Java, respeitando os padrões de arquitetura mínimos e de referência; (r3) Construção de funcionalidades no SIGEP somente em Java, respeitando os padrões de arquitetura mínimos e de referência.*
>
> *Nesse contexto,* ***os participantes definiram que como atividade preliminar à execução do projeto, será feito um estudo a respeito do modelo de dados do SGRH, com parecer conclusivo de grupo técnico especialista na matéria. O gtPessoalJT definirá os requisitos que embasarão a avaliação e aguardará os resultados para instruir as suas decisões****.*
>
> *Os participantes da reunião também chegaram a um consenso que as manutenções mandatórias no SRGH devem ficar a cargo do TRT 2ª Região, visto que ele está*

prestes a colocar o seu sistema (SGRH) em produção e a atividade de mantê-lo será natural e indispensável ao funcionamento da área de pessoal do Regional.

Também foi levantada a possibilidade de envolvimento de mais tribunais na conjugação de esforços para o desenvolvimento da futura solução SIGEP.

Então, ***ficou acertado que a coordenação do gtPessoalJT fará a convocação da equipe em meados de março a fim de se reunirem com o propósito de demandar o grupo técnico responsável pela elaboração do parecer de avaliação do modelo de dados do SGRH, para em seguida produzir a revisão da proposta de projeto nos termos idealizados.***" (grifos nossos)

A formalização do pedido ao Comitê Temático de Engenharia de Software - ctEngSoft se deu por meio de ofício 1/2015 com os seguintes termos:

"O Grupo de Trabalho gtPessoalJT, instituído pelo Ato CSJT.GP.SG.SEIT.CGGOV n. 314/2014, solicita ao Comitê Técnico Temático de Engenharia de Software do Conselho Superior da Justiça do Trabalho, conforme Anexo I, parecer técnico: (a) sobre o Modelo de Dados do SGRH, com o objetivo de verificar sua capacidade de evolução, esclarecendo, alfim (sic): (a.1) considerando o conjunto de critérios avaliados, se o Comitê considera viável o uso do modelo de dados do SGRH como ponto de partida para a construção de um sistema corporativo de gestão de pessoas para a Justiça do Trabalho e (a.2) em caso afirmativo, se existe algum ponto de atenção que deva ser considerado ao se utilizar este modelo;

Para avaliação do Modelo da Base de Dados do SGRH, foram submetidos:

1. documento com doze itens, cada qual contendo escala de 0 a 10, com respectivo peso e aspectos a serem considerados na análise deste Comitê; e

2. modelo de dados nos formatos Designer da Oracle e Powerdesigner.

Para avaliação de requisitos arquiteturais foram submetidos:

1. arquivos demonstrando as arquiteturas mínima, de evolução e de referência.

É o relatório.

2. FUNDAMENTAÇÃO

Análise do modelo (Escopo e Metodologia adotada)

A análise do modelo realizada por este Comitê não versa sobre aspectos semânticos ou de abrangência às necessidades de negócio dos usuários da Justiça do Trabalho. Assim, aderência dos requisitos implementados e, consequentemente, a estrutura de dados para comportar tais necessidades, não é escopo deste trabalho.

Para análise do modelo de dados seguiu-se com a seguinte abordagem:

1) Oitiva do servidor HUDSON LINCOLN GOMES DOS SANTOS. Devido à quantidade enorme de objetos de dados e necessidade de apurar se o software abrange aspectos de históricos do servidor bem como, por exemplo, entender o comportamento do sistema quanto a possível estratégia utilizada para replicação de dados, o Comitê fez convite ao técnico Hudson do TRT 2ª Região, envolvido na implantação e evolução da solução SGRH, para participar da reunião do Comitê ocorrida no dia 15/7;
2) Espaço amostral das tabelas para 600 objetos, envolvendo os módulos: Afastamento, Benefícios, Capacitação, Dependentes, Designação de Magistrados e de Folha de Pagamento;
3) Não nos foi passada documentação de análise de requisito, assim a análise foi realizada a partir do conhecimento dos integrantes do Comitê de seus sistemas de Recursos Humanos;
3) Para pontuar o documento repassado pelo gtPessoalJT, atribuição da seguinte significação para os números abaixo:

0 - Não atende
3 - Atende minimamente
5 - Atende parcialmente
7 - Atende majoritariamente
10 - Atende integralmente

4) A partir da pontuação atribuída, foram feitas considerações nos itens que tiveram pontuação menor ou igual a 3 (três) e peso maior ou igual a 3 (três), onde a ênfase foi

dada aos itens de menor "qualidade" e de maior relevância, seguindo os critérios de relevância sugeridos pelo gtPessoal.

Análise do modelo de dados

Segue abaixo, conforme constante do documento de análise do modelo repassado, os requisitos analisados pelo Comitê com as notas atribuídas:

Requisito	Sub-requisito	P.*	Peso
Documentação mínima		**5**	4
	Dicionário de Dados	5	
	Modelo de Dados	7	
	Documentação de requisitos	3	
Normalização		**3**	3
Integridade Referencial		**7**	4
Número de colunas das tabelas		**5**	2
Regra de Negócio implementada na base		**5**	3
Dados redundantes		**3**	3
Relacionamentos redundantes (ciclos no modelo)		**7**	3
Padrões de nomeação		**3**	1
Utilização de chaves naturais ou surrogate		**3**	4
Modularização		**7**	1
Necessidade de Particionamento da Base		**3**	1
Metadados integrados ao banco		**7**	1

* Pontuação dada pelo Comitê baseada em análise qualitativa

Conforme definido na metodologia adotada por este Comitê os pontos objeto de análise deste parecer são:

1. Normalização

A existência de tabelas com falhas de normalização é um ponto preocupante, pois geram replicação de dados e o aumento da complexidade do modelo. Apesar de não ser escopo do

trabalho deste Comitê a análise semântica, segue abaixo alguns exemplos dos problemas encontrados:

- a inexistência de uma entidade Vínculo, forçando que um servidor seja duplicado sempre que muda de cargo, pois essa é a única maneira de se manter o histórico;
- duplicidade dos mecanismos de controle de acesso. Existem tabelas específicas por papel de usuário do sistema e o controle é feito através da inclusão/exclusão de usuários destas tabelas. Como exemplo, temos a tabela SM_USUARIO_REEMBOLSO, que lista os usuários (servidores) autorizados a utilizar o sistema de administração de reembolso farmacêutico;
- colunas relativas à entidade Vínculo e a processos administrativos ficam na entidade servidor: NUM_PROC_TCU_DESLIG, VEIC_PUBLIC_TCU_DESLIG, TAB_VEIC_PUBLIC, DT_PUBLIC_PROC_TCU_DESLIG, DT_JULGAMENTO , ATA_JULGAMENTO, CD_VEIC_PUBLIC_PROV, CD_VEIC_PUBLIC_DESLIG, SEQ_CONCURSO, SEQ_CANDIDATO. Isso leva servidor a ter potencialmente várias colunas que na maioria dos casos é nula.

2. Dados redundantes

Devido às falhas de normalização supracitadas, dados replicados são encontrados em algumas tabelas projetadas inicialmente pelo próprio TSE. Teoricamente dados redundantes, além de acumular espaço desnecessário, dificultam também a identificação dos dados mais recentes. Os dados redundantes encontrados no *schema* SRH2 foram:

- Duplicidade no cadastro de servidores.
- Duplicidade dos mecanismos de controle de acesso;

Existe a possibilidade de utilização de duplicidade de dados com o intuito de facilidade de acesso aos dados, ou ainda, para melhorar o desempenho do sistema. Porém, a utilização do recurso de duplicidade de dados deve ser feita de forma criteriosa e que garanta a integridade dos dados.

Essa hipótese é considerada pelo renomado escritor, conferencista, pesquisador e consultor independente especializado em sistemas de bancos de dados relacionais C. J. Date:

"A idéia geral de normalização é que o projetista do banco de dados deveria almejar ter relações em forma normal 'última' (5FN). Porém, esta recomendação não deve ser tomada como lei. Ocasionalmente podem existir boas razões para desafiar os princípios de normalização. A única exigência firme é que as relações estejam pelo menos na primeira forma normal." [DATE, 2012]

Conforme apontado nas entrevistas com o técnico da 2ª Região, no sistema SGRH utilizado por eles, a integridade dos dados é garantida por meio de *Stored Procedures* e/ou código Delphi.

3. Utilização de chaves naturais ou surrogate

O modelo se utiliza mais de chaves naturais do que *surrogate key*.

A utilização de *surrogate key* facilita a implementação, pois desacopla o modelo do negócio, garante a imutabilidade do registro e facilita a construção de consultas, diminuindo custos de manutenção e evolução. Já a abordagem de se utilizar chaves naturais é vantajosa quando permite a fácil identificação de informações corporativas com o objetivo de geração de relatórios.

Em alguns casos é possível gerar chaves naturais de acordo com a estabilidade do dado ou pela necessidade do sistema, para tanto seria necessário uma análise de requisitos, e posteriormente uma análise semântica para identificação da possibilidade de geração de chaves naturais.

A migração do atual modelo de chaves para a utilização de *surrogate key* demandaria possivelmente profundas mudanças na aplicação e no banco de dados, porém este Comitê não está apto a fazer nenhuma estimativa acerca disso.

Análise da Arquitetura (Escopo e Metodologia adotada)

Foram analisados três documentos para emissão deste parecer: "Roadmap de Arquitetura SGRH" , "Arquitetura de Integração de Software (Etapa 1 e Etapa 2 - *Road Map* Arquitetural)" e "Arquitetura de Software de Referência para Aplicações na plataforma Java EE" .

O escopo desta análise focou-se na arquitetura mínima e na arquitetura de evolução, tendo em vista que diversos aspectos da arquitetura única serão desdobramentos da arquitetura de

evolução e, como esta é para médio ou longo prazo, as tecnologias sugeridas estão aderentes ao Java EE 7 e poderão ser revistas ou atualizadas.

Por questões de economicidade, a utilização de barramento de serviços deverá ser submetida ao Comitê IntegraJT. Nesse Comitê há ações para tratar do assunto.

Análise da Arquitetura

A solução arquitetural proposta foi inteligente em dividir os riscos da integração dos módulos em três etapas em que gradativamente aumentam a complexidade e demandam mais esforço.

O objetivo da primeira etapa é integrar os módulos e tentar homogeneizar as tecnologias de banco de dados e do servidor de aplicação. A arquitetura dos módulos será respeitada e as integrações serão feitas diretamente entre os módulos, utilizando-se serviços ou *views* de banco de dados, com intuito de diminuir o acoplamento e estabelecer contratos para troca de informações. Essa estratégia acelera a integração e permite a entrega mais rápida de valor.

Nos documentos enviados a este Comitê não foram encontradas recomendações que orientem como as integrações via serviços ou *views* de banco de dados devem ser criadas.

De acordo com a proposta, a segunda etapa foca no aprofundamento das integrações entre as diversas aplicações por meio da criação de um conjunto de serviços que permitirão o Single Sign On (SSO), centralização da auditoria e estatísticas, barramento de serviços e protocolos para o consumo e prestação de serviços. Além disso, determina que novas funcionalidades devem obrigatoriamente utilizar a arquitetura de referência.

A intenção de aprofundamento através do SSO e da padronização dos serviços é importante, porém a obrigatoriedade de utilizar a arquitetura de referência pode levar a uma estagnação dos sistemas, gerando riscos a todo projeto. A simples recomendação de uso parece ser mais adequada, permitindo que cada caso seja avaliado separadamente.

Segundo a proposta apresentada, a etapa três objetiva que todos os sistemas sejam reescritos, utilizando os mesmos padrões tecnológicos (Arquitetura de Referência) e compartilhem um único modelo, o chamado modelo canônico. A arquitetura de referência é vanguardista, o que oferece uma sobrevida para a proposta, contudo até junho de 2015 não existem servidores de aplicação com suporte empresarial homologados que sustentem tal

arquitetura, fato que não parece ser um empecilho, pois pelo que foi proposto para etapa três pode-se considerar que a implantação levará tempo significativo para ser concluída.

É o nosso parecer.

3. CONCLUSÃO

Realizadas cinco reuniões não presenciais, ocorridas entre os dias 7/5 e 11/6, o Comitê Temático de Engenharia de Software do CSJT analisou o modelo de dados do sistema SGRH e as arquiteturas propostas.

Destaca-se que a análise do modelo realizada por este Comitê não versa sobre aspectos semânticos ou de abrangência às necessidades de negócio dos usuários da Justiça do Trabalho.

É importante salientar, ainda, que não foi objeto deste trabalho a quantificação do esforço para realização de tarefas de manutenções para a remodelagem de dados ou para as implantações das arquiteturas propostas.

Com base na análise do modelo de dados, apesar da evolução do modelo ser possível, este Comitê reconhece dificuldades inerentes ao modelo escolhido, principalmente no que tange à normalização. Pensando em um modelo nacional, seria importante a adoção de um modelo de dados onde a normalização não exigisse a duplicação de dados e que as chaves primárias fossem desacopladas do negócio, diminuindo esforço de manutenção e o evitando assim o aumento do débito técnico com o passar do tempo.

Diante da possibilidade de utilização do modelo, o Comitê sugere que os pontos de atenção/recomendações abaixo sejam considerados:

1. **Para os aspectos de Modelagem de dados:**

1.1. Melhorar a documentação dos objetos de banco de dados (dicionário de dados, tabelas, views) ;

1.2. Avaliar a possibilidade de correção de normalização, principalmente, dos pontos apontados pelo Comitê neste parecer;

1.3. Adotar, preferencialmente, o uso de *surrogate key*, contudo, avaliar a possibilidade de se identificar dados estáveis e, caso seja pertinente, utilizar-se da criação de chaves naturais em detrimento de *surrogate keys*.

2. Para os aspectos de Arquitetura:

2.1. É recomendável que as integrações ocorram por meio de *views* (objetos de banco de dados) durante a primeira etapa do *Road Map* proposto, e nas etapas subsequentes recomenda-se avaliar a substituição dessas pela integração por meio de serviços, dependendo da finalidade da integração;

2.2. Por questões que envolvem risco para o negócio, recomenda-se a utilização de tecnologias suportadas e homologadas pela Justiça do Trabalho, inclusive para servidor de aplicações.

Referências Bibliográficas

DATE, C. J. - Introdução a Sistemas de Banco de Dados. 8ª ed. Rio de Janeiro: Elsevier. 2012. 896 pág.

SOMMERVILLE, Ian – Engenharia de Software. 6ª ed. São Paulo: Addison Wesley, 2003.

http://www.techrepublic.com/blog/10-things/10-tips-for-choosing-between-a-surrogate-and-natural-primary-key/ Acesso em 2 de junho de 2015.

http://sqlmag.com/business-intelligence/surrogate-key-vs-natural-key Acesso em 2 de junho de 2015.

http://www.agiledata.org/essays/keys.html#SurrogateKeyImplementation Acesso em 2 de junho de 2015.

Equipe do Comitê Temático de Engenharia de Software que trabalhou na confecção deste parecer:

André Soares Farias – Tribunal Regional da 4ª Região

Claudson dos Santos Melo – Tribunal Superior do Trabalho

Euler Cruz de Souza – Conselho Superior da Justiça do Trabalho

Felipe Pompeo Pereira – Tribunal Regional da 12ª Região

Maurício Augusto Figueiredo – Tribunal Regional da 19ª Região

Thiago Ferauche – Tribunal Regional da 2ª Região

Vera Lúcia Rocha – Tribunal Regional da 5ª Região

SIGEP-SGRH	Versão 1.2
Arquitetura de Integração de Software	Data: 28/09/2015

Arquitetura de Integração de Software - SGRH-SIGEP

(Arquitetura Mínima)

Histórico de Revisões

Data	Versão	Descrição	Autor
11/02/2015	1.0	Definição inicial em conjunto em reunião de videoconferência.	
23/02/2015	1.1	Refinamento e versão final do documento pelo grupo de arquitetura	
28/09/2015	1.2	Adequações referentes à 3ª reunião de trabalho do gtPessoalJT	

SIGEP-SGRH	Versão 1.2
Arquitetura de Integração de Software	Data: 28/09/2015

Sumário

- Introdução
  - Referências
- Metas e Restrições da Arquitetura
  - Metas
  - Restrições
- Visão Lógica
  - Visão Geral
- Visão de Processos
  - Autenticação na perspectiva dos Módulos Existentes
  - Autenticação na perspectiva da Camada de Integração
  - Autenticação na perspectiva da Base de Dados
- Visão de Implantação
  - Requisitos mínimos à implantação dos Módulos Existentes
  - Requisitos mínimos à construção da Camada de Integração
- Visão da Integração
- Gerência de Configuração
  - Gerência de Configuração dos Módulos Existentes
  - Gerência de Configuração da Camada de Integração
  - Testes Automatizados
- Segurança
  - Comunicação
  - Auditoria

Arquitetura de Integração de Software SGRH-SIGEP

1. Introdução

Este documento oferece uma visão geral abrangente da arquitetura para a integração dos módulos do SGRH-SIGEP da Justiça do Trabalho a ser utilizada na primeira fase do Plano de Evolução da Arquitetura do *SGRH-SIGEP*, objetivando no futuro um sistema único integrado. Usando diversas visões arquiteturais para representar diferentes aspectos do sistema, o objetivo deste é estabelecer um conjunto de padrões arquiteturais a serem mantidos no processo de integração entre os módulos, para facilitar a adaptação dos mesmos, sua implantação e posterior manutenção evolutiva e corretiva.

1.1 Referências

- Arquitetura de Referência JEE v1.1.pdf
- Análise Versão JEE - Arq. Ref v1.2.pdf
- Plano de Evolução de Arquitetura (Roadmap Arquitetural) SGRH-SIGEP.PDF

2. Metas e Restrições da Arquitetura

2.1 Metas

- Estabelecer a **Arquitetura Mínima** para integrar os módulos existentes do SGRH/SIGEP;
- Prover um login único e Single Sign-on de maneira segura;
- Manter registros de auditoria (logs) de forma centralizada;
- Utilizar a arquitetura de referência para qualquer módulo novo a ser desenvolvido.

2.2 Restrições

- Alterar o mínimo necessário a arquitetura interna dos módulos já existentes que estão sendo integrados, na primeira fase do Plano de Evolução da Arquitetura;

- A Base de Dados que contém as informações sobre os servidores (relativos ao cadastro funcional e folha de pagamento) será centralizada no SGRH e seus dados disponibilizados por meio de serviços. Informações funcionais não devem ser replicadas nos módulos da solução;

- Utilizar o padrão de comunicação de serviços chamado REST, uma vez que a arquitetura prevista para a segunda fase a define como padrão. Eventualmente, durante a primeira etapa, será permitida a integração apoiada em modelo de contrato (padrão SOAP e WSDL), ainda que não utilize o modelo de de serviços, nos casos em que a tecnologia envolvida entre os módulos permitir essa comunicação de maneira menos onerosa (Ex. Java). Na fase 1, a utilização de views pode ser realizada desde que a regra de seleção de dados esteja estabelecida explicitamente na consulta.

3. Visão Lógica

3.1 Visão Geral

É possível decompor a Arquitetura de Integração em 3 partes: Módulos Existentes, Camada de Integração e Base de Dados;

- **Módulos Existentes:** refere-se aos módulos já utilizados nos regionais, que foram avaliados pela equipe de negócio, e devem ser integrados para compor o ferramental administrativo mínimo de todos os regionais. Estes módulos possuem arquitetura (interna) própria, utilizam tecnologia heterogênea, tanto para banco de dados quanto para linguagem de programação. Na primeira fase do plano de evolução da arquitetura esforços serão necessários e concentrados em adequar estes módulos com intuito de:
  - Consumir, e dependendo do caso prover, serviços no padrão REST;
  - Utilizar os bancos de dados e servidores de aplicação nas versões mínimas explícitas na seção “Visão de Implantação”;
  - Adequar-se as ferramentas sugeridas na seção “Gerência de Configuração”
  - Efetuar registro para fins de auditoria de maneira local.

Os módulos que serão integrados na primeira fase, tendo como base as definições de Arquitetura Mínima contidas neste documento são os seguintes:

Regional	**Módulo**	**Linguagem e Frameworks**	**Banco de dados**	**Servidor de Aplicação**	**Sistema Operacional / Navegadores**	**Forma de autenticação**
TRT2	SGRH (departamentos administrativos)	Delphi 7, Quick Reports, SAD Framework	Oracle 11G	-	Windows XP ou WIndows 7	LDAP
TRT2	SGRH Online (auto-atendimento)	Java 5, Struts 1.3.1	Oracle 11G	TomCat 5.5.1 / JBoss EAP 6.1	FireFox e Chrome	LDAP
TRT3	Diárias	Java EE 6	Oracle 11G	JBoss EAP 6.3	Firefox e Chrome	LDAP
TRT2	Registro de Logs	Delphi 7 / SAD Framework	Oracle 11G	-	Windows XP ou Windows 7	LDAP
TRT12	Autoatendimento	SEAM 2, JSF 1.2, richfaces 3.3, hibernate 3.2, resteasy-jaxrs 2, struts-tiles 1.2.7, JCompany 5.2.1, Java 6	Oracle 9, 10, 11	Jboss 4.3 , Jboss 5.1 (com pequenos ajustes),Tomcat 6(Desenvolvimento)		Suporta Ldap, AD e BD
TRT6	Avaliação de Desempenho	JEE5, JBoss Seam 2.2.2, JSF 1.2, JPA 1.0	Oracle 11G	JBoss AS 5.1.0.GA	Chrome (10.0 ou superior), Firefox (5.0 ou superior), Internet Explorer (8.0 – 9.0)	LDAP
TRT6	Avaliação por Competências	JEE5, JBoss Seam 2.2.2, JSF 1.2, JPA 1.0	Oracle 11G	JBoss AS 5.1.0.GA	Chrome (10.0 ou superior), Firefox (5.0 ou superior), Internet Explorer (8.0 – 9.0)	LDAP
TRT2	Lotação - Quadro de Vagas	Delphi 7 / SAD	Oracle 11G	-	Windows XP ou	LDAP

	por Lotação	Framework			Windows 7	
TRT18	Pasta Funcional Eletrônica	Java/JEE JDK 1.6.0_37 / Seam 2.1.2 / RichFaces 3.3.2.GA / Hibernate 4.0.2.GA	Oracle 9i	JBoss EAP 4.3.0.GA	Mozilla Firefox	Biblioteca própria SCSEG, que trabalha integrada com o LDAP

- **Camada de Integração:** refere-se a pacotes de software que serão desenvolvidos para prover os serviços necessários à integração dos módulos existentes. Esses pacotes deverão ser desenvolvidos utilizando a arquitetura de referência e registrar logs para fins de auditoria de maneira local.
- **Base de dados:** os módulos existentes deverão manter em suas bases somente as informações que dizem respeito ao seu domínio, evitando assim qualquer tipo de replicação de dados. Deverão também, caso seja necessário, adequar suas bases para a versão de banco de dados indicada na visão de Implantação. Os módulos existentes caso necessitem de informações que fogem ao seu domínio, deverão requisitá-las via serviços REST utilizando a camada de Integração, e nos casos de atualização de informações, nunca diretamente por meio do Banco de Dados.

4. Visão de Processos

4.1 Autenticação na perspectiva dos Módulos Existentes

Os módulos existentes utilizarão de suas próprias interfaces gráficas para prover a autenticação do usuário, porém a autenticação será requisitada para a Camada de Integração via serviços.

4.2 Autenticação na perspectiva da Camada de Integração

A camada de integração irá prover o serviço de autenticação, podendo este ser parametrizado para realizar a autenticação via LDAP ou Banco de Dados. A gerência do tempo de sessão do usuário é de responsabilidade de cada módulo.

4.3 Autenticação na perspectiva da Base de Dados

Deverá existir um usuário padrão para conexão ao banco de dados, sendo um usuário padrão diferente para cada módulo existente e um usuário padrão para os pacotes de integração. Estes usuários deverão possuir privilégios para tratar somente as informações pertinentes ao domínio do problema de seu respectivo módulo.

5. Visão de Implantação

A meta para a primeira fase do Plano de Evolução da Arquitetura é integrar os módulos definidos como prioritários pelos usuários de negócio e definir uma linha de corte, reduzindo a heterogeneidade tecnológica de banco de dados e servidor de aplicação com o objetivo de minimizar a dificuldade de

implantação. Dessa maneira são porpostos os requisitos mínimos abaixo:

5.1 Requisitos mínimos à implantação dos Módulos Existentes

- Banco de Dados: Oracle 11g ou PostgreSQL 9.3 (preferencialmente nessa ordem)
- Servidor de Aplicação: JBoss EAP 5.1 ou JBoss EAP 4.2 (em casos excepcionais)
- Plataforma de Desktop: Windows 7 / Windows XP

Os módulos já existentes, em sua grande maioria, fazem uso do Banco de Dados Oracle já na versão 11g. O banco de dados PostgreSQL já faz parte da infraestrutura de TI dos Regionais e pode ser utilizado eventualmente em substituição ao Oracle. O servidor de aplicação JBoss EAP 5.1 é o servidor atual da solução PJe e portanto de amplo conhecimento pelos Regionais.

5.2 Requisitos mínimos à construção da Camada de Integração

- Banco de Dados: Oracle 11g
- Servidor de Aplicação: WildFly 8

Conforme documento de referência “Arquitetura de Referência Técnica JEE’”, o servidor de aplicação WildFly 8, utilizando JEE7 provê melhor solução técnica para trabalhar com serviços.

6. Visão da Integração

Os módulos existentes deverão se comunicar por meio da camada de integração, que deverá seguir as recomendações abaixo:

- Nunca fazer integração de módulos ou sistemas por acesso direto às bases de dados (*shared database*)
- Criar um modelo intermediário, aderente às regras de negócio da Justiça do Trabalho, independente dos módulos em si.
  - Algo similar ao modelo canônico, mas “menos burocrático”
  - Esse modelo independente seria necessário apenas entre os módulos que precisam se comunicar e tem modelos internos distintos.
- Utilizar em cada aplicação de serviços a solução REST

7. Gerência de Configuração

7.1 Gerência de Configuração dos Módulos Existentes

Para os módulo existentes sugere-se como infraestrutura mínima para a gerência de configuração:

- Versionamento de código: Utilizar o versionador Git, por ser mais apropriado para o desenvolvimento distribuído.

7.2 Gerência de Configuração da Camada de Integração

Para a camada de integração obrigatoriamente a infraestrutura mínima deve ser composta:

- Versionamento de código: Git
- Integração contínua: considerar o uso de servidor de integração contínua (Jenkins)
- Construção: os artefatos devem ser construídos utilizando obrigatoriamente Maven 3.

7.3 Testes Automatizados

Obrigatoriamente para a camada de integração, e quando possível nos Módulo Existentes, deverão ser utilizados testes automatizados:

- Testes unitários cobrindo partes críticas do código;
- Considerar o uso de testes funcionais automatizados;

8. Segurança

Os módulos só deverão ser integrados respeitando os critérios de segurança explícitos a seguir.

8.1 Comunicação

A comunicação entre os módulos existentes e a camada de integração deverá se utilizar de certificados digitais de servidores (A1), e comunicação através de SSL, com o intuito de garantir a procedência da requisição e resposta do serviço, mesmo que o módulo existente e o provedor de serviço estejam implantados no mesmo servidor físico.

8.2 Auditoria

Os módulos existentes e a camada de integração deverão registrar eventos de auditoria de maneira local, a serem coletados de maneira automatizada posteriormente, e armazenados em servidor centralizado. O evento de auditoria deverá conter o seguinte padrão de informação:

- Usuário: usuário que executou a funcionalidade do módulo existente/camada de integração
- Funcionalidade: nome da funcionalidade do módulo/camada de integração
- Dados de entrada: dados de entrada utilizados na funcionalidade do módulo/camada de integração
- Data e hora: data e hora do registro
- Endereço IP: Endereço da máquina requisitante

Obs.: Algumas das informações trafegadas pelos serviços podem, eventualmente, não ser gravadas por questão de sigilo (ex.: redefinição de senha), ou inviabilidade técnica.

PODER JUDICIÁRIO
TRIBUNAL REGIONAL DO TRABALHO DA 3ª REGIÃO
TRIBUNAL REGIONAL DO TRABALHO DA 6ª REGIÃO

Arquitetura de Software de Referência para Aplicações na plataforma Java EE

Tribunal Regional do Trabalho da 3ª Região
Tribunal Regional do Trabalho da 6ª Região

Histórico de Revisões

Data	Versão	Descrição	Aprovado por
15/01/2014	1.0	Versão inicial da Arquitetura, unindo e evoluindo as arquiteturas do TRT3 e TRT6	
09/02/2014	1.1	Versão para apreciação dos demais Órgãos da Justiça do Trabalho	

Autoria

Nome	Cargo	Orgão	E-mail
Alexandre A. Drummond	Analista Judiciário, TI	TRT3	alexadb@trt3.jus.br
Augusto Breno de F Lima	Analista Judiciário, TI	TRT6	augusto.breno@trt6.jus.br
David Henrique Sales Vieira	Analista Judiciário, TI	TRT6	david.vieira@trt6.jus.br
Denis A. Altoé Falqueto	Analista Judiciário, TI	TRT3	denisf@trt3.jus.br
Fabiano G. Souza	Analista Judiciário, TI	TRT3	fabianog@trt3.jus.br
Juliano Amorim de Oliveira	Analista Judiciário, TI	TRT3	julianoo@trt3.jus.br

Classificações do Documento

Meio de Distribuição	(x) Impresso (x) Eletrônico
Reprodução	( ) Parcial (x) Total
Uso	(x) Interno ( ) Público

Sumário

1. Introdução
    1.1. Objetivo
    1.2. Escopo
    1.3. Definições, Siglas e Abreviaturas
    1.4. Referências
2. Direcionadores
    2.1. Plataforma de Desenvolvimento Java
    2.2. Infra-estrutura Tecnológica
    2.3. Princípios Arquiteturais
    2.4. Front-End Web e suporte a dispositivos móveis
    2.5. Acessibilidade
    2.6. Tendências futuras
3. Requisitos Inversos
4. Visão Geral da Arquitetura
    4.1. Visão Lógica
        4.1.1. Visão de Camadas
            4.1.1.1. Camada de Apresentação (Presentation tier)
            4.1.1.2. Camada de Negócios
            4.1.1.3. Camada de Integração (Integration tier)
                4.1.1.3.1. Sub-camada: Persistência
                4.1.1.3.2. Sub-camada: Adapter
                    4.1.1.3.2.1. Serviços REST
    4.2. Visão de Implementação
        4.2.1. Pacotes e Classes
            4.2.1.1. Visão Externa
            4.2.1.2. Visão Interna
                4.2.1.2.1. Camada de Apresentação
                4.2.1.2.2. Camada de Negócios
                4.2.1.2.3. Camada de integração
                4.2.1.2.4. Camada de Persistência
                4.2.1.2.5. Camada Adapter
                4.2.1.2.6. Classes comuns
        4.2.2. Montagem da aplicação (Application Assembly)
            4.2.2.1. Regras de dependências entre módulos
                4.2.2.1.1. Divisão em mais de uma Enterprise Application
                4.2.2.1.2. Divisão em mais de uma Web Application
    4.3. Visão de Implantação
    4.4. Visão de Integração
5. Mecanismos Arquiteturais
    5.1. Autenticação e Autorização

5.1.1. Requisitos de Segurança
5.1.1.1. Visão Geral da Solução
5.1.2. Identity Manager (IdM)
5.1.2.1. Modelo de Identidades (Identity Model)
5.1.3. Identity Provider (IdP)
5.1.4. Service Provider (SP)
5.1.4.1. Autenticação e Single Sign-On
5.1.4.2. Single Sign-Out
5.1.4.3. Autorização
5.1.4.3.1. RBAC - Role Based Access Control
5.1.4.3.2. ACL - Access Control List
5.2. Logging
5.2.1. Integração do Log4J 2 com o Wildfly
5.3. Auditoria
5.4. Transações
5.4.1. Conexão a múltiplos bancos de dados
5.5. Mecanismo de Tratamento de Exceções
5.5.1. Exceções de negócio e de sistema
5.6. Agendamento de Tarefas/Processamento Batch
5.6.1. Introdução
5.6.2. Agendamento de tarefas
5.7. Relatórios
6. APÊNDICES
6.1. Apêndice A. Construção
6.1.1. Ferramenta padrão para ciclo de vida de build
6.2. Apêndice B. Gestão de configuração
6.2.1. B.1. Gerenciamento de Versão
6.2.1.1. Versionamento de código
6.2.1.2. Versionamento de serviços
6.2.2. B.2 Integração contínua
6.3. Apêndice C. Automação de Controle de Qualidade e Testes
6.3.1. C.1. Introdução
6.3.1.1. Abordagem em Camadas
6.3.1.2. Testes contínuos
6.3.1.3. Unitário
6.3.1.4. Integração
6.3.1.4.1. Deployment com ShrinkWrap
6.3.1.4.2. Maven Profile: Remote x Managed
6.3.1.4.3. Controle Transações
6.3.1.4.4. Compartilhamento de Deployables
6.3.1.5. Testes Funcionais
6.3.1.5.1. Testes caixa-branca
6.3.1.5.2. Testes caixa-cinza

6.3.1.5.2.1. Diferença entre modos de execução Client e In-Container
6.3.1.5.2.2. Arquillian Warp
6.3.1.5.3. Testes caixa-preta
6.3.1.6. Aceitação (JBehave/Cucumber-JVM)
6.3.1.7. Cobertura de código Jacoco Eclemma
6.3.1.8. Vulnerabilidade
6.4. Apêndice D. QBE - Consulta Dinâmica de Entidades
6.4.1. D.1. Princípios Básicos do QBE
6.4.2. D.2. Implementando QBE: Principais Elementos
6.4.3. D.2.1. Exemplo
6.4.4. D.2.2. QBERepository
6.4.5. D.2.3. Filter
6.4.5.1. Configurando Operações e Operadores
6.4.5.2. Paginação da Consulta no Servidor
6.4.5.3. Otimização de Consultas
6.4.5.4. Ordenação do Resultado
6.4.5.5. Container - Agrupando Operações
6.5. Apêndice E. CRUD - Padronização de Solução para Cadastros

1. Introdução

Arquiteturas de Software são reconhecidas pelo seu importante papel na determinação da qualidade de sistemas de software. Quando a arquitetura de uma determinada solução de software é definida o que se busca é evitar problemas (ainda que futuros) relativos a desempenho, integração, segurança e até problemas que envolvam o ciclo de vida de desenvolvimento de software, tal como produtividade e a introdução de novas tecnologias (que pode impactar pela falta de conhecimento na equipe ou validação da tecnologia como solução).

Um problema adicional é que a própria arquitetura precisa ser validada, visto que ela própria se torna um risco ao projeto.

A arquitetura de software de um sistema precisa endereçar todos os seus requisitos funcionais e não-funcionais, que são específicos ao novo sistema. Mas alguns desses requisitos são comuns a uma série de sistemas. Neste ponto é que entra a Arquitetura de Referência.

A Arquitetura de Referência é composta por uma série de definições arquiteturais que podem ser usadas como base para a definição da Arquitetura de Software. A grande vantagem é que os principais requisitos da maior parte das aplicações já são endereçados pela arquitetura de referência e as soluções já foram validadas.

Este documento apresenta a Arquitetura de Software de Referência para aplicações desenvolvidas na plataforma Java Enterprise Edition (JEE). Nela são tratados os principais riscos e atributos de qualidade arquiteturais:

- Desempenho
- Integração
- Produtividade
- Segurança
- Escalabilidade
- Disponibilidade
- Ambiente
- Novas tecnologias
    - Skill
    - Validação

1.1. Objetivo

Este documento detalha as definições da Arquitetura de Software de Referência para a plataforma Java Enterprise Edition - JEE em aspectos arquiteturais e de projeto, baseando se nas melhores práticas de arquitetura. Este documento é um guia para a definição da Arquitetura de Software para as aplicações dos Tribunais envolvidos.

1.2. Escopo

A Arquitetura de Referência para a Plataforma JEE é o conjunto de recomendações de tecnologias, *frameworks*, componentes, padrões de projeto, ferramentas, guias e modelos que definem padrões a serem seguidos no desenvolvimento de sistemas dos Tribunais que a adotam, assim como aplicações e exemplos que demonstram o seu uso.

1.3. Definições, Siglas e Abreviaturas

JEE - Java Platform Enterprise Edition
RMI - Remote Method Invocation
SOAP - Simple Object Access Protocol
JSF - JavaServer Faces
XML - Extensible Markup Language
EJB - Enterprise JavaBeans
MVC - Model-View-Controller
XML - eXtensible Markup Language
EJB - Enterprise JavaBeans
POJO - Plain Old Java Objects

1.4. Referências

A arquitetura proposta é aderente aos seguintes guias e tecnologias. Qualquer software desenvolvido com base na arquitetura de referência deve ter aderência aos mesmos.

#	Nome	Localização
1	Guias de codificação Java	http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html https://google-styleguide.googlecode.com/svn/trunk/javaguide.html
2	Java Platform, Enterprise Edition 7 (Java EE 7)	http://jcp.org/aboutJava/communityprocess/final/jsr342/index.html
3	Java EE 7 Technologies	http://www.oracle.com/technetwork/java/javaee/tech/index-jsp-142185.html
4	The Java EE 7 Tutorial	http://docs.oracle.com/javaee/7/JEETT.pdf
5	JasperReports Library	http://community.jaspersoft.com/project/jasperreports-library

6	JasperSoft Studio	https://community.jaspersoft.com/project/jaspersoft-studio
7	Arquillian	http://arquillian.org/
9	RestEASY	http://resteasy.jboss.org/
10	Maven 3	http://maven.apache.org/ref/3.0/
11	Jenkins	https://wiki.jenkins-ci.org/display/JENKINS/Home
12	SonarQube	http://docs.sonarqube.org/display/SONAR/Documentation
13	GitLab	https://gitlab.com/gitlab-org/omnibus-gitlab/blob/master/README.md
14	PicketLink	www.picketlink.org

2. Direcionadores

Existem diversos direcionadores que guiaram as decisões da arquitetura de referência documentada aqui. A fim de manter claros os motivos que levaram a tais decisões, os direcionadores são discutidos abaixo. Vale ressaltar que nenhuma decisão é definitiva, sendo possível revê-la em caso de necessidade. É importante, nesse caso, que o direcionador originário de tal decisão seja devidamente analisado, minimizando os impactos.

2.1. Plataforma de Desenvolvimento Java

Uma das principais plataformas de desenvolvimento das últimas décadas é a plataforma Java. Por meio da plataforma JEE (Java Enterprise Edition) é possível desenvolver aplicações distribuídas e escaláveis, capazes de atender os requisitos de negócio.

A arquitetura descrita neste documento se baseia na versão 7 da Plataforma Java EE.

2.2. Infra-estrutura Tecnológica

Um dos princípios adotados para a arquitetura de referência é a neutralidade tecnológica. A arquitetura deve possibilitar independência em relação aos fabricantes de hardware e software, evitando assim a dependência de um único fornecedor, permitindo maior flexibilidade e capacidade de mudança.
Apesar dessa neutralidade há uma infra-estrutura já suportada e isso foi levado em consideração nas definições arquiteturais.

- Sistema Operacional Linux Red Hat

- Servidor de Aplicação JBoss AS, JBoss EAP e WildFly 8.2
- Banco de Dados Relacional Oracle (10g e 11g)

Essa infra-estrutura já suportada vai de encontro a outro direcionador que é a utilização efetiva de recursos de monitoramento e controle. Uma infra-estrutura já conhecida e de domínio da área de suporte da Justiça do Trabalho possibilita maior controle e monitoração dos ambientes. Não significa, entretanto, que qualquer componente da infra-estrutura já suportada não possa ser trocado com um esforço baixo ou até sem nenhum esforço de atualização na arquitetura.

2.3. Princípios Arquiteturais

Os princípios arquiteturais que guiaram as definições da arquitetura são variados. Alguns deles, de maior relevância para o escopo dos sistemas, são discutidos abaixo:

- A arquitetura serve às necessidades do negócio, garantindo que seus objetivos sejam atingidos, viabilizando melhoria do desempenho operacional;
- Aderência aos níveis de serviço requeridos: desempenho, disponibilidade, escalabilidade, estabilidade, confiabilidade, manutenibilidade, interoperabilidade, custos. São fatores essenciais em cada componente de arquitetura;
- Priorizar o uso de padrões abertos visando a neutralidade tecnológica. Por exemplo XML e Web Services (SOAP e REST) para compartilhamento de informações;
- Utilizar soluções e tecnologias comprovadas. Isso mitiga grande parte do risco de desenvolvimento e implementação, reduzindo impactos de custos e prazos que são menos previsíveis em componentes construídos sob medida.
- Arquitetura baseada em componentes com baixo acoplamento: As soluções devem ser desenhadas considerando um baixo grau de acoplamento e fronteiras bem definidas, facilitando o reuso, adição, remoção ou substituição de funções e componentes.

2.4. Front-End Web e suporte a dispositivos móveis

Grande parte das soluções de software já migraram da plataforma Desktop para a plataforma baseada na Web. Outra tendência é o suporte a dispositivos móveis, possibilitando o acesso às aplicações a partir de qualquer local.

Podemos destacar algumas vantagens das aplicações Web:

- facilidade de acesso;
- menor dependência de recursos do computador do usuário, incluindo hardware, sistema operacional e plugins;
- maior facilidade para disponibilizar atualizações;
- maior escalabilidade da aplicação vertical e horizontal (melhor uso dos recursos disponíveis e capacidade de incluir novos servidores).

Umas das preocupações da arquitetura aqui definida é o suporte à plataforma Web e aos dispositivos móveis. Uma vantagem é que esses dispositivos, como smartphone e tablets, já suportam os padrões Web, sendo necessário apenas a preocupação em construir aplicações

Web responsivas, que se adaptem a esses dispositivos. Essa característica é suportada pela arquitetura na sua camada de visão.

Algumas aplicações podem ser mais sensíveis e necessitar de aplicações que rodem nativamente nos dispositivos móveis. Para esses casos o suporte também é contemplado pela arquitetura. A divisão bem definida de camadas permite que a camada de visão seja substituída por uma aplicação nativa em um *smartphone* ou *tablet,* e que este se comunique diretamente com a camada de negócios da aplicação, utilizando os recursos da camada de integração (Web Services REST e mecanismos de segurança).

2.5. Acessibilidade

Acessibilidade significa incluir a pessoa com deficiência nas atividades, promovendo autonomia e independência. Por exemplo, prédios terem rampas para acesso com cadeira de rodas, banheiros adaptados. O computador e a Internet representam um enorme passo para a acessibilidade. Na visão do governo federal a inclusão digital é um dos caminhos para a inclusão social. Dessa forma, acessibilidade deve ser endereçado pela arquitetura.

Nos sistemas de informação, acessibilidade se refere principalmente às recomendações do WCAG (*World Content Accessibility Guide*) do W3C e no caso do Governo Brasileiro ao e-MAG (Modelo de Acessibilidade em Governo Eletrônico). O e-MAG consiste em um conjunto de recomendações a ser considerado para que o processo de acessibilidade dos sítios e portais do governo brasileiro seja conduzido de forma padronizada e de fácil implementação. Ele é coerente com as necessidades brasileiras e em conformidade com os padrões internacionais. Seu objetivo é orientar profissionais que tenham contato com publicação de informações ou serviços na Internet a desenvolver, alterar e/ou adequar páginas, sítios e portais, tornando-os acessíveis ao maior número de pessoas possível. O e-MAG está atualmente na versão 3.1, revisada.

Outra especificação que está sendo considerada na arquietura para o tema acessibilidade é WAI-ARIA 1.1. Ela provê features para definir elementos de interface com o usuário acessíveis com o objetivo de melhorar a acessibilidade e interoperabilidade de aplicações e conteúdo web.

Sobre o tema Acessibilidade, vale ainda ressaltar que esta característica foi um dos motivadores para a adoção de Java EE 7. A especificação JSF 2.2 com o uso de PrimeFaces já possui recursos HTML5 aprimorados para Acessibilidade.

2.6. Tendências futuras

- Adoção das práticas de arquitetura orientada a serviços (SOA);
    - Adoção de um barramento de serviços (ESB - *Enterprise Service Bus*)
- Suporte a adoção de Sistemas de Gerenciamento de Processos de Negócio (BPMS - Business Process Management System);

- Suporte a adoção de Sistemas de Gerenciamento de Regras de Negócio (BRMS - Business Rules Management System).

3. Requisitos Inversos

- Não existe a necessidade de internacionalização na camada de apresentação e nem na camada de persistência. Todos os dados são tratados e apresentados no idioma português do Brasil;

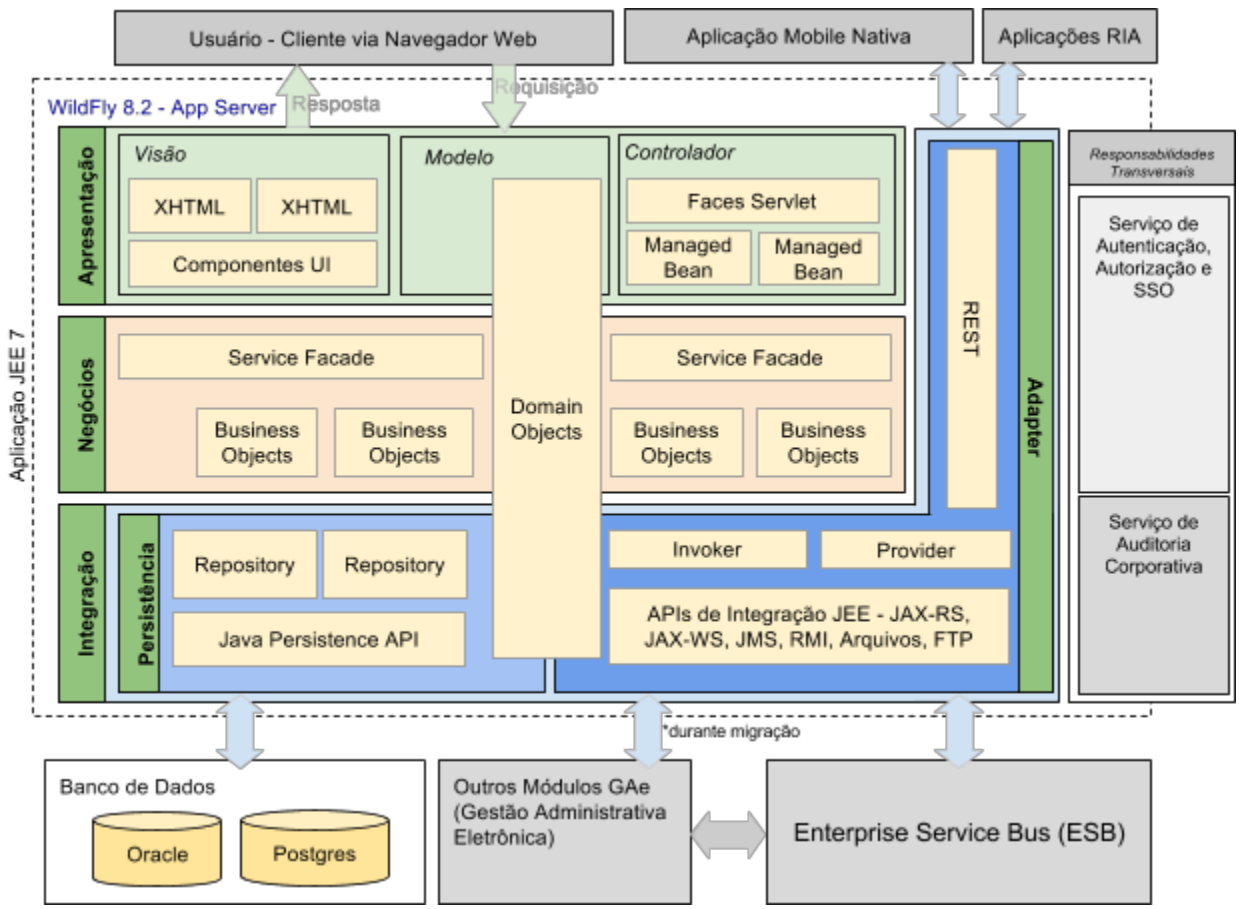
4. Visão Geral da Arquitetura

Esta seção será expandida para detalhar as soluções e padrões de projeto (*design patterns*) adotados para cada camada das aplicações JEE. O objetivo é mostrar as camadas, frameworks e padrões que compõem a solução e como eles se relacionam.

As metodologias atuais de documentação de arquiteturas estruturam as definições arquiteturais em Visões, onde cada visão documenta a arquitetura por uma perspectiva distinta. Atualmente, a metodologia mais reconhecida no mercado é o modelo “4+1 View”, que é adotada nas seções seguintes.

4.1. Visão Lógica

Nesta seção é apresentada a arquitetura lógica a ser adotada para as soluções de softwares a serem desenvolvidos pela Justiça do Trabalho. A figura abaixo é uma representação lógica da arquitetura, destacando os componentes de cada camada.



A arquitetura foi elaborada visando estruturar os diferentes processos de negócio em serviços interoperáveis. Para atender às necessidades dos sistemas e agregar valor ao negócio. a solução é baseada em padrões abertos que podem ser combinados e reutilizados. Além disso, a arquitetura se baseia em um conjunto consistente das melhores práticas para construção de software e modelagem utilizando conceitos de Arquitetura Orientada a Serviços e Desenvolvimento baseado em Componentes.

A definição das versões das tecnologias apresentadas é tão importante quanto a definição da própria tecnologia, pois existe um conjunto de produtos que se integram apenas com combinações de versões específicas.

4.1.1. Visão de Camadas

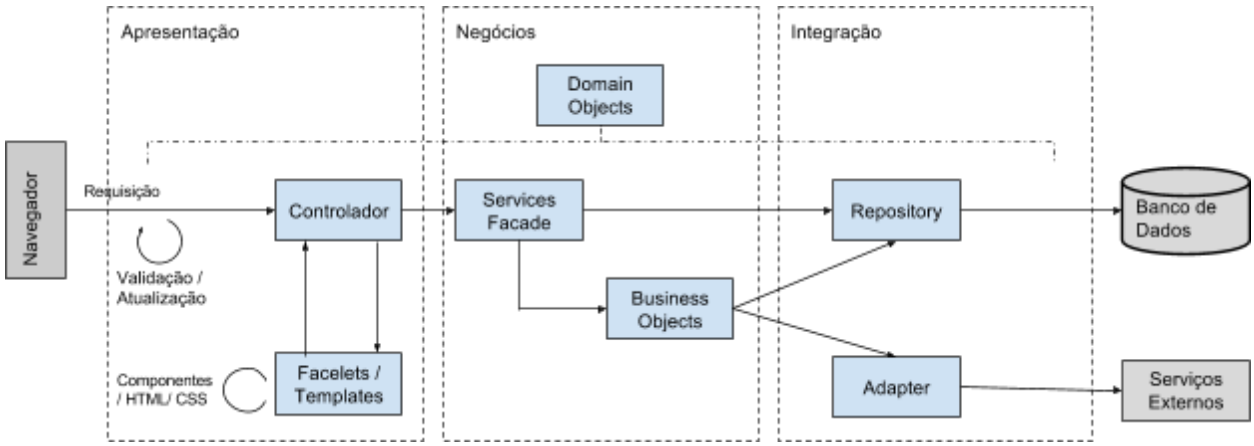
A arquitetura de referência é baseada em uma estrutura com múltiplas camadas que podem potencialmente ser implantadas em servidores físicos separados, ou seja, a arquitetura é *n-layer* e *n-tier*.

Uma camada deve se comunicar apenas com as camadas adjacentes, e a comunicação entre camadas não adjacentes deve ser evitada. Objetos de domínio, que representam a informação do domínio do sistema, podem trafegar através das camadas da arquitetura.

Para manter a aplicação coerente e suas camadas isoladas, alguns pontos são importantes:

- Cada camada se comunica apenas com a camada adjacente, exceto os casos permitidos e documentados abaixo;
- Injeção de dependência é utilizada para instanciar os objetos dentro das camadas;
- As camadas desconhecem a implementação (interna) de outras camadas e se comunicam apenas pelas interfaces definidas;
- Os objetos de domínio que são utilizados para trafegar dados entre as camadas não tem regras de negócio ou lógicas de acesso a dados, podendo ser utilizados em qualquer camada da aplicação. Eles são apenas Beans (POJOs), anotados conforme as necessidades de cada camada.

Para ficar mais claro a comunicação entre camadas, a figura abaixo exemplifica a interação entre elas e entre seis componentes, durante o atendimento de uma requisição de um usuário utilizando um navegador Web.



4.1.1.1. Camada de Apresentação (Presentation tier)

Essa camada contém serviços que serão usados pelas aplicações para interagir com os usuários do sistema. A arquitetura de referência é baseada em uma solução Web e por isto esta camada é responsável pelo tratamento das ações vindas dos usuários através de um browser assim como a interação de mensagens, validações de dados, respostas e erros.

- **Validator**: Dentro do fluxo do framework utilizado e descrito a seguir, a validação de dados enviados para o Controller é feita antes mesmo da chamada da ação em si. Desta maneira, se existir alguma inconsistência nestes dados o usuário já é notificado para o ajuste em tela antes de executar uma ação no sistema com dados inconsistentes. A validação fará uso da API Bean Validation 1.1 da Java EE, onde, conforme definido nesta arquitetura de referência, as anotações de validação são incluídas na camada de negócio.

- **Controller**: Ao acionar um evento na aplicação, tal como o acionamento de uma funcionalidade ou a requisição de uma página, a aplicação aciona ações de um determinado Controller para o tratamento daquela funcionalidade ou requisição. Este controller é responsável por tratar esta requisição e determinar se algum serviço da aplicação deve ser acionado e responder à esta requisição de acordo com o fluxo e navegação.

- **Páginas XHTML**: Após os dados recebidos e corretamente manipulados o Controller redireciona para uma tela onde são montados os componentes de tela, layout e usabilidade de acordo com templates definidos para aquele fluxo. Os dados são obtidos do Controller e renderizados para a visualização do usuário.

Contemplando as necessidades desta camada são propostos os seguintes frameworks e versões:

Framework	Versão/ Release	Responsabilidade
JSF	2.2	Especificação do Java Server Faces
JSF - Facelets	2.2	Criação de templates de páginas para exibição
JSF - FacesFlow	2.2	Gerenciador de fluxos e escopo de conversação para a aplicação. Utilizado para telas de assistente, por exemplo.
Primefaces	5.1	Implementação da especificação JSF com o funcionamento e componentes padrões, componentes JSF de usabilidade rica e AJAX
CDI	1.1	Injeção de Dependências
Bean Validation	1.1	API JEE responsável pela validação de objetos de negócios e entidades
WAI-ARIA	1.0	Especificação W3C que define uma maneira de tornar o conteúdo Web e aplicações Web mais acessíveis a pessoas com deficiência.
e-MAG	3.1	Especificação do governo brasileiro sobre acessibilidade.

4.1.1.2. Camada de Negócios

A camada de negócios contem os serviços que executam as regras de negócio envolvidas nas ações do usuário que não estão relacionadas com telas, fluxos ou consistência dos dados e sim na lógica implementada para que a funcionalidade seja executada. Contém todos os objetos de negócio e as regras de negócio, simples ou complexas.

É a camada de negócio que acessa a camada de integração ou de persistência para obter dados que serão manipulados, processados e, quando acionada pelo usuário, retornados para a camada de apresentação.

Esta camada cuida do contexto transacional, ou seja, a chamada de métodos de objetos pertencentes a esta camada abrem transações (no contexto do banco de dados ou do servidor de aplicações, dependendo da configuração).

Os objetos desta camada estão divididos em três tipos:

- **Service Facade**: Definem as interfaces de cada componente de negócio a serem acessados por componentes externos para invocar as regras de negócio. São também as fachadas para as regras implementadas. Ou seja, as classes responsáveis por serem acessadas pelas interfaces dos componentes de negócio e executar as regras associadas às entidades ou fluxos. São elas que tratam exceções, controlam as transações e transformam os dados para resposta. Quando uma fachada se tornar complexa esta pode ser dividida em dois objetos.
- **Business Objects**: São os objetos que representam regras de negócios propriamente ditas e implementam regras de negócio mais complexas que as regras contidas em CRUDs (por exemplo). Elas são invocadas pelas fachadas para separar as regras de negócio complexas de um certo fluxo de tratamento, controle de negócio ou outra atividade que não são regras de negócio em si. Devem ser organizados de forma a aglutinar os métodos que atuem sobre os mesmos conjuntos de entidades de negócio. Por exemplo: *FolhaPagamentoBO*, representa os procedimentos utilizados para o cálculo de um sistema de folha de pagamento, mas não necessariamente existirá uma entidade equivalente na camada de persistência. As Business Objects não devem ser acessadas diretamente por outras camadas, devendo sempre ter uma Service ou uma Facade para acessá-la e cuidar do tratamento de exceções, transações e transformação de dados para a resposta.
- **Domain Objects**: representam as entidades de negócio (independente das entidades de banco de dados, gerenciados pela camada persistência). Os Domain Objects são Beans e não devem conter as regras de negócios que operam sobre as entidades em si, podendo ter apenas métodos simples.

Framework	**Versão/ Release**	**Responsabilidade**
CDI	1.1	Injeção de Dependências
EJB	3.2	Objetos de negócio distribuídos com suporte a transações.

4.1.1.3. Camada de Integração (Integration tier)

Esta camada é responsável pelos serviços e configurações que permitem a comunicação integrada com outros sistemas envolvidos, seja consumindo ou provendo informações. Entenda-se por comunicação qualquer operação invocada em um serviço remoto, uma aplicação adjacente. A Camada de Integração deve ser a única responsável por concretizar este tipo de comunicação com o "ambiente externo", realizando também todas as transformações, controles, tratamentos de mensagens ou qualquer outro comportamento inerente aos detalhes de API de integração do serviço acessado.

Os sistemas adjacentes podem ter diferentes naturezas e objetivos, mas geralmente são voltados para operações de persistência de informações, consulta de dados ou realização de regras de negócio. Duas sub-camadas foram definidas para esclarecer os comportamentos destes sistemas, além das tecnologias e padrões de projetos utilizados na implementação, de fato, de sua integração:

4.1.1.3.1. Sub-camada: Persistência

A sub-camada de Persistência tem como principal objetivo orquestrar o processo de persistência e recuperação de dados a partir de um SGBD (tipo de fonte de dados naturalmente estabelecida nesta definição de arquitetura). Sua grande responsabilidade é tornar transparente ao restante da aplicação, os detalhes técnicos necessários referentes à todo o procedimento de estabelecimento de conexão, operação e controle de qualquer comunicação realizada com a base de dados.

Considerando que serão utilizados Bancos de Dados Relacionais e que a linguagem Java é orientada a objetos, as melhores práticas adotadas no mercado indicam a utilização de um framework ORM (*Object-Relational Mapping*), permitindo a realização transparente de operações relacionais através de uma interface orientada a objetos.

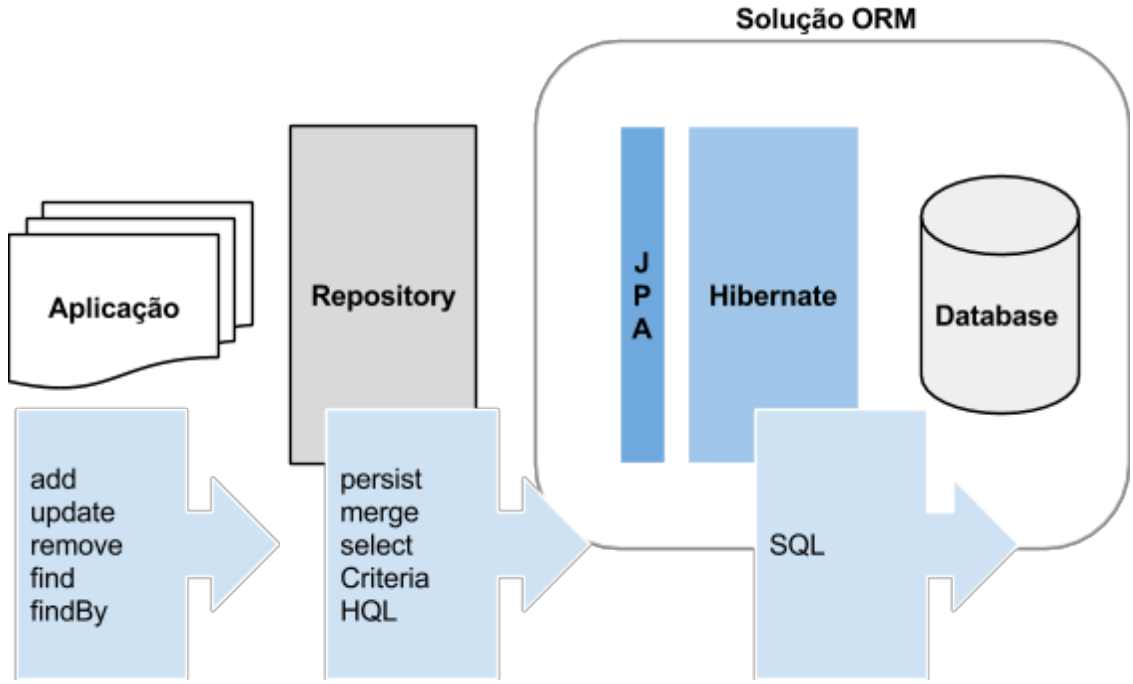
A Plataforma JEE disponibiliza uma especificação amadurecida e com considerável adoção pelo mercado corporativo, trata-se do JPA (Java Persistence Api). A sua utilização traz importantes vantagens no estabelecimento de uma identidade arquitetural, destacando-se o fato de que, por ser uma solução nativa e indicada na arquitetura de referência do JEE, permitirá uma integração natural com as demais tecnologias absorvidas nesta proposta, pelo mesmo conceito. Para implementação da especificação JPA, sugere-se o uso do Hibernate, framework também bastante conceituado no mercado, sendo, inclusive, precursor da atual especificação.

O JPA disponibiliza uma API bastante completa para realização de operações sobre uma base de dados, através da interface ***java.persistence.EntityManager***. O métodos disponibilizados têm fins genéricos, para que possam atender a diversos tipos de solicitações, representando uma grande evolução em relação à antiga API de acesso *JDBC* (*Java DataBase Connectivity*). Visto que a interface *EntityManager* não é facilmente extensível e não foi projetada com os conceitos de *GENERICS* da linguagem Java, foi adotado o padrão de projeto ***Repository*** para encapsular e personalizar as operações de persistência e consulta, de acordo com as entidades de domínio da aplicação.

O padrão *Repository* implementa uma camada semântica de forma que ofereça uma API de acesso aos dados como se estes estivessem organizados na forma de um “conjunto/sacola de objetos”. Isto permite padronizar a maneira de manusear e consultar qualquer dado da aplicação através de operações (métodos) padrões do tipo *add, remove, find, findBy, update,*

etc. A figura abaixo demonstra a interação que ocorre entre a aplicação e a sub-camada de Persistência:

Interação entre a aplicação e a sub-camada de persistência.



Para realização do conceito de *Repositório de Entidades* foi adotado o framework Apache DeltaSpike, através de um dos seu módulos: *Data Module* (http://deltaspike.apache.org/documentation/data.html). Este módulo disponibiliza uma solução dinâmica com diversos comportamentos e operações pré-estabelecidos. O seu funcionamento básico ocorre através da interface ***EntityRepository<Tipo, Pk>***, que define diversas operações básicas de persistência sobre uma entidade de domínio. A implementação é gerada dinamicamente, de acordo com o tipo da entidade informada, e é possível através da ação de um interceptador *CDI* acionado através da anotação ***@Repository(forEntity=Entidade.class)***.

No exemplo abaixo, quando for solicitado ao container CDI uma instância de *ServidorRepository*, o interceptador entra em ação e implementa dinamicamente os métodos da interface *EntityRepository*.

```
@Repository(forEntity = Servidor.class)
public abstract class ServidorRepository extends EntityRepository<Servidor, Long> {
}
```

Outros métodos podem ser adicionados à classe *ServidorRepository* e, quando seguirem um padrão pré-estabelecido pelo *DeltaSpike*, também serão implementados dinamicamente,

conforme o código abaixo (outros exemplos e detalhes podem ser consultados na documentação do *DeltaSpike Data Module*):

```
@Repository(forEntity = Servidor.class)
public abstract class ServidorRepository extends  EntityRepository<Servidor, Long> {
	/**
	 * @return Todos os servidores com o nome igual ao informado
	 */
	public abstract List<Servidor> findByNome(String nome);

	/**
	 * @return Servidor com o cpf informado
	 */
	@Query("select s from Servidor s where s.cpf = ?1")
	public abstract Servidor findByCpf(String cpf);
}
```

Visto que o *Repository* encapsula a solução *ORM*, as classes deste pacote deverão ter livre acesso ao *EntityManager* para implementar operações que não podem ser alcançadas com as demais já disponibilizadas. Vale a pena ressaltar que, por via de regra, um ***EntityManager não deve ser injetado em qualquer outra camada acima da camada de Persistência***. Nos casos onde for necessário o acesso à base de dados, este deverá ser realizado através da disponibilização de um objeto específico nesta camada.

Para obter uma instância do *EntityManager*, pode-se fazer uso do mecanismo de injeção de dependência do CDI (desde que um *CDI Producer* compatível tenha sido definido), como no trecho abaixo:

```
@Inject
private EntityManager entityManager;

public Servidor get(Long id) {
	return entityManager.find(Servidor.class, id);
}
```

Os seguintes Frameworks e versões foram validados nesta proposta:

Framework	**Versão/ Release**	**Responsabilidade**
JPA	2.1	Especificação *JEE.*
Hibernate	4.3.5.Final	Implementação da especificação *JPA.*

Apache DeltaSpike - Data Module	1.1.0	Implementação de recursos dinâmicos para persistência e recuperação de dados. Implementação do padrão *Repository*.

4.1.1.3.2. Sub-camada: Adapter

Compondo a cama da de integração, a sub-camada Adapter é responsável por acessar outros sistemas ou disponibilizar informações para estes sistemas. Serviços são criados que encapsulam a tecnologia deixando transparente a integração. Essa camada possui os compoentes **Provider** e **Invoker**, que são responsáveis por realizar conversões e transformações de dados, respectivamente para expor e prover serviços.

Os frameworks adotados para a realização dessa camada e validados para a Arquitetura de Referéncia são:

Framework	**Versão/ Release**	**Responsabilidade**
JAX-WS	2.2	Especificação *JEE*
JAX-RS	2.0	Especificação *JEE*
RestEASY	3.0.10Final	Implementação da especificação *JAX-RS*
JMS	2.0	Especificação *JEE*
JAXB	2.2.5	Bindind XML

4.1.1.3.2.1. Serviços REST

Na Sub-camada Adapter, a implementação de serviços REST é fornecida pelo RestEASY. Sempre que possível deverão ser implementados serviços RESTfull, facilitando a integração com frameworks como angular.JS e aplicaçoes mobile nativas.

Alguns padrões para implementação de serviços, tais como padrões para serialização e deserialização de objetos de e para, *routes* e sua semântica nas APIs, entre outros padrões, são recomendados.

Para permitir serialização e deserialização transparent de/para JSON, que é o padrão de formato adotado neste sistema para formato de dados para serviços REST, é necessário o uso de algumas anotações nas classes de domínio:

```
@Entity
@JsonAutoDetect(fieldVisibility=Visibility.NONE, getterVisibility=Visibility.PUBLIC_ONLY,
isGetterVisibility=Visibility.NONE)
public class Pessoa implements Serializable {
.
.
```

Como pode ser visto, nenhum *field* é serializado automaticamente em JSON a não ser que tenha um *getter* público.

No caso de um field que possui relacionamento um para muitos (1..N) com outra entidade, para correta serialização é necessário anotar essa relação com `@JsonManagedReference` como no exemplo abaixo:

```
@Entity
@JsonAutoDetect(fieldVisibility=Visibility.NONE, getterVisibility=Visibility.PUBLIC_ONLY,
isGetterVisibility=Visibility.NONE)
public class Pessoa implements Serializable {
    .
    .
    .
    @JsonManagedReference
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "pessoa", orphanRemoval = true)
    private Set<Telefone> telefones = new HashSet<Telefone>();
```

e na classe referenciada fazer a referência reversa com `@ManyToOne`:

```
@Entity
@JsonAutoDetect(fieldVisibility = Visibility.NONE, getterVisibility = Visibility.PUBLIC_ONLY,
setterVisibility = Visibility.NONE)
public class Telefone implements Serializable {
    .
    .
    .
    @JsonBackReference
    @ManyToOne
    @JoinColumn(nullable = false)
    private Pessoa pessoa;
```

Para dar suporte ao uso de HATEOAS, os métodos das classes de serviço devem ser anotadas para permitir links para recursos associados ao objeto retornado (vide http://tools.ietf.org/html/rfc5988). Isso facilita a navegação entre os recursos interrelacionados e

também a documentação da API. Também recomendado que o ObjectWriter utilizado pelo serviço esteja configurado para para gerar JSON formatado. Com a compressão de dados o ganho na alocação de banda é inferior a 5% entre a resposta formatada e desformatada, por isso prefira a resposta formatada. Isso pode ser feito dentro das subclasses de Application

As convenções de verbos de acordo com o protocolo HTTP devem obedecidas, ou seja, GET para recuperar recursos, POST para criar um recurso, PUT para alterar um recurso, DELETE para remover um recurso. Os retornos esperados são os listados abaixo:

Verbo HTTP	**Coleção (ex.: /pessoas)**	**Item específico (ex.: /pessoas/{id}**
GET	200 (OK) 400 (Bad Request) quando erro nos dados da requisição enviada pelo cliente (ex.: versão da API inexistente)	200 (OK) quando um item e retornado 404 (Not Found) quando não encontrado 400 (Bad Request) quando erro nos dados da requisição enviada pelo cliente
PUT	404 (Not Found) a não ser que as regras de negócio permitam alterar todos os itens da coleção, o que no caso resultaria em 200 (OK)	204 (No Content) quando item alterado 404 (Not Found) quando id do item for inválido
POST	201 (Created), field 'Location' do header com o caminho para o recurso com o id recém criado (ex.: /pessoas/{id})	404 (Not Found) - nenhuma ação é executada
DELETE	404 (Not Found) , a não ser que haja regra de negócio que permita a remoção de toda a coleção	200 (OK) quando remoção tiver sucesso 404 (Not Found) quando id inválido ou não encontrado

Cada classe de serviço deverá implementar acesso a somente um tipo de recurso, assim uma dada URL retornará somente objetos de uma mesma classe de domínio. O tipo de dado é definido no serviço através do *header Accept*. A definição do tipo retornado via content-type se dá através do mecanismo descrito na RFC 4288 sessão 3.2 (https://tools.ietf.org/html/rfc4288#section-3.2).

Recursos podem ser aninhados, como por exemplo: /pessoas/1234/telefones?tipo=Celular

Para resolver o problema de identificação da versão da API por quem a consome existem três maneiras: versionamento por *URL*, versionamento através de *header* customizado e versionamento por *content-type*. Cada uma das três tem suas vantagens e desvantagens.

O versionamento por URL, apesar de ser mais prático, é semanticamente incorreto e portanto não será utilizado. O versionamento por *header* customizado é pouco prático e fora de qualquer padrão/RFC e também não ser adotado. Assim restou o versionamento por *content-type*.

O versionamento por content-type se dá através do mesmo mecanismo usado para definir o tipo de dado que será retornado.

Por padrão também será adotada a compressão dos dados transmitidos com o algoritmo GZIP. Isso se torna possível anotando a classe que implementa o serviço com `@GZIP`.

Um exemplo de *resource* disponibilizado de acordo com as definições acima seria:

```
@GZIP
@Transactional
@Produces("application/vnd.br.jus.jt.gae.pessoas+json;version=2;charset=UTF-8")
@Consumes("application/vnd.br.jus.jt.gae.pessoas+json;version=2;charset=UTF-8")
@Path("/pessoas")
public class PessoasRS {
    .
    .
    .
    @AddLinks
    @LinkedResources({
        @LinkResource(value = Pessoa.class, rel = "telefones"),
        @LinkResource(value = Telefone.class)
    })
    @GET
    @Path("/{id: \\d+}/telefones")
    public List<Telefone> getTelefonesPessoa(
            @QueryParam("tipo") String tipoTelefone) {
        .
        .
        .
    }
```

Um dado recurso pode suportar múltiplas versões de uma api, bastando anotar de forma adequada (aceitando múliplos `@Consumes`, por exemplo).

Em relação à documentação da API, ela no deve ser feita em documento estático. Ou é publicada em uma URL ou é auto-documentada via HATEOAS. A geração da URL de forma automatizada pode ser feita a partir do processamento do código fonte das classes de serviço. Por isso é de grande importância que os comentários JavaDoc estejam atualizados e corretos.

A geração do site de documentação pode ser feito pela ferramenta compatível com Swagger (http://swagger.io/).

4.2. Visão de Implementação

4.2.1. Pacotes e Classes

Os pacotes são o primeiro nível de organização física do sistema. Definem como as classes que implementam os diversos componentes da solução devem ser agrupadas de acordo com o seu papel no sistema. A estrutura de pacotes de um projeto componentizado pode ser observada de dois pontos de vista: externa e interna. Ambos são orientados pelos componentes de negócio da aplicação.

A visão **externa** é a visão de componentes de negócio da aplicação, mapeada para os seus respectivos pacotes do sistema. A visão **interna** é a visão da estrutura interna de cada componente de negócio independentemente.

4.2.1.1. Visão Externa

Os componentes de negócio devem aparecer na hierarquia de pacotes antes da separação das camadas (que aparecerão na visão interna). A estrutura de pacotes deve ser definida da seguinte maneira:

```
br.jus.jt.<área_funcional>.<sistema>.<subsistema/módulo>[.<aplicação_usuario>][.
<componente>[.<sub-componente>]]
```

- <área_funcional> : sigla que representa a área funcional ou unidade de negócio onde o projeto é desenvolvido;
- <sistema> : sigla do sistema;
- <subsistema/módulo> : nome do subsistema ou módulo;
- <aplicação_usuário> : opcionalmente utilizada para casos em que há necessidade de separar o código desenvolvido, por exemplo, no lado do servidor do código desenvolvido no lado do cliente, por serem implantados em locais diferentes
- <componente> : nome do componente de negócio
- <sub-componente> : nome do sub-componente do negócio

Um exemplo é **`br.jus.jt.rh.sigep.folha.contracheque.mensal`**

Para os nomes de pacotes todos os caracteres devem ser em letra minúscula e o número total de caracteres na estrutura de pacotes externa se recomenda não ultrapassar 100 caracteres.

Apesar de não haver limitação da linguagem Java, algumas ferramentas de gerência de configuração e versões antigas de sistema operacional podem ter problemas com nomes e arquivos que excederem 255 caracteres. Lembrando que o nome de um item de configuração será composto pelo nome de pacote externo seguido pelo nome de pacote na visão interna e pelo nome da classe Java.

4.2.1.2. Visão Interna

Dentro de cada componente de negócio, as classes devem ser organizadas segundo o seu papel no sistema. Assim a estrutura de pacotes, do ponto de vista interno aos componentes de negócio, está fortemente ligada à estrutura lógica da arquitetura.

4.2.1.2.1. Camada de Apresentação

- `presentation` - pacote que contem todos os itens da camada de apresentação;
- `presentation.jsf` - pacote para as classes que implementam as ações de tratamento de eventos da camada de apresentação para Java Server Faces.
    - O nome dos *Managed Beans* deve terminar com o sufixo *Controller*. (ex: Pessoa*Controller*)

4.2.1.2.2. Camada de Negócios

- `business` - pacote que contém as classes de serviços de negócio do componente funcional
- `business.model` - pacote que receberá as classes de domínio do projeto não relacionados às tabelas ou entidades persistidas;
- `business.model.entity` - pacote que receberá as classes de domínio que representam entidades persistidas, referentes às tabelas do banco de dados;
    - Nome da entidade de negócio, sem qualquer prefixo ou sufixo. (Ex: Cliente, Pedido, Fornecedor);
- `business.service`: pacote que receberá todas as fachadas das classes que contém as regras de negócio;
    - As fachadas dos serviços de negócio devem ter o sufixo ServiceFacade (Ex.: ServidorServiceFacade)
- `business.service.bo`: pacote que receberá as classes que contém as implementações das regras de negócio, associadas a uma Service e as implementações de Business Objects para regras de negócio complexas.
    - As classe que implementam Business Objects devem ter o sufixo BO (Ex.: ServidorBO);

4.2.1.2.3. Camada de integração

- `integration` - pacote que receberá todas as classes de integração que gerenciam a conexão com todos os sistemas externos, incluindo banco de dados e serviços SOAP e REST;

4.2.1.2.4. Camada de Persistência

- `integration.persistence` - pacote que receberá interfaces dos padrões DAO e Repository, que serão fachadas para acessar banco de dados ou outro tipo de armazenamento voltado à persistência de dados da aplicação;
    - O uso dessas interfaces na camada de persistência dever ser considerado apenas quando houver possibilidade de mais de um meio de armazenamento. Quando o requisito de negócio indicar apenas persistência em banco de dados não existe necessidade de criar essa fachada, visto não haver necessidade de independência da implementação de persistência.
- `integration.persistence.jpa` – pacote que receberá as classes que utilizam JPA como persistência
    - Todas as classes deverão utilizar o sufixo *Jpa*;
    - As classes que utilizam o padrão DAO deverão o nome do *Entity* e o sufixo *Dao* (Ex.: Pedido*DaoJpa*);
    - As classes que utilizam o padrão Repository deverão utilizar o sufixo *Repository* após o nome da *Entity* (Ex.: Pedido*RepositoryJpa*).

4.2.1.2.5. Camada Adapter

- `integration.adapter.remote`: receberá a implementação concreta das classes que fazem o uso do RMI para invocar ou prover serviços para o sistema remoto;
- `integration.adapter.ws`: receberá a implementação concreta das classes que fazem o uso dos webservices do tipo SOAP para invocar ou prover serviços para o sistema remoto;
- `integration.adapter.rest`: receberá a implementação concreta das classes que fazem o uso dos webservices do tipo REST para invocar ou prover serviços para o sistema remoto;
- `integration.adapter.rest.service` - pacote que receberá todas as classes de integração que implementam serviços RESTful;
- `integration.adapter.rest.config` - pacote que receberá todas as classes de integração que implementam *middleware* para os serviços RESTful, por exemplo, implementações de `javax.servlet.Filter` (para implementar um *CORS* filter, por exemplo), `javax.ws.rs.ext.ExceptionMapper` (para definir *responses* padronizados para quando ocorrem certas *Exceptions)* ou ainda classes que estendam `javax.ws.rs.core.Application`

- `integration.adapter.rest.model` – pacote que receberá as classes de domínio do serviço (que não sejam de domínio do negócio e que não estejam relacionadas às tabelas ou entidades persistidas);
- `integration.adapter.messaging.jms`: receberá a implementação concreta das classes que fazem o uso de mensageria JMS para invocar ou prover serviços para o sistema local ou remoto;

Em cada um dos pacotes acima, quando houver necessidade de implementação de regras de conversão, transformação, extração de dados (casos em que os objetos de domínio da aplicação não podem ser utilizados diretamente na integração) deverão ser criados os seguintes pacotes:

- `invoker` – para as classes utilizadas pelos serviços para invocar recursos externos. Deverá ser utilizado o nome da integração associado juntamente com o sufixo Invoker (Ex. SegurancaInvoker);
- `provider` – para as classes que expõem os serviços da aplicação para outras aplicações. Deverá ser utilizado o nome do serviço exposto associado com o sufixo Provider (Ex. EgestaoProvider)

4.2.1.2.6. Classes comuns

`util` - pacote que contém classes utilitárias do componente de negócio. Deve ser usado com cautela pois pode virar um pacote genérico demais. Como o nome diz são classes que não tem ligação direta com o negócio e são apenas utilitárias. Em versões futuras da aplicação são classes que poderiam ser direcionadas para um framework utilitário.
`exception` - pacote que contém classes que representam erros de negócio, conforme orientações do mecanismo de exceções.

4.2.2. Montagem da aplicação (Application Assembly)

A especificação JEE 7 especifica que a plataforma Java EE suporta três tipos de unidades de implantação:

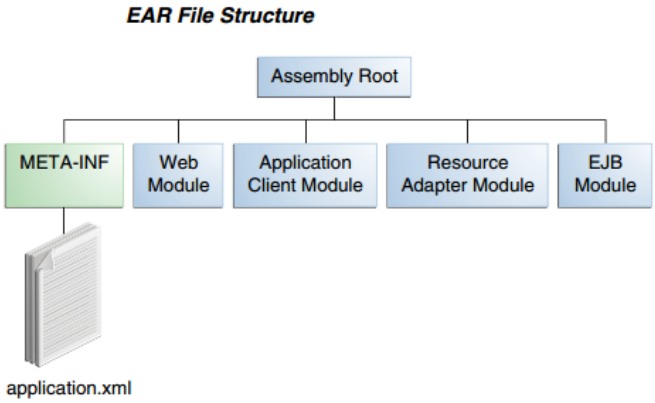
- Módulos Java EE *standalone*;
- Aplicações Java EE, consistindo de um ou mais módulos Java EE;
- Bibliotecas de classes empacotadas como arquivos .jar, que se tornam bibliotecas instaladas.

Os módulos Java EE consistem de um ou mais componentes Java EE e um arquivo XML de configuração correspondente, denominado "*deployment descriptor*", que contém os parâmetros da implantação de cada componente incluído no módulo. Os módulos JEE são:

- Módulo EJB, que contém arquivos de classes de Enterprise Beans e, opcionalmente, um descritor de implantação EJB. Módulos EJB são empacotados no formato de arquivos JAR com a extensão .jar.

- Módulo Web, que contem arquivos de classe Servlets, arquivos web, classes de suporte, arquivos GIF e HTML e, opcionalmente, um descritor de implantação Web. São empacotados no formato de arquivos JAR com a extensão .war.
- Módulo Aplicação Cliente que contem arquivos de classe e, opcionalmente, um descritor de implantação de aplicação cliente. Módulos deste tipo são empacotados no formato de arquivos JAR com a extensão .jar.
- Módulo Resource Adapter, que contem interfaces Java, classes, bibliotecas nativas e, opcionalmente, um descritor de implantação. Juntos eles implementam um adaptador JCA - Java Connector Architecture. São empacotados no formato de arquivos JAR com a extensão .rar (*resource adapter archive*)

Os módulos JEE podem ser implantados independentemente em um servidor de aplicações ou podem ser agrupados em um "Enterprise Application Archive" (arquivo ".ear"), que contém um conjunto de um ou mais módulos, com um deployment descritor ("application.xml") que descreve como os módulos compõem a aplicação.



Esta Arquitetura de Referência apresenta uma estrutura padrão para empacotar uma "web application" (.WAR) e para empacotar esta "web application" em um "Enterprise Application Archive" (.EAR). **A recomendação é sempre implantar uma aplicação utilizando um .EAR.**

4.2.2.1. Regras de dependências entre módulos

A especificação JEE define a seguinte hierarquia de *classloader* num servidor de aplicação:

1. *Java Classloaders* – são os classloader que contém as classes necessárias para o funcionamento básico da JVM (também conhecidas como classes de “bootstrap”), as classes de extensão (localizadas abaixo do diretório <JAVA_HOME>/bin/ext) e as classes do classloader do sistema (localizadas nas entradas definidas na variável de ambiente “CLASSPATH”);
2. *Application Server Extensions Classloader* – é o classloader que contém as classes proprietárias dos servidores de aplicação, necessárias para o funcionamento desses.

3. *Applications Classloaders* – são os classloaders de cada uma das enterprise applications implantadas no servidor. Cada EAR tem um único classloader para todos os JARs, RARs e EJBs implantados no nível da *Enterprise Application*.
4. *Web Applications Classloaders* – são os classloaders de cada uma das *web applications* implantadas no servidor. Cada *web application* tem o seu próprio classloader, independente do classloader da enterprise application a qual pertença.

De forma geral, os classloaders de nível inferior costumam delegar aos seus "pais" a instanciação de uma classe, antes de verificarem em si mesmos se possuem ou não a classe solicitada. Os servidores de aplicação, apesar de suportarem a hierarquia e o comportamento acima descritos, costumam oferecer recursos que permitem alterar ou influenciar essas características. Entretanto, a fim de manter o máximo de independência do servidor de aplicação e aderência a especificação JEE, qualquer configuração específica modificando o comportamento do classloader deve ser evitada.

Dentro de uma Enterprise Application e de suas Web Applications é comum a existência de bibliotecas (JARs) que são usadas em mais de um nível da aplicação (ex: biblioteca de log, usada pela camada de apresentação e de negócio de um determinado componente). Nessas situações, devem ser observadas as seguintes regras:

- JAR é usado por apenas uma web application
    - Deve ser colocado dentro do diretório "WEB-INF/lib";
- JAR é usado por mais de uma Web Application, dentro da mesma Enterprise Application
    - Deve ser colocado no diretório "lib" da Enterprise Application e deve ser referenciado via arquivo MANIFEST.MF das web applications.
- JAR é usado por uma ou mais Web Applications e por um ou mais EJBs ou JARs de componentes de negócio
    - o tratamento é igual ao caso anterior: o jar comum deve ser colocado no diretório "lib" da enterprise application e ser referenciado via arquivo MANIFEST.MF nas web applications e EJB´s que dele dependam.
- JAR é usado por mais de uma Enterprise Application
    - Nesse caso, a Arquitetura de Referência sugere que a biblioteca comum seja replicada em todos os EAR´s (diretório "lib") que dela façam uso. Deve-se evitar a colocação de jar nos diretórios de bibliotecas do servidor de aplicação ou no CLASSPATH deste.
    - Os servidores de aplicação, tal como JBoss e WildFly, permitem que bibliotecas comuns a mais de uma aplicação sejam compartilhadas entre elas sem necessidade de replicação de arquivos. Estes mecanismos podem ser usados quando permitirem determinar a localização dos JARs compartilhados e mapear quais aplicações dependem deles. O uso de módulos no JBoss AS e WildFly é um exemplo de alternativa.

Enterprise Application Archive

A seguinte estrutura é sugerida para o EAR da aplicação:

```
<nome-aplicacao>.ear/
    META-INF/
        application.xml
        jboss-deployment-structure.xml
    lib/
        <componente-externo_A>.jar
        <componente-negocio_A>.jar
        <componente-negocio_B>.jar
    resources.jar
    <modulo-web.war>
```

- O diretório "<nome-da-aplicação>.ear" é a raiz da enterprise application e deve ter um nome representativo à aplicação.
- O arquivo "application.xml" é o "deployment descriptor" da aplicação. Ele define todos os módulos JEE da aplicação, inclusive a web application.
- No diretório “lib” ficam:
    - os JARs das bibliotecas ou componentes de terceiros dos quais a aplicação depende, a não ser que as regras de dependência, na seção abaixo, indiquem outra localização;
    - os JARs dos componentes de negócio da aplicação. No caso de serem muitos componentes de negócio e houver independência entre eles recomenda-se separar em diferentes arquivos JAR.
- O arquivo “resources.jar” contém os arquivos de propriedades da aplicação e dos componentes de negócio que sejam comuns a toda aplicação.

4.2.2.1.1. Divisão em mais de uma Enterprise Application

Apesar de não recomendado, em alguns casos, a divisão da aplicação em mais de um arquivo EAR se justifica. Alguns critérios que podem levar a essa divisão:

- A aplicação possui um elevado número de componentes de negócio, os quais podem ser agrupados semanticamente em “super-partes”, com baixo ou inexistente nível de acoplamento entre essas.
- Existe um conjunto de componentes de terceiros que oneram consideravelmente as tarefas de implantação e manutenção (aumento do tempo necessário para subir a aplicação, maior complexidade no empacotamento dos diversos módulos, etc). Nesse caso, esses componentes de terceiros poderiam ser desacoplados do restante da aplicação, implantados em um EAR separado.

Alguns pontos de atenção que devem ser considerados:

- A divisão deve ser feita agrupando os componentes de forma semântica, eliminando ou minimizando ao máximo as dependências entre os EAR. Deve-se evitar a criação de

dependências cíclicas. Uma sugestão para evitar esse problema é fatorar os componentes comuns, usados pelos outros módulos, para um único EAR do qual os demais dependam.

- A divisão deve ser orientada a aumentar a produtividade no trabalho de implementação, no sentido de que o desenvolvedor não precise ter vários EARs instalados no seu ambiente para poder trabalhar e que não haja sobrecargas relacionados à tecnologia necessária para prover a comunicação entre os EARs.
- O risco de degradação de desempenho na aplicação deve ser levado em consideração, visto que a divisão pode implicar no uso de alguma tecnologia de distribuição (EJB e RMI) para prover a comunicação entre os módulos, que podem estar separados fisicamente.

Web Application Archive

A seguinte estrutura é sugerida para o WAR da aplicação:

```
<nome-modulo_web>.war/
    META-INF/Manifest.mf
    WEB-INF/
        classes/
        web.xml
        faces-config.xml
        beans.xml
        jboss-web.xml
        picketlink.xml
        lib/
            <componente-externo-A>.jar
            <componente-externo-B>.jar
            <componente-local-A>.jar
    protected/
    resources/
        css/
        images/
        js/
    index.xhtml
```

- O diretório "<nome-modulo-web>.war" é a raiz da web application e deve ter um nome representativo para a aplicação.
- A pasta resource contém estilos (CSS), imagens e bibliotecas javascript (js), fazendo uso do Resource Handler do JSF 2.2. Recomenda-se a criação de subpastas para agrupar os recursos estáticos de acordo com algum critério. Cabe ao arquiteto do projeto evoluir essas definição, conforme haja necessidade de novos tipos de recursos estáticos.
- Na pasta lib ficam:

    - Os JARs dos componentes externos dos quais as classes e demais recursos Web dependem, desde que as características de dependências não sugiram a colocação desses JARs em outra localização;
    - Os JARs com as classes da camada de apresentação dos componentes de negócio;
    - Os arquivos de propriedades dos componentes que estejam implantados no escopo da Web Application. Recomenda-se que esses arquivos sejam empacotados em um único JAR, por exemplo “resources.jar”.
- O arquivo "Manifest.mf", contém algumas informações de empacotamento da Web Application, entre elas a referência às bibliotecas utilizadas por aquela.
- A pasta protected contém arquivos protegidos pelo framework Pickelink e não podem ser acessados até que o usuários esteja devidamente logado na aplicação.
    - Essa pasta deve conter subpastas destinadas a separar os componentes de negócio, com nomes significativos para cada componente.
- Para o empacotamento utilizado no ambiente de produção, não deve haver classes compiladas na Web Application que não estejam empacotadas dentro de um arquivo JAR. A pasta "WEB-INF/classes", presente na especificação, só deve ser utilizado no ambiente de desenvolvimento, visando aproveitar os recursos de “*hot deployment*” dos servidores de aplicação.
- O arquivo "web.xml" é o "deployment descriptor" da web application. Ele contém informações sobre a implantação desta web application, como a definição dos servlets, o controller do JSF, listeners de segurança do picketlink.
- O arquivo “faces-config.xml” contém as configurações do framework JSF.
- O arquivo “beans.xml” contém as configurações do framework CDI. Ele é opcional.
- O arquivo jboss-web.xml contém configurações específicas para o WildFly, como configurações de Valve para o Picketlink e domínio de segurança.
- O arquivo picketlink.xml contém configurações de inicialização do framework de segurança Picketlink para configurar a aplicação como um Service Provider e efetuar a autenticação no Identity Provider.

4.2.2.1.2. Divisão em mais de uma Web Application

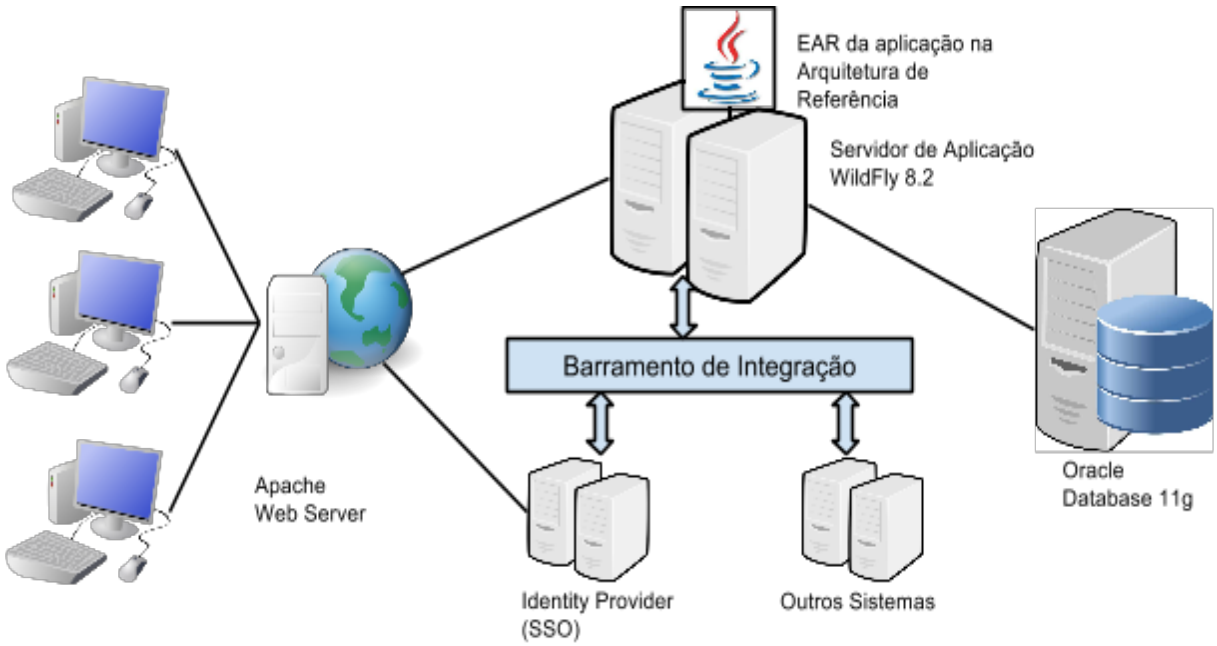
De forma geral, sugere-se que haja apenas uma Web Application por EAR, independente dos componentes de negócio que façam parte dessa última. A divisão das páginas entre os diferentes componentes de negócio deve ser feita através da criação de pastas específicas para cada componente.

Em algumas situações especiais, pode ser interessante a divisão dos diversos recursos da camada de apresentação em mais de uma Web Application. Exemplo seriam sub-sistemas, que podem até ter uma tema e interface diferentes.

Quando se se optar pela separação por mais de uma Web Application, deve-se ter o cuidado de definir claramente se haverá (e de que forma) o compartilhamento de informações entre esses módulos, visto que a especificação JEE define que cada Web Application tenha a sua própria sessão HTTP, o que impede o uso desse repositório como meio de compartilhar informações do usuário. Além disso, o classloader de cada uma das Web Applications, de acordo com a especificação, é independente.

4.3. Visão de Implantação

Esta seção tem como objetivo, descrever a estrutura de implantação padrão utilizada pelas aplicações desenvolvidas na arquitetura aqui definida.



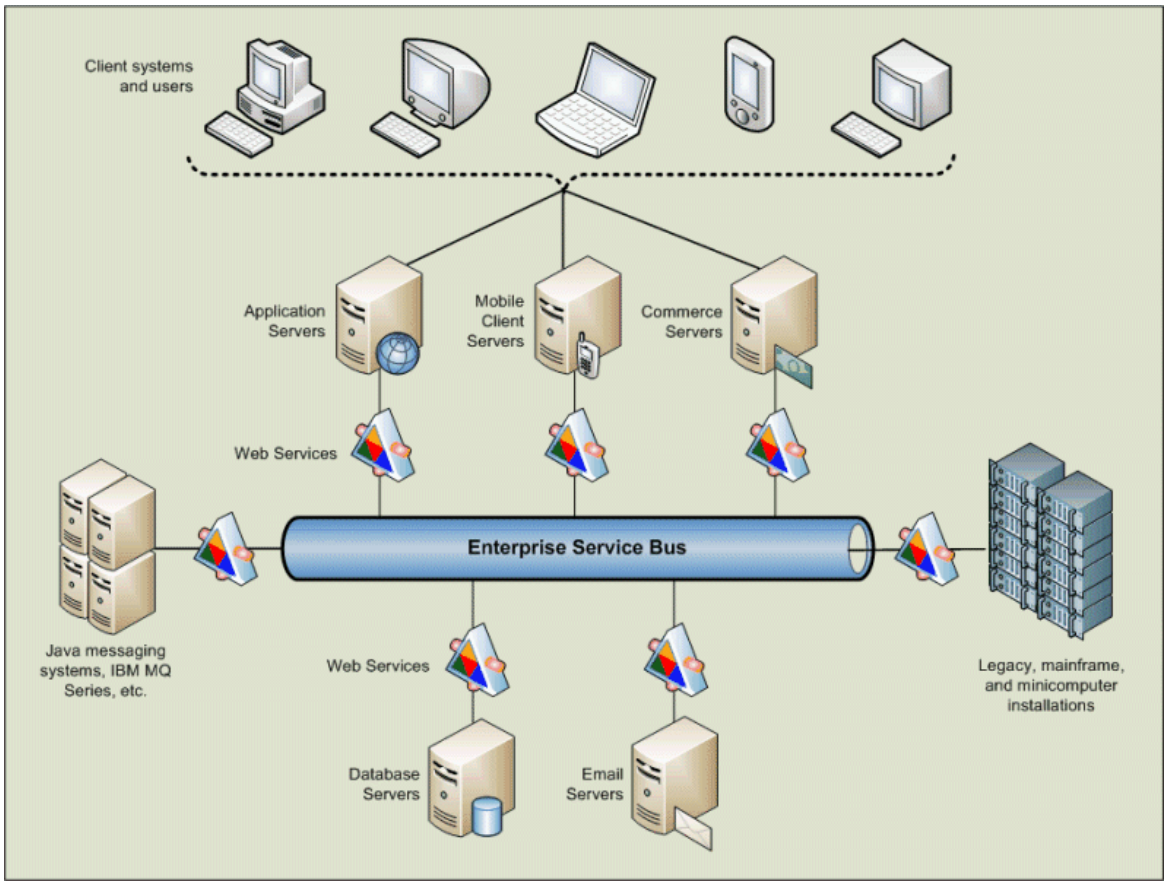
Vale ressaltar que a estrutura de implantação é dependente de cada projeto específico a fim de atender requisitos de disponibilidade, desempenho, escalabilidade. Por se tratar de uma arquitetura multicamadas (*n-layer* e *n-tier*) diferentes configurações de servidores podem ser utilizadas para atender os requisitos da aplicação e do ambiente de infra-estrutura.

4.4. Visão de Integração

Esta seção tem como objetivo apresentar como uma aplicação construída sob essa arquitetura de integração é capaz de se integrar a outros sistemas e módulos.

A integração numa arquitetura distribuída deve seguir os princípios de desacoplamento com uso de interfaces bem definidas e protocolos abertos. Algumas abordagens devem ser evitadas como transferência de arquivos e banco de dados compartilhado. Por outro lado outras abordagens são recomendadas como o uso de mensageria (JMS) e serviços REST (*Representational State Transfer*). A arquitetura de referência Java EE dispõe de todos os recursos necessários para que uma aplicação se integre com outras aplicações.

Um ESB (*Enterprise Service Bus*) permite integrar aplicações com baixo acoplamento e levando em conta uma miríade de tecnologias diversas em sistemas dispares. O ESB atua como um *middleware,* fazendo a intermediação de chamadas de serviços entre sistemas diversos.



Fonte: https://msdn.microsoft.com/en-us/library/ff648282.aspx

O ESB é uma ferramenta muito útil, que possibilita integração e desacoplamento entre sistemas, incluindo aplicações legadas. Mas há de se considerar que num ambiente corporativo, onde existem diversos sistemas e inúmeras integrações, a governança de serviços é mais importante do que o barramento. A adoção de um ESB é apenas um caminho para a adoção de uma Arquitetura Orientada a Serviços (SOA).

Para a adoção de SOA, é necessário:

- Uma ferramenta de ESB;
- Governança de serviços;
- Modelo canônico;
- Sistemas capazes de expor seus dados por meio de serviços;
    - Não é mandatório, visto ser possível integrar aplicações legadas, mas reduz consideravelmente o trabalho do ESB e as responsabilidades da integração.

Ambientes corporativos que adotam SOA e buscam maior eficiência nas suas integrações, contam normalmente com uma equipe de integração, que trabalham num modelo conhecido como ICC - *Integration Competency Center* (Centro de Competência em Integrações). É a equipe responsável pela definição e realização das integrações, manutenção e evolução do modelo canônico, governança dos serviços e implementação da integração na ferramenta ESB adotada. É também essa equipe que orienta as equipes de cada sistema que compõe uma integração, a fim de expor e consumir serviços no ESB.

Algumas referências rápidas sobre o assunto:

- http://www.amazon.com/Integration-Competency-Center-Implementation-Methodology/dp/0976916304
- http://www.inteligencija.com/download/Designing_an_ICC.pdf
- http://integrationwarstories.com/2013/11/27/establishing-integration-competency-center-part-1/
- http://www.aqueleblogdesoa.com.br/2013/01/mas-o-que-e-modelo-canonico-afinal/
- http://www.aqueleblogdesoa.com.br/2011/07/o-modelo-canonico-em-uma-abordagem-soa/

Integração é um assunto amplo e não é o objetivo esgotar o assunto. No que tange à capacidade das aplicações, que utilizam essa arquitetura, se integrarem a um ESB ou diretamente a outros sistemas veja as seções referentes à camada de integração.

5. Mecanismos Arquiteturais

5.1. Autenticação e Autorização

Segurança é um dos itens mais importantes de qualquer aplicação. Garantir a habilidade de uma aplicação proteger seus dados e informações de acesso não autorizado enquanto também garante o acesso às pessoas e sistemas autorizados é uma responsabilidade que deve ser endereçada pela arquitetura. A arquitetura de referência trata de todas as características de segurança (confidencialidade, integridade, disponibilidade, autenticação, não-repúdio e autorização), atendendo os em diferentes camadas da aplicação e da infra-estrutura de implantação.

A fim de garantir a segurança na aplicação os mecanismos de autenticação e autorização são definidos e descritos abaixo. Apesar de uma seção específica para explicar os mecanismos, eles são totalmente integrados às camadas da aplicação, garantindo que apenas usuários autenticados acessem aquelas transações para as quais tenham sido previamente autorizados. Uma tendência nas aplicações empresariais e no ambiente corporativo é também endereçado por esses mecanismos: o Single Sign-On.

5.1.1. Requisitos de Segurança

Antes de detalhar a solução de segurança proposta para a arquitetura vamos listar os requisitos de segurança que nortearam as decisões descritas abaixo. Os requisitos de segurança identificados e endereçados pela arquitetura são:

- Autenticação
    - baseada em identificador de usuário e senha
    - baseada em certificado digital
- Autorização
    - RBAC - *Role Based Control Access*
    - ACL - *Access Control List*
- Single Sign-On
- Integração com LDAP
- *Multi-Tenancy*[1]
- Base única de identidades, integrada às aplicações.

5.1.1.1. Visão Geral da Solução

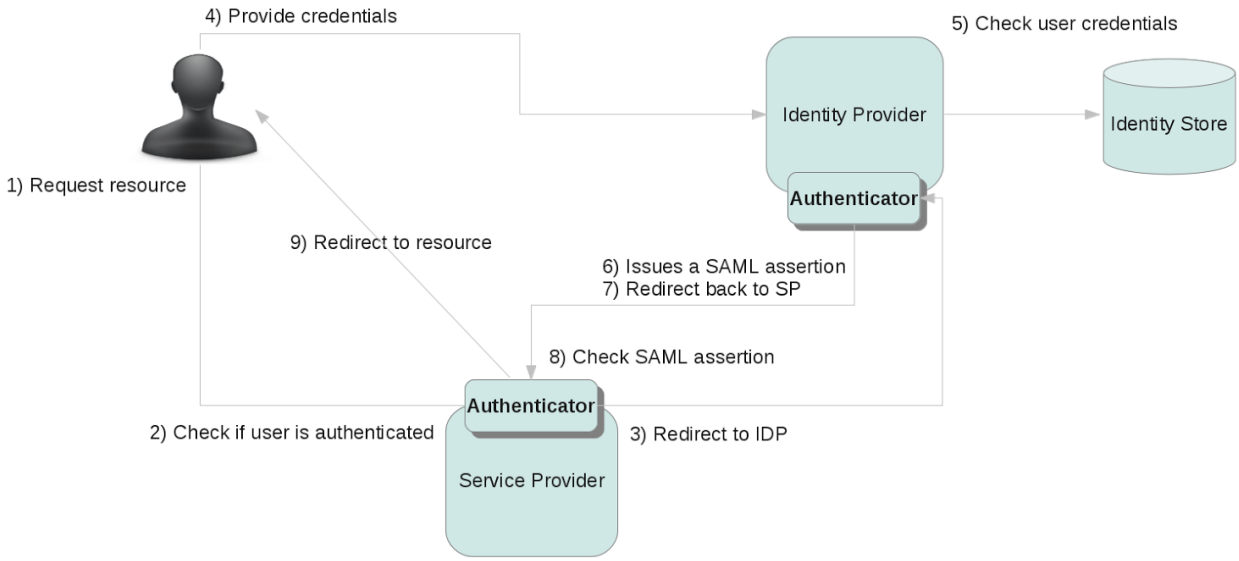
Para suportar os requisitos das aplicações da Justiça do Trabalho, descritos acima a solução de segurança se baseia nos conceitos de Federação. Dessa forma os componentes de segurança são divididos em:

- **Identity Manager (Identity Store)** - É o componente que mantém todos os dados de usuários e suas credenciais, seus perfis e relacionamentos. Além de armazenar os dados necesários para autenticação e autorização é o componente que permite a manipulação desses dados, por usuários autorizados.

[1] *Multi-Tenancy* é um modelo de desenho de arquitetura que visa compartilhar recursos de TI com custo eficiente e com segurança. Exatamente como um condomínio de apartamentos em que muitos inquilinos (*tenants*) compartilham uma infra-estrutura comum do prédio, mas tendo janelas e portas que garantem sua privacidade de outros condominos, uma arquitetura *multi-tenancy* compartilha recursos de TI de forma segura entre múltiplas aplicações. Esse conceito é bastante difundido em cloud e SaaS (Software as a Service). Tendo como base as implicações dessa características sobre a arquitetura e os dados de cada aplicação este conceito precisa ser tratado de forma que seja transparente para a aplicação.

- **Identity Provider** - Também conhecido como Identity Assertion Provider. É o componente que permite o login único (Single Sign-On) entre as aplicações. Utilizando recursos de SAML (Security Assertion Markup Language) permite realizar asserções sobre a identidade de um usuário e verificar se mesmo já está logado na federação. O Identity Provider consulta o Identity Store para verificar a identidade.
- **Service Provider** - É o componente do lado das aplicações que garante a comunicação com o Identity Provider, permitindo a troca de mensagens SAML.

A figura abaixo ilustra o processo de autenticação, mostrando o funcionamento desses componentes:



Quando um usuário deseja se autenticar numa aplicação (que se baseia nessa arquitetura) ele digita no navegador o endereço de tal aplicação. Um processo se inicia com os seguintes passos:

1. Navegador do cliente envia uma requisição para a aplicação, que pode ser entendida também como o Service Provider - SP;
2. O SP verifica se o usuário está autenticado, caso contrário delega ao Autenticador configurado, que autentique o usuário;
3. O Autenticador redireciona o navegador do usuário para o Identity Provider
    a. Neste momento a comunicação ocorre entre navegador do usuário e o Identity Provider
4. O usuário informa no navegador os dados necessários para a autenticação. Por exemplo identificador de usuário e senha;
5. O Identity Provider verifica as credenciais informadas pelo usuário, recorrendo ao Identity Store;

6. Neste passo o navegador e o Identity Provider fazem uma negociação utilizando SAML de forma que um token seja gerado e armazenado. Basicamente esse processo que garante o Single Sign-On;
7. O Identity Provider redireciona o navegador do usuário de volta para o Service Provider;
8. O Service Provider realiza asserções SAML, que basicamente significa verificar o token mantido pelo navegador que indica que o usuário foi autenticado com sucesso;
9. O Service Provider concede acesso ao recurso autorizado e responde para o navegador d usuário.

Para realizar esses componentes o principal framework adotado é o PicketLink. O PicketLink é um framework de Segurança para aplicações Java EE. Ele é originário do projeto JBoss Seam. Ele provê recursos para autenticar usuário, autorizar o acessos aos métodos de negócio da aplicação; gerenciar usuários, grupos, perfis, permissões e seus relacionamentos; Federação com SAML, OAuth, OpenID; Segurança de serviços REST e armazenamento de identidades em banco de dados e LDAP.

A arquitetura de segurança do Java EE, a especificação JAAS, também é utilizada, juntamente com CDI e o Módulo *Security* do framework Apache Deltaspike.

Framework	**Versão/ Release**	**Responsabilidade**
CDI	1.0 ou 1.1	Injeção de Dependências
Apache Deltaspike Security Module	1.1.1+	Módulo de Segurança do Apache Deltaspike para permitir utilizar anotações de autorização
Picketlink	2.6.0-Final ou 2.7.0-CR3	Autenticação, Autorização e Single Sign-On

Nas secões seguintes os detalhes de implementação de cada um dos componentes é detalhado. Foco é dado ao Service Provider, visto se tratar diretamente do mecanismo de autenticação e autorização da Arquitetura de Referência.

5.1.2. Identity Manager (IdM)

O Identity Manager é uma aplicação voltada às necessidades de gerenciamento de identifidades da Justiça do Trabalho. Ela está sendo desenvolvida pelas equipes de arquitetura do TRT3 e do TRT6 e tem o intuito de ser uma base de dados única de usuários e seus perfis em cada aplicação.

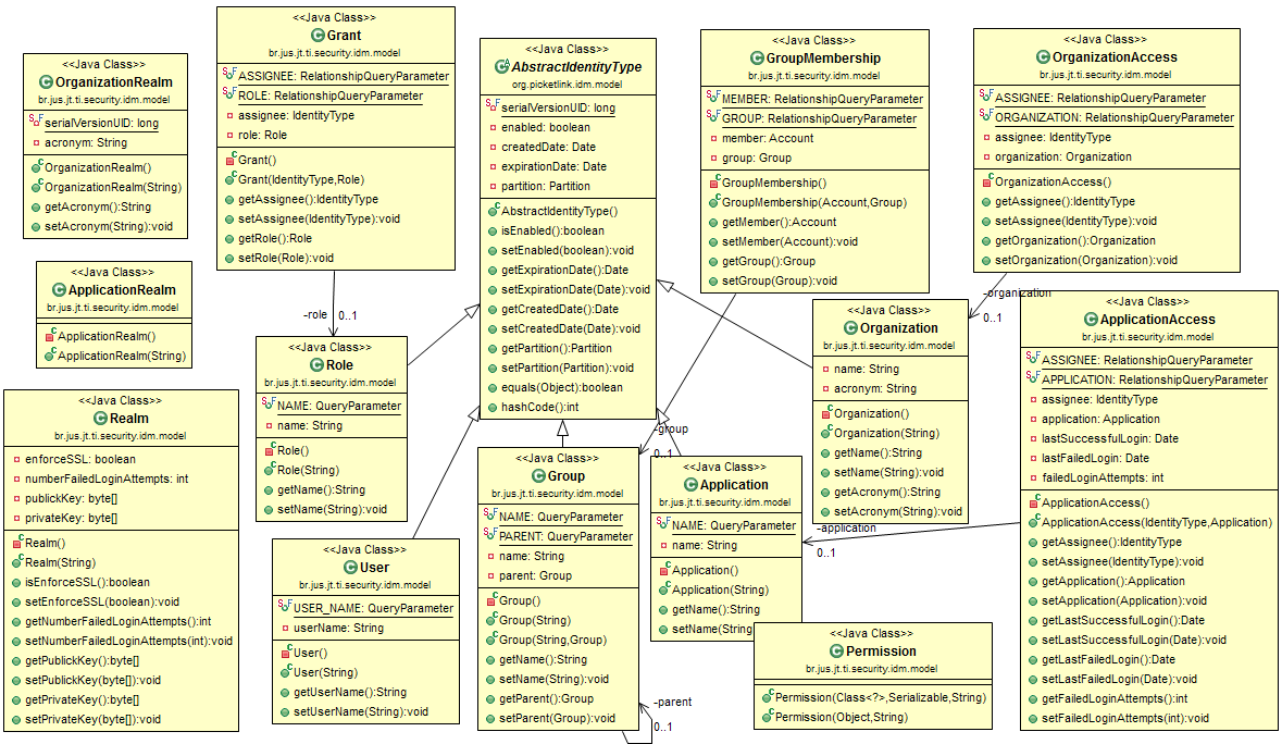
Essa aplicação será integrada aos demais módulos de gestão administrativa, permitindo responder a eventos funcionais que alterem as autorizações dos usuários. Por exemplo, um usuário X, que esteja substituindo por um tempo determinado o seu Diretor poderá receber neste período as mesmas autorizações concedidas ao Diretor da unidade.

O Identity Manager poderá ser operado por usuários especializados e permitirá também acesso de usuários de negócios que tenham poder de delegar autorizações. A interface será amigável, utilizando os mesmos recursos da camada de apresentação proposta nesta arquitetura de referência. Por exemplo o Diretor de Recursos Humanos poderá acessar diretamente a aplicação Identity Manager para conceder permissões aos seus chefes de suas secretarias, sem necessidade de intervenção da TI.

Grande parte das capacidades desse componente se deve ao modelo de Identidades adotado, descrito abaixo.

5.1.2.1. Modelo de Identidades (Identity Model)

O modelo de identidades da figura abaixo é uma extensão do modelo básico do PicketLink.



5.1.3. Identity Provider (IdP)

A exemplo do Identity Manager, a aplicação Identity Provider também está sendo desenvolvida pelas equipes de arquitetura do TRT3 e do TRT6. Essa aplicação será única, implantada em cada órgão da Justiça do Trabalho, e será capaz de autenticar os diferentes módulos administrativos que utilizem a arquitetura de referência.

Embora não seja o foco da arquitetura de referência, visto que as aplicações não precisam implementar um novo Identity Provider, as aplicações precisarão se comunicar para realizar autenticação e garantir o login único (SSO). Dessa forma, alguns detalhes de implementação interna são descritos abaixo.

O Identity Provider utiliza SAML v2. Ele usa a estrutura provida pelo Picketlink, sendo necessário apenas configurar alguns recursos (detalhado abaixo). A autenticação do usuário utiliza os domínios de segurança (security-domain) do servidor de aplicação. A especificação JAAS é utilizada e o **IdP** recebe o Principal do **IdM**.

A primeira configuração necessária para a aplicação IdP é o autenticador SAML que torna a aplicação um Identity Provider. O `IDPWebBrowserSSOValve` é um componente especial provido pelo PicketLink que sabe processar mensagens SAML.

No JBoss EAP 6 ele é configurado como um *valve Tomcat* no arquivo jboss-web.xml. A outra configuração é o security domain.

WEB-INF/jboss-web.xml

```
<jboss-web>
    <security-domain>idp-sigep</security-domain>
     <valve>

<class-name>org.picketlink.identity.federation.bindings.tomcat.idp.IDPWebBrowserSSOValve</clas
s-name>
    </valve>
</jboss-web>
```

No caso do WildFly, que usa Undertow no lugar do Tomcat é necessário configurar um Servlet Filter no arquivos web.xml.

WEB-INF/jboss-web.xml

```
<filter>
  <filter-name>IDPFilter</filter-name>
  <filter-class>org.picketlink.identity.federation.web.filters.IDPFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>IDPFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

No arquivo web.xml é também necessário restringir o acesso a todos os recursos da aplicação utilizando security constraints e configurar o formulário de login.

```
<security-constraint>
	<web-resource-collection>
		<web-resource-name>IdP Application</web-resource-name>
		<url-pattern>/*</url-pattern>
	</web-resource-collection>
	<auth-constraint>
		<role-name>RoleIdP</role-name>
	</auth-constraint>
</security-constraint>
<login-config>
	<auth-method>FORM</auth-method>
	<realm-name>SIGEP IDP App</realm-name>
	<form-login-config>
		<form-login-page>/login/login.xhtml</form-login-page>
		<form-error-page>/login/error.xhtml</form-error-page>
	</form-login-config>
</login-config>
<security-role>
	<role-name>RoleIdP</role-name>
</security-role>
```

O PicketLink permite uma série de configurações como suporte a assinatura, criptografia, tempo de duração de token. Essas configurações ficam no arquivo picketlink.xml. Abaixo há um exemplo com as configurações atuais da aplicação utilizadas na prova de conceito.

WEB-INF/picketlink.xml

```
<PicketLink xmlns="urn:picketlink:identity-federation:config:2.1">
   <PicketLinkIDP xmlns="urn:picketlink:identity-federation:config:2.1">
      <IdentityURL>${idp.url::http://localhost:8080/idp/}</IdentityURL>
      <Trust>
         <Domains>localhost,trt3.jus.br,trt6.jus.br</Domains>
      </Trust>
   </PicketLinkIDP>
   <Handlers xmlns="urn:picketlink:identity-federation:handler:config:2.1">
      <Handler
class="org.picketlink.identity.federation.web.handlers.saml2.SAML2IssuerTrustHandler" />
      <Handler
class="org.picketlink.identity.federation.web.handlers.saml2.SAML2LogOutHandler" />
```

```
      <Handler
class="org.picketlink.identity.federation.web.handlers.saml2.SAML2AuthenticationHandler" />
      <Handler
class="org.picketlink.identity.federation.web.handlers.saml2.RolesGenerationHandler" />
   </Handlers>
   <PicketLinkSTS xmlns="urn:picketlink:identity-federation:config:2.1" TokenTimeout="5000"
ClockSkew="0">
      <TokenProviders>
         <TokenProvider
ProviderClass="org.picketlink.identity.federation.core.saml.v1.providers.SAML11AssertionTokenP
rovider" TokenType="urn:oasis:names:tc:SAML:1.0:assertion" TokenElement="Assertion"
TokenElementNS="urn:oasis:names:tc:SAML:1.0:assertion" />
      <TokenProvider
ProviderClass="org.picketlink.identity.federation.core.saml.v2.providers.SAML20AssertionTokenP
rovider" TokenType="urn:oasis:names:tc:SAML:2.0:assertion" TokenElement="Assertion"
TokenElementNS="urn:oasis:names:tc:SAML:2.0:assertion" />
      </TokenProviders>
   </PicketLinkSTS>
</PicketLink>
```

O Identity Provider precisa de um Identity Store para recuperar as informações de usuário. Essas informações são usadas durante o processo de autenticação e autorização. No nosso caso o Identity Store é a aplicação Identity Manager, descrita acima, que tem como base o uso de banco de dados e LDAP.

O Identity Provider usa JAAS para conectar ao Identity Store, por meio de uma implementação customizada de LoginModule. Basta configurar o IdMLoginModule no security-domain da aplicação por meio do servidor de aplicação. Outra opção é configurar o arquivos login-config.xml.

```
public class IdMLoginModule extends org.jboss.security.auth.spi.UsersRolesLoginModule {
   //Implementação
}
```

login-config.xml

```
<policy>
   <application-policy name="idp-sigep">
       <authentication>
           <login-module code="br.jus.jt.ti.security.idp.IdMLoginModule" flag="required">
                   <module-option name="lookup">
```

```
                    java:comp/env/ejb/IdMAuthenticatorBean
                </module-option>
            <module-option name="ejbUser">admin</module-option>
            <module-option name="ejbPasswd">12345</module-option>
        </login-module>
    </authentication>
  </application-policy>
</policy>
```

5.1.4. Service Provider (SP)

No conceito de Federação e de Single Sign-On as aplicações que se autenticam no Identity Provider são conhecidas como Service Providers - SP. O que diferencia uma aplicação comum de um SP é um conjunto de configurações necessárias para garantir que a aplicação se autentique, utilizando SAML, passando pelo IdP. Novamente a estrutura provida pelo Picketlink é usada, sendo necessário realizar as configurações abaixo.

Na nossa implementação utilizamos redirecionamento para o IdP. Desse modo, tudo o que o SP faz é verificar que o usuário não está autenticado e repassar para o formulário de login do IdP.

Para que isso aconteça novamente a primeira configuração necessária é o autenticador. No JBoss EAP 6 ele é configurado como um *valve Tomcat* no arquivo jboss-web.xml. A outra configuração é o security domain que está sendo utilizado. Vamos falar daqui a pouco.

WEB-INF/jboss-web.xml

```
<jboss-web>
    <security-domain>sp-sigep</security-domain>
     <valve>
      <class-name>
        org.picketlink.identity.federation.bindings.tomcat.sp.ServiceProviderAuthenticator
      </class-name>
    </valve>
</jboss-web>
```

No caso do WildFly, que usa Undertow no lugar do Tomcat, é necessário configurar um *Servlet Extension* adicionando o arquivo **/WEB-INF/classes/META-INF/services/io.undertow.servlet.ServletExtension** ao WAR com o seguinte conteúdo.

WEB-INF/classes/META-INF/services/io.undertow.servlet.ServletExtension

```
org.picketlink.identity.federation.bindings.wildfly.sp.SPServletExtension
```

No arquivo web.xml é também necessário restringir o acesso aos recursos da aplicação utilizando security constraints e configurar o formulário de login.

```
<security-constraint>
	<web-resource-collection>
		<web-resource-name>SP Application</web-resource-name>
		<url-pattern>/*</url-pattern>
	</web-resource-collection>
	<auth-constraint>
		<role-name>RoleIdP</role-name>
	</auth-constraint>
</security-constraint>
<security-role>
	<role-name>RoleIdP</role-name>
</security-role>
```

As configurações acima ativam o PicketLink, mas é necessário configurá-lo como um Service Provider por meio do arquivo picketlink.xml, assim como no IdP. Abaixo há um exemplo com as configurações atuais da aplicação utilizadas na prova de conceito.

WEB-INF/picketlink.xml

```
<PicketLink xmlns="urn:picketlink:identity-federation:config:2.1">
   <PicketLinkSP xmlns="urn:picketlink:identity-federation:config:2.1" BindingType="REDIRECT">
      <IdentityURL>${idp.url::http://localhost:8080/idp/}</IdentityURL>
      <ServiceURL>${sp.url::http://localhost:8080/apptest/}</ServiceURL>
   </PicketLinkSP>
   <Handlers xmlns="urn:picketlink:identity-federation:handler:config:2.1">
                                                                        <Handler
class="org.picketlink.identity.federation.web.handlers.saml2.SAML2LogOutHandler" />
                                                                        <Handler
class="org.picketlink.identity.federation.web.handlers.saml2.SAML2AuthenticationHandler" />
                                                                        <Handler
class="org.picketlink.identity.federation.web.handlers.saml2.RolesGenerationHandler" />
   </Handlers>
</PicketLink>
```

Assim como Identity Provider, o Service Provider utiliza JAAS para se autenticar. Dessa forma precisamos configurar um LoginModule. O PicketLink já provê tal componente que cuida de toda a comunicação SAML v2 com o IdP.

standalone.xml ou domain.xml

```
<security-domain name="sp" cache-type="default">
  <authentication>
    <login-module
code="org.picketlink.identity.federation.bindings.jboss.auth.SAML2LoginModule"
flag="required"/>
  </authentication>
</security-domain>
```

login-config.xml

```
<policy>
   <application-policy name="idp-sigep">
       <authentication>
                                                                    <login-module
code="org.picketlink.identity.federation.bindings.jboss.auth.SAML2LoginModule"
flag="required">
           </login-module>
       </authentication>
   </application-policy>
</policy>
```

5.1.4.1. Autenticação e Single Sign-On

No Service Provider não é necessário realizar mais nenhuma configuração para realizar autenticação. As configurações acima já são suficiente para redirecionar para o IdP, que realizará a autenticação (seja por senha, certificado) e retornará. Toda a comunicação SAML para realizar o Single Sign-On também fica transparente para a aplicação.

PicketLink possui um componente central chamado *Identity* bean. Ele fica no pacote org.picketlink e representa o usuário autenticado na sessão atual. Ele provê diversos métodos úteis para controlar o processo de autenticação e verificar os privilégios do usuário autenticado.

Este bean pode ser injetado na aplicação utilizando CDI e poder ser utilizado diretamente em páginas JSF utilizando a Expression Language.

```
@Inject Identity identity;
```

```
@Named(“identity”)
```

5.1.4.2. Single Sign-Out

Um efeito colateral do Single Sign-On é a necessidade de um Logout Global e não mais apenas ou logout da aplicação. Quando o usuário efetua logout apenas na aplicação local ele apenas desconecta do Service Provider. A sessões no Identity Provider e em outros Service Provider continuam ativas, ou seja, é possível efetuar login novamente utilizando o SSO. Dessa forma é extremamente recomendado sempre realizar um Logout Global.

Uma vez que a especificacão SAML v2 possui suporte a Logout Global é bem simples utilizar este recurso nas aplicações que utilizam esta arquitetura.

Para efetuar o Global Logout basta adicionar à URL da aplicação Service Provider o seguinte: “?GLO=true”. Por exemplo:

http://localhost:8080/apptest?GLO=true

Para realizar o logout apenas local o sufixo é “?LLO=true”.

5.1.4.3. Autorização

Existem diversas formas de restringir o acesso de usuários a determinados recursos. O Modelo de Identidades, visto acima, já diz algumas das possibilidade com o PicketLink: usuário autenticado, por grupo, por perfil, por recurso. O PicketLink provê diversos mecanismos para declarar quem tem acesso a que no nosso código. Ele também utilizar o módulo Security do Apache DeltaSpike para autorização.

Para realizar a checagem de autorização em classes e métodos que desejamos restringir acesso com anotações de segurança é preciso primeiro habilitar o Security Interceptor no arquivos beans.xml. Essa configuração só é necessária para CDI 1.0. Já utilizamos a versão 1.1.1 do Apache DeltaSpike Security Module.

```
<beans>
    <!-- Not needed with CDI 1.1+ and DeltaSpike v1.1.1+ -->
    <interceptors>
        <class>org.apache.deltaspike.security.impl.extension.SecurityInterceptor</class>
    </interceptors>
</beans>
```

Autorizando usuários logados

É muito comun restringir o acesso a algum recurso baseado apenas no fato do usuário estar autenticado. Isso pode ser feito com a anotação @LoggedIn.

No exemplo abaixo somente usuários autenticados podem acessar o método.

```
@LoggedIn
public void logout() {
        // only authenticated users can logout
}
```

A anotação também pode ser aplicada a tipos. No exemplo abaixo todos os métodos são restritos ao usuário logado.

```
@LoggedIn
public class MyRESTAPI {

}
```

No modelo de Identidade utlizado atualmente existe apenas um tipo de Usuário. Mas o PicketLink permite especialização. Se houvesse especialização, seria possível restringir o acesso apenas a servidores ou magistrados. No exemplo abaixo o acesso é restrito a Servidores.

```
@LoggedIn(requiresAccount = {Servidores.class})
```

Restringindo acesso à Organização e Aplicação

No PicketLink partições são separações lógicas para usuários, seus perfis e seus grupos. Este é um recurso importante para Multi-Tenancy e também para um serviço transversal de segurança capaz de autorizar um usuário em diferentes aplicações. No Modelo de Identidades há partições do tipo **Realm**, que reúnem configurações de segurança e partições por **Organização** e **Aplicação**.

A anotação @PartitionsAllowed permtie restringir o acesso apenas a partições que o usuário pertença. No exemplo abaixo apenas o usuário da partição TRT3 poderá acessar.

```
@PartitionsAllowed("TRT3")
public void someMethod() {
        // somente usuários da partição TRT3 poderão acessar o método.
```

```
}
```

A anotação @PartitionsAllowed também se aplica a classes e pode ser utilizado para configurar múltiplas partições. É ainda possível especificar o tipo da partição.

```
@PartitionsAllowed({"TRT3", "TRT6"})
public class MinhaClasse {

}

@PartitionsAllowed(name="SIGEP", type=ApplicationRealm.class})
public class MinhaClasse {
```

5.1.4.3.1. RBAC - Role Based Access Control

Um dos modelos mais comuns de autorização é baseado nos perfils (Roles) atribuídos para um usuário. Dessa forma, PicketLink provê uma anotação específica, com a qual basta especificar o nome do perfil que deverá ter acesso. No exemplo abaixo apenas os usuários com o perfil Administrator tem acesso ao método shutdown.

```
@RolesAllowed("Administrator")
public void shutdown() {
        // only users granted with this role can access this method
}
```

A anotação também se aplica a classes e neste caso restringe todos o métodos da mesma. Também é permitido especificar mais de um nome de perfil para a anotação, como no exemplo abaixo:

```
@RolesAllowed({"Servidor", "Magistrado"})
```

5.1.4.3.2. ACL - Access Control List

Outra forma de restringir acesso e também disponível na arquitetura, por meio do PicketLink, é o ACL. Ela está disponível por meio da anotação @RequiresPermission.
No Modelo de Identidades existe a classe Permission. Um objeto dessa classe permite associar um recurso da aplicação e uma operação possível nesse recurso. O objeto da classe Permission pode ser atribuído a um usuário, indicando que ele tem permissão de realizar determinada operação em um recurso.

Utilizando a anotação **@RequiresPermission** dizemos que para acessar um determinado método o usuário precisa ter acesso concedido ao recurso para realizar uma determinada operação. O que PicketLink faz, portanto, é checar o modelo de identidades para verificar se o usuário possui um objeto Permission que atenda a restrição.

```
@RequiresPermission(resource = "contracheque", operation = "read")
public javax.ws.rs.core.Response getContracheque() {

}
```

Similiarmente o objeto Identity possui o método hasPermission.

O PicketLink é normalmente implantado no JBoss como módulo. Nesse caso é necessário ativá-lo na implantação do Identity Manager, do Identity Provider ou do Service Provider, configurando a arquivo jboss-deployment-structure.xml, conforme abaixo.

```
<jboss-deployment-structure>
  <deployment>
    <dependencies>
      <module name="org.picketlink"/>
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

5.2. Logging

O log deve ser feito usando a biblioteca Apache Log4j 2. Alguns pontos positivos:

- Separação entre a API e a implementação. É possível utilizar qualquer uma das diversas API's existentes (Log4j, Commons Loggin, SLF4J, etc), assim como gerar log em qualquer uma dessas implementações, além da implementação nativa do Log4j 2
- A API é muito bem definida e permite a criação de mensagens de forma bem estruturada. Existem métodos específicos para mensagens de entrada ou saída de métodos, lançametno de exceções e mensagens gerais.

O logger será injetado nas classes através de CDI, com o uso da anotação @Inject. Por exemplo:

```
import org.apache.logging.log4j.Logger;
import javax.inject.Inject;
```

```
public class xyz {
   @Inject
   private Logger log
}
```

Daí em diante, é possível utilizar a propriedade log para emitir as mensagens nos momentos apropriados.

A API do Log4j 2 possui métodos convencionais de log, como warn, error e info. Além disto, ela possui métodos extras, que são bastante interessantes. Abaixo, seguem algumas sugestões de uso:

- entry: utilizado para marcar um ponto de entrada em um método. Pode ser usado para registrar os parâmetros passados para o método. Utiliza o nível de logging TRACE.
- exit: utilizado para marcar um ponto de saída do método. Pode receber o resultado do método como parâmetro de entrada, para que o valor seja registrado no log. Utiliza o nível TRACE.
- catching: utilizado para registrar a captura de uma exceção que **não será relançada**. Utiliza o nível ERROR.
- throwing: utilizado para registar o lançamento de uma nova exceção, cuja possibilidade de captura seja baixa. Principalmente, exceções de tempo de execução, como RuntimeException. Utiliza o nível ERROR.
- trace: usado para emitir mensagens que detalhem o fluxo de execução do programa, num nível mais detalhado do que DEBUG. Por exemplo, mensagens que mostrem conteúdos de variáveis ou valores que o usuário forneceu. Utiliza o nível TRACE.
- debug: usado para mostrar mensagens relativas à execução do programa, num nível mais alto do que TRACE. Não deve conter conteúdo de variáveis ou valores entrados pelo usuário. Utiliza o nível DEBUG.
- info: usado para mensagens úteis em tempo de execução e que não estejam envolvidas com o fluxo interno do sistema. Por exemplo, informar que uma operação determinada operação foi finalizada com sucesso. Utiliza o nível INFO.
- warn: emite mensagens de aviso ao usuário. Mensagens de aviso são aquelas que indicam que algo está diferente do esperado, mas ainda sim é possível continuar com a execução. Utiliza o nível WARN.
- error: emite mensagens de erro, que indicam ao usuário a ocorrência de um problema que interrompe a execução da tarefa atual. Todavia, o programa em si continua executando. Utiliza o nível ERROR
- fatal: indica ao administrador que o programa encontrou uma situação em que não é mais possível continuar a execução como um todo. Deve ser usado apenas numa situação totalmente irrecuperável e indicará ao administrador que o sistema foi interrompido. Utiliza o nível FATAL

A maior dúvida dos desenvolvedores costuma ser entre o uso dos níveis info, debug e trace. Uma diretriz importante é o nível de informação que está sendo passado na mensagem de log. Se a mensagem mostrar o conteúdo de variáveis ou do estado interno do sistema, é aconselhável que seja usado o trace. Se a mensagem mostrar o ponto em que o sistema está atualmente ou algum detalhe da execução, deve ser usado o debug. Se for uma mensagem que interessa mais ao usuário/administrador do sistema do que ao desenvolvedor, deve ser usado info. Os métodos entry e exit ajudam muito na clareza do uso de logging, pois deixa mais explícita a intenção do desenvolvedor. Também permite a documentação dos parâmetros e resultado dos métodos, sem que o desenvolvedor tenha de se preocupar com isto.

Além dos níveis de severidade, um logger pode fazer uso de LogMarkers para identificar o tipo de ação que está sendo logada, em geral, ligado à camada em que a ação ocorre e o tipo da ação. Abaixo um exemplo de LogMarker:

```
public abstract class LogMarker {
    public static final Marker REST = MarkerManager.getMarker("REST");
    public static final Marker REST_PESSOAS =
        MarkerManager.getMarker("PESSOAS").setParents(REST);
    public static final Marker REST_TELEFONES =
        MarkerManager.getMarker("TELEFONES").setParents(REST);
}
```

Uso do LogMarker:

```
   logger.error(LogMarker.REST_TELEFONES, "tipoTelefone {} invalido", id);
```

5.2.1. Integração do Log4J 2 com o Wildfly

O Log4J 2 pode conflitar com classes já existentes no servidor de aplicação. Por este motivo, é importante configurar o deploy da aplicação para que sejam excluídos alguns módulos. Para registro, eles são:

- org.apache.log4j
- org.apache.commons.logging
- org.jboss.as.logging
- org.jboss.log4j
- org.jboss.logging
- org.jboss.log4j.logmanager
- org.jboss.logmanager

O mecanismo de exclusão do Wildfly é o arquivo jboss-deployment-structure.xml, que deve ser colocado no diretório /META-INF do arquivo EAR.

5.3. Auditoria

Os requisitos de auditoria são comuns em grandes sistemas corporativos. Através deles é possível ter rastreabilidade de ações no sistema e de alterações de dados feitas pelo usuário. Em geral a auditoria deve conter algumas informações básicas como ação executada, timestamp, nome do usuário, endereço IP do usuário.

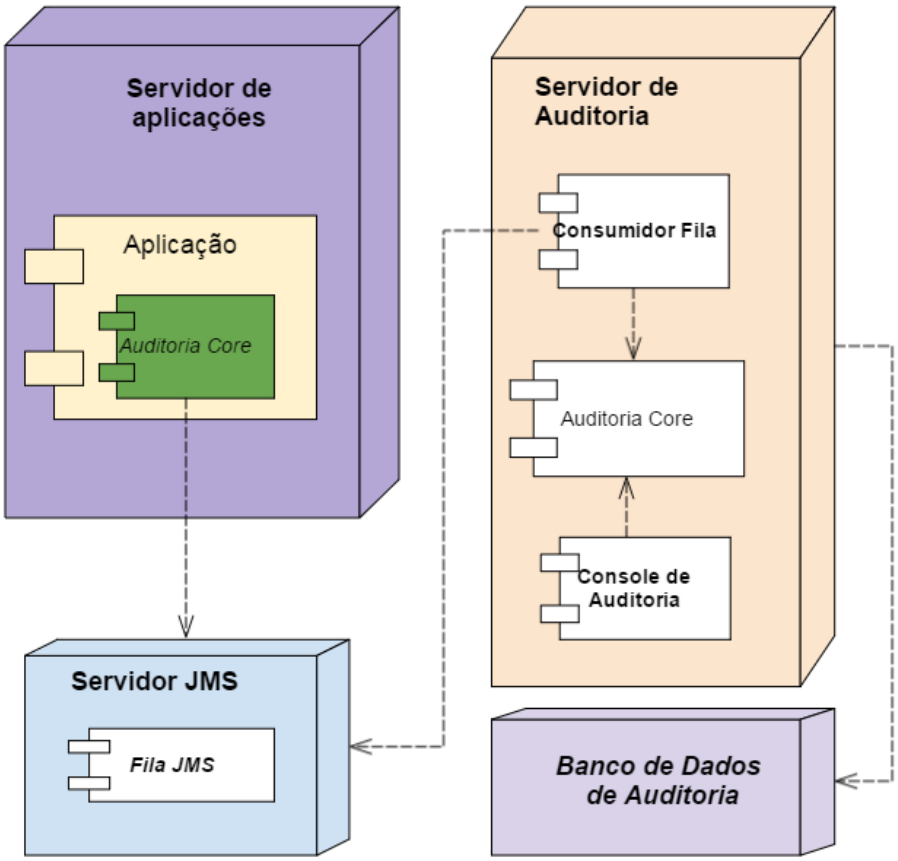
É importante diferenciar auditoria de logging. Enquanto o logging foca a geração de logs para uso dos desenvolvedores e/ou administradores visando detecção de erros e visualização do funcionamento do sistema, a auditoria esta fortemente ligada a requisitos funcionais, inclusive devem ser definidas na fase de levantamento de requisitos.

Outro ponto é que a auditoria em geral não se preocupa com histórico de dados. Caso exista essa necessidade funcional o requisito pode ser melhor atendido por frameworks como o Jboss Envers.

Os dados de auditoria não tem natureza online pois são acessados a priori. Portanto sua natureza é de coleta online mas o armazenamento e processamento pode ser assíncrono. Deste modo o recurso de auditoria pode onerar ao mínimo as operações no sistemas auditado. Como regra geral as mensagens de auditoria são geradas com bem menos frequência do que as de logging.

Diante dessas características o uso de mensageria se torna bastante interessante na implementação de um componente ou modelo de auditoria.

Deverá existir uma camada do nosso framework dedicada à emissão de mensagens de auditoria, de forma que o desenvolvedor não utilize o Log4j 2 diretamente. Esta camada deverá encaminhar as mensagens de forma assíncrona para que outro componente deva armazenar estas mensagens de maneira segura e assim permitir sua análise posterior.



5.4. Transações

As transações deverão ser delimitadas na Camada de Negócios. A chamada a métodos de objetos desta camada deve envolver a abertura e fechamento automático de uma transação. Se já houver uma transação em andamento, esta deverá ser aproveitada. O uso de transações aninhadas deve ser estudado caso a caso.

A API de gerenciamento de transações deve ser a JTA, pois é a API definida pelo padrão JEE. O uso de transações distribuídas deve ser evitado e, se estritamente necessário, avaliado com cautela e, de preferência, deve ser configurado de forma não programática. O uso de delineamento transacional programático deve ser evitado ao máximo, para permitir o reaproveitamento do código, independente dos detalhes de conexão ao banco de dados.

As transações deverão ser delimitadas pelo container. Assim usaremos EJB Stateless, como no exemplo abaixo:

```
@Stateless
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class CidadeUfFacade implements Serializable, Facade {

	@TransactionAttribute(TransactionAttributeType.READ_ONLY)
	public List<Cidade> findAllCidades() {
		...
	}
}
```

Neste exemplo, a classe foi anotada com @Stateless, o que torna esse bean um EJB, ganhando, assim, um contexto transacional. A anotação @TransactionAttribute informa ao container que todos os métodos desta classe exigem a existência prévia de uma transação. A menos que algum método tenha uma anotação diferente, como é o caso de findAllCidades().

O uso de EJB do tipo @Stateful não é recomendado. Os EJBs estão numa camada que é independente do cliente da aplicação. Não é recomendado que seja armazenado estado da aplicação em objetos das camadas de negócio e de integração. Isto permite um escalonamento melhor e evita problemas de sincronização. O estado deve ficar sempre o mais perto possível do cliente. No caso de JSF, é recomendado o uso de componentes de controle, que podem ser configurados para ficarem em escopos apropriados para cada caso, de forma que eles mantenham estado o mínimo possível.

Os conceitos de delimitação transacional do EJB foram provados ao longo do tempo, sendo as últimas evoluções apenas a simplificação do código. Assim os seguintes atributos de transação definidos na especificação podem ser utilizados: *Required, RequiresNew, Mandatory, NotSupported, Supports, Never.*

5.4.1. Conexão a múltiplos bancos de dados

As situações em que a aplicação precisa buscar ou gravar em mais de um banco de dados demandam um pouco mais de configuração. Isto porque a framework só poderá fornecer a solução para o caso padrão, com uma conexão e um entity manager. Quando houver mais de um, a configuração ficará a cargo da própria aplicação.

Primeiro, é necessário criar as interfaces qualificadoras, uma para cada Entity Manager extra (além do default):

```
@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface ExtraEntityManager {}
```

A aplicação deverá fornecer um producer para os vários Entity Managers (apenas para os que não forem o Entity Manager padrão):

```
public class ExtraEntityManagerProducer {
   @Inject Logger log;

   @PersistenceUnit(unitName = "extra")
   private EntityManagerFactory emfExtra;

   @Produces
   @RequestScoped
   @ExtraEntityManager
   public EntityManager produceEntityManager() {
       log.entry();
       return log.exit(emfExtra.createEntityManager());
   }

   public void closeEntityManager(@Disposes EntityManager em) {
       if (em.isOpen()) {
           em.close();
       }
   }
}
```

Este producer obterá instâncias de EntityManager's da unidade de persistência injetada e injetara em objetos que tenham a anotação @ExtraEntityManager. Como estamos utilizando a implementação de Repositories do DeltaSpike, vamos precisar criar ainda um EntityManagerResolver, que depois será associado aos repositórios que precisarem acessar esse entity manager. Por exemplo:

```
public class ExtraEntityManagerResolver implements EntityManagerResolver {
   @Inject @ExtraEntityManager // Qualificador utilizado para diferenciar o producer
   private EntityManager em;

   @Override
   public EntityManager resolveEntityManager()
   {
       return em;
   }
}
```

Agora, associamos um repositório usando a anotação @EntityManagerConfig:

```
@Repository
@EntityManagerConfig(entityManagerResolver = ExtraEntityManagerResolver.class, flushMode =
FlushModeType.COMMIT)
public interface PersonRepository extends EntityRepository<Person, Long> {
   ...
}
```

5.5. Mecanismo de Tratamento de Exceções

O mecanismo de exceções do Java é bastante controverso e existem varias formas diferentes de utilizá-lo. Neste documento, vamos propor uma maneira de lidar com as exceções, de forma que o desenvolvedor tenha o mínimo de preocupação com a existência delas, mas também possa utilizar as exceções de formas mais avançadas, caso seja necessário.

As exceções na linguagem Java podem ser classificadas como Checadas (checked) ou de Runtime. As exceções Checadas são as que descendem diretamente de java.lang.Exception. Para estas exceções, o compilador garante que elas sejam tratadas ou declaradas como lançáveis pelo método que as origina.

As exceções do tipo Runtime herdam de java.lang.RuntimeException e o compilador não exige que elas sejam tratadas ou declaradas como lançáveis.

Idealmente, o uso de exceções checadas leva a um código mais bem documentado, visto que ele declara explicitamente as possibilidades de resultado da chamada: ou se retorna o que se pediu ou uma das exceções declaradas será lançada. Porém, com o aumento da complexidade dos sistemas, o que se percebe é que essas exceções são apenas declaradas nos métodos ou encapsuladas sem um tratamento apropriado. Além disso, a tendência é que cada tratamento com relançamento faça um log da exceção, duplicando o registro da exceção nos arquivos de log.

Por este motivo, a proposta atual favorece o uso de exceções de runtime. Isto leva a um código menos verboso e aglutina o tratamento das exceções em um local comum (um interceptador).

5.5.1. Exceções de negócio e de sistema

Uma outra forma de classificar exceções é em relação à origem: exceções de negócio e exceções de sistema.

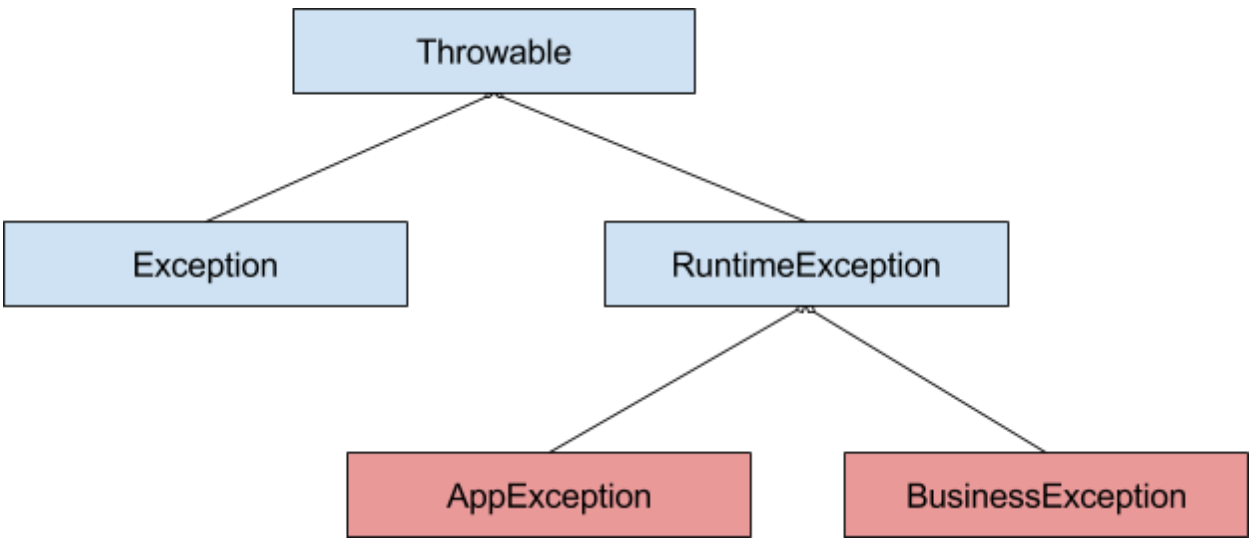
Exceções de negócio estão associadas a problemas de validação interna ou regras de negócio da aplicação. Elas são de interesse principal para o usuário. Em alguns casos, pode-se tratar

uma exceção de negócio sem que o usuário tenha que se preocupar. Em outros, deve-se avisar o usuário para que ele tome a decisão para corrigir o problema.

Exceções de sistema são originadas fora da aplicação, em alguma de suas dependências. Por exemplo, o banco de dados não está online ou estourou o espaço de armazenamento, ou a JVM ficou sem memória para alocar objetos. Todos estes problemas são de pouca utilidade para o usuário e são do interesse do administrador do sistema ou do desenvolvedor da aplicação.

A classe utilizada para lançar exceções de negócio será a br.jus.trt.lib.common_core.exception.BusinessException. Essa exceção é do tipo runtime e pode ser lançada sem que precise ser declarada na definição do método. Além disso, o contêiner EJB fará o rollback da transação quando receber uma exceção de runtime. A BusinessException possui ainda o conceito de código de erro, útil para a identificação da mensagem para efeitos de suporte ao usuário. Ela também pode armazenar uma lista de mensagens a serem mostradas ao usuário, o que permite que uma validação complexa retorne varias mensagens simultaneamente. Assim, o usuário não precisa reenviar o formulário várias vezes até que todos os erros estejam resolvidos.

A classe utilizada para exceções de sistema será a br.jus.trt.lib.common_core.exception.AppException. Ela também é uma exceção de runtime, mas não possui o mecanismo para código de erros ou múltiplas mensagens.

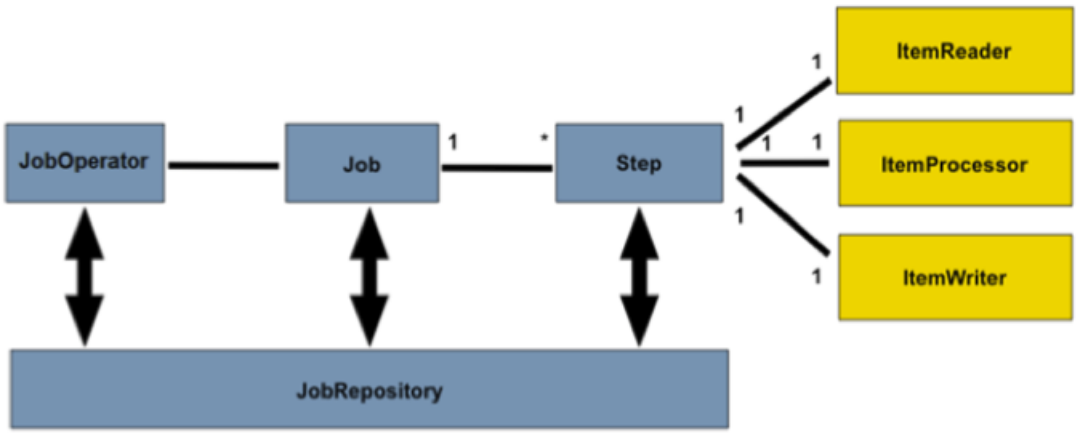


A camada de Service Facade é a responsável por delimitar as transações. Por isto, ela é o ponto em que as exceções devem ser interceptadas. Assim, se alguma exceção que não for BusinessException ou AppException, ela será envolvida em uma AppException. Para que isto ocorra, e necessário que os Service Facades sejam anotados com a @BusinessExceptionHandler. Por exemplo:

```
@Stateless
@LocalBean
@TransactionAttribute(TransactionAttributeType.REQUIRED)
@BusinessExceptionHandler
public class CidadeUfFacade implements Serializable, Facade {
	public void …..() {
		…..
	}
}
```

Na camada de apresentação também existe um conceito semelhante, porém, é aplicado a beans do JSF. Estes beans podem ser chamados de controllers ou actions. Eles devem ser anotados com @ActionExceptionHandler para que o interceptador correspondente possa capturar as exceções que ocorrerem e agir de acordo. O interceptador possui um tratamento especial para exceções do tipo BusinessException (exceções que representam erros da camada de negócio da aplicação). Para estas exceções, as mensagens contidas nelas são passadas para o contexto JSF e podem ser capturadas para serem mostradas pela camada de apresentação.

5.6. Agendamento de Tarefas/Processamento Batch



5.6.1. Introdução

Esta nova especificação disponibiliza um conjunto de recursos para criação de aplicações batch em Java com as seguintes características:

- criação de JobRepositories para cadastramento de trabalhos (*jobs*) a serem executados;

- disponibilização de filas de processamento batch controladas por serviços chamados JobOperators, que podem ser distribuídos;
- JobOperators recebem solicitação de execução de Job via linha de comando, diretamente das aplicações ou pela interface de gerenciamento disponibilizada no container;;
- os passos de um job (steps) podem ser agrupados em um workflow (flow e split) configurados via um schema xml chamado JSL (Job Specification Language);
- existe o recurso de herança de job, step, flow e split para reuso/especialização de tarefas, passos ou partes de um workflow;
- passos podem ser paralelizados (particionamento/concorrência na execução de steps);
- controle transacional incluído e customizável, que pode ser por registro, por grupo de registros (chunks de tamanho determinado), por step, ou por critério de negócio (regras de controle de transação criadas pela aplicação);
- recursos para recuperação automatizada de falhas na execução de um Job;

5.6.2. Agendamento de tarefas

A implementação da especificação JSR 352 no Wildfly provê ferramenta de solicitação de execução de tarefa por CLI (command line interface) ou pela interface de gerenciamento do container (JBeret Web ou a padrão). No entanto, pelas necessidades do projeto no tocante a agendamento de execução de Jobs, será adotado o agendamento em código, que dever ser feito de duas das 3 maneiras possíveis:

- ManagedScheduledExecutorService usando javax.enterprise.concurrent.Trigger como mostrado abaixo:

```
@Stateless
public class MeuEJBStateless {
	@Resource
	ManagedScheduledExecutorService executor;

	public void runJob() {
	executor.schedule(new MeuJob(), new Trigger() {
		public Date getNextRunTime(LastExecution ultimaExecucao, Date horario) {
		Calendar cal = Calendar.getInstance();
		cal.setTime(horario);
		cal.add(Calendar.DATE, 1);
		return cal.getTime();
		}

		public boolean skipRun(LastExecution ultimaExecucao, Date horario) {
		return null == ultimaExecucao;
		}
	});
	}

	public void cancelJob() {
	executor.shutdown();
	}
```

```
}
```

Chamando runJob para iniciar a execução e cancelJob para terminar a execução da tarefa. Nesse exemplo, uma nova execução é iniciada um dia depois da execução anterior. E nova execução não é iniciada até que a execução anterior termine. O exemplo acima é trivial e não possui todas as verificações necessárias para a execução adequada da tarefa.

Abaixo um exemplo trivial de Job:

```
public class MeuJob implements Runnable {
    public void run() {
        BatchRuntime.getJobOperator().start("meuJob", new Properties());
    }
}
```

- ou uma variação da forma acima, que permite que a execução de um job seja iniciado depois de um período de tempo:

```
public void runJob2() {
    executor.scheduleWithFixedDelay(new MeuJob(), 2, 3, TimeUnit.HOURS);
}
```

- **deverá ser evitada a forma abaixo**, uma vez que o agendamento fica fixo no código:

```
@Singleton
public class MeuEJB {
    @Schedule(hour = "23", minute = "59", second = "59")
    public void myJob() {
        BatchRuntime.getJobOperator().start("meuJob", new Properties());
    }
}
```

- Ainda não foi descartado o uso do Quartz Scheduler para agendamento de tarefas, que caso faça parte da arquitetura de referência, pode ser usado para acionar a execução de `BatchRuntime.getJobOperator().start("meuJob", new Properties());` de qualquer método escalonado/executado pelo Quartz.

Quanto ao uso de Batchlets, deve ser direcionado para tarefas de inicialização do Job, finalização (limpeza) do Job ou pequenas tarefas que alocam poucos recursos e por periodo pequeno de tempo.

Já para a maioria das tarefas batch, o processamento deverá ser feito em Chunks com Checkpoints bem definidos. É recomendado que o *checkpoint-policy* seja *item,* com um item-count de tamanho razoável, que não seja muito pequeno nem que seja grande demais que

demore vários minutos para ser processado, com um tamanho que seja relevante para o negócio, mas não deve ser vinculado a regra de negócio (para isso use recurso de Split em vez de chunks ou partitions).

Para execução com processamento massivo de dados, deverá ser montado um plano de particionamento (*partition plan*), para permitir execução em mais de uma instância, caso necessário.

5.7. Relatórios

Em grande parte dos projetos de software existe a necessidade de gerar relatórios e exporta-los em formatos específicos, como HTML, PDF ou excel.
O mecanismo de Relatórios visa padronizar e facilitar a geração de relatórios em aplicações que utilizem a Arquitetura de Referência.

A implementação destes relatórios será realizada com a biblioteca JasperReports e o designer de relatórios JasperSoft Studio[2].

Framework	**Versão/ Release**	**Responsabilidade**
JasperSoft Studio	6.0.1	Ferramenta de Design para relatórios JasperReports
JasperReports Library	6.0.0	Biblioteca para manipulação e exportação de relatórios

6. APÊNDICES

6.1. Apêndice A. Construção

6.1.1. Ferramenta padrão para ciclo de vida de build

Todos os sistemas de informação corporativos, bibliotecas e serviços *devem* utilizar a ferramenta *Maven 3* para o controle do ciclo de vida de *build*. O uso dessa ferramenta, além de prover um padrão, permite que os projetos sejam relativamente independentes de *IDE*.

[2] Desde sua versão 5.5.0 a JasperSoft Studio é a ferramenta oficial de design para JasperReports. O iReport será mantido apenas em forma de manutenção até 31 de dezembro de 2015. O iReport até esta data continuará recebendo correção de defeitos críticos, mas nenhuma nova feature nos release futuros.Dessa forma, a recomendação é manter os relatórios existente e novos já utilizando Jaspersoft Studio.

Para criação de componentes reutilizáveis deverão ser utilizados de arquétipos disponibilizados no repositório de componentes disponibilizado para esta função. O mesmo é válido para subsistemas (sejam web, batch ou outros).

6.2. Apêndice B. Gestão de configuração

6.2.1. B.1. Gerenciamento de Versão

6.2.1.1. Versionamento de código

Em um projeto com desenvolvimento descentralizado onde há participação de várias equipes se faz necessário utilizar um ponto central para o armazenamento do código fonte dos projetos. Hoje, a ferramenta mais robusta para isto é o Git. Porém, seu uso direto é relativamente difícil, se não for intermediado por uma ferramenta visual.

Para o gerenciamento dos repositórios, utilizamos o software GitLab, que está hospedado pela infraestrutura do TRT da 3ª Região e acessível para os outros TRTs que estejam participando do projeto. Com esta ferramenta, é possível criar, visualizar e acompanhar a evolução da base de código dos projetos nela hospedados.

O controle de versão de componentes deverá ser feito através de BOMs (Bill Of Materials) definidos pelos arquitetos do sistema ou do POM parent do projeto (na sessão de DependencyManagement). A criação e alteração da versão tanto de BOMs quanto de POM parent deverá ser controlada pelos arquitetos do sistema.

O processo de versionamento de código e o processo de promoção de novas versões de artefatos são detalhados no documento Plano de Gestão de Configuração.

Ferramenta	**Versão/ Release**	**Responsabilidade**
Git	2.2.2	Controle de versao
Omnibus GitLab	7.3.2	Ferramenta de gestão de repositorios

6.2.1.2. Versionamento de serviços

Apesar do controle de versão das APIs de cada serviço, seja JAX-WS, JAX-RS, JMS ou outro, ser de responsabilidade do Plano de Gestão de Configuração, há definições de versionamento de API que interferem na implementação da arquitetura, e estão demonstradas no código de exemplo de cada tipo de serviço demonstradas nesse documento (a exemplo, vide sessão sobre serviços REST).

6.2.2. B.2 Integração contínua

Em um projeto colaborativo com equipes geograficamente dispersas a construção e integração manual do sistema se torna inviável.

Para solucionar esse problema o TRT da 3ª Região faz uso da ferramenta Jenkins. A ferramenta permite:

- Compilação, execução de testes unitários automatizados e publicação de componentes/subsistemas
- Integração e **publicação** dos sistemas em ambientes de testes e homologação, assim como execução de testes de aceitação automatizados do sistema
- Pode integrar-se à ferramenta de revisão de código (Gerrit) caso esta seja utilizada
- Pode integrar-se à ferramenta de gestão de demandas (Redmine ou outra que for adotada)
- Geração automatizada, a cada construção, de relatório da "saúde" do código, com a integração com o Sonarqube

Além disso, a ferramenta Jenkins, devidamente configurada, pode ser utilizada como ferramenta de **deploy automatizado** e para promoção de artefatos.

O mecanismo de **promoção** de cada versão de artefato gerado é definido no documento Plano de Gestão de Configuração.

Ferramenta	**Versão/ Release**	**Responsabilidade**
Jenkins	1.587	Serviço de integração continua
Analysis Collector Plugin	1.42	Exibe tendência de métricas coletadas via Checkstyle, Dry, FindBugs, PMD, Task Scanner
Blame Upstream Comitters	1.2	Envia notificações para *commiters* das dependências alteradas em caso de falha da *build* sem modificação no código.
Copy Artifact Plugin	1.32	Copia artefatos gerados em outro *job* para o *job* corrente
Credentials Plugin	1.22	Habilita o gerenciamento de credenciais no Jenkins
Downstream build view	1.9	Visualiza graficamente a saúde das *builds* dos *jobs* dependentes do *job* corrente.

Git Plugin	2.3	Suporte ao Git
GitLab Merge Request Builder	1.2.2	Gera builds a aprtir de merge requests no GitLab
GitLab Plugin	1.1.14	Integração com o GitLab
Mailer	1.15	Envio de notificações por e-mail
LDAP Plugin	1.11	Integração com serviço de autenticação via LDAP
Matrix Authorization Strategy Plugin	1.2	Provê autorização por projeto usando matriz de autorização
Promoted Builds Plugin	2.19	Provê recursos para promoção de builds.

A lista de plugins não é extensiva, mas é a configuração mínima para se ter o Jenkins funcionando em um ambiente colaborativo.

6.3. Apêndice C. Automação de Controle de Qualidade e Testes

6.3.1. C.1. Introdução

O controle de qualidade não pode ser totalmente controlado por ferramentas automatizadas, dado que qualidade não é uma medida única e sim um conjunto de características, algumas das quais podem ser avaliadas de forma automatizada, outras não.

C.2. Análise estática de código

No caso de avaliação automatizada de qualidade de código, existem ferramentas maduras no mercado. Uma dessas ferramentas que avaliam o código de acordo com vários critérios objetivos é o Sonarqube, já disponível para uso no projeto, e que deverá ser utilizado por todas as equipes.

É recomendado elencar algumas poucas métricas, mínimos indicadores de qualidade, e depois ir crescendo nos requisitos de qualidade. Inicialmente cada componente/subsistema deverá passar nos seguintes critérios:

Métrica	**Descrição da métrica**	**Critério de aceitação**
StrictDuplicateCodeCheck	Duplicação de código com mais de 100 LOC (desconsiderando comentários)	0% na camada de controle
CyclomaticComplexity	Complexidade ciclomática (McCabe) por método	0% > 15 até 10% > 10 e <= 15

		até 50% > 7 e <= 10
ClassFanOutComplexity	Complexidade Fan-Out de classe	0% > 30 até 20% > 20 e <= 30
LOCsPerClass	Linhas de código por classe	0% > 1000 LOC
LOCsPerMethod	Linhas de código por método	até 10% > 150
DepthInTree	Profundidade de herança	0% > 4 na camada de controle
ClassDataAbstractionCoupling	Quantidade de classes da qual uma classe necessita	0% > 15 20% > 7 e <= 15
LackOfCohesionOfMethodsIV	Medida de coesão. Como ela não avalia corretamente quando há herança de frameworks externos à aplicação envolvida, tem seu uso limitado à camada de controle.	0% > 3 na camada de controle 20% > 1 e <=3 na camada de controle
AfferentCoupling	Mede o acoplamento aferente, ou seja, de quantas classes uma dada classe depende	Somente coleta
EfferentCoupling	Mede o acoplamento eferente, ou seja, quantas classes dependem de uma dada classe.	Somente coleta

Ferramenta	**Versão/ Release**	**Responsabilidade**
SonarQube	4.5.2(LTS)	Coleta, processamento e exibição de métricas feitas a a partir de análise estática de código

C.3. Automação em Testes

Na avaliação automatizada de código, é recomendado adotar uma abordagem em camadas para cobrir toda a extensão de possíveis bugs e testar constantemente num ambiente de integração contínua.

6.3.1.1. Abordagem em Camadas

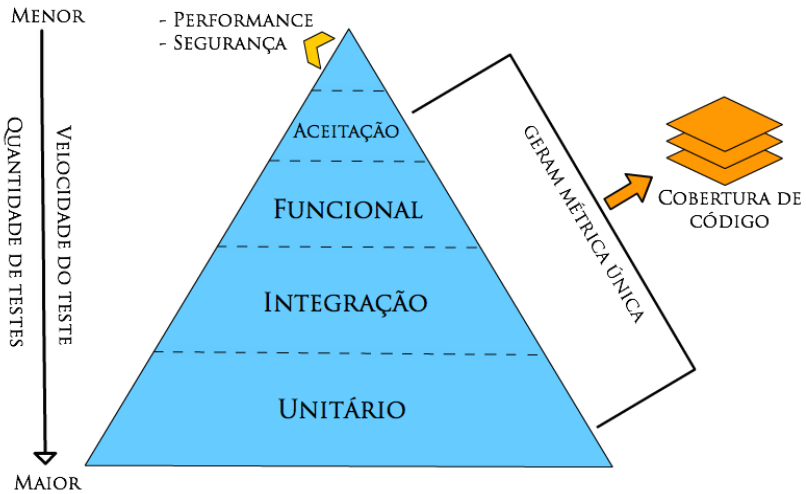
Apesar dos testes funcionais e de aceitação serem necessários, eles têm limitações inerentes:

- Quando realizados manualmente consomem muito tempo.
- Mesmo os scripts bem escritos de testes tornam-se frágeis, e scripts mal escritos exigem manutenção constante.
- Scripts de testes funcionais e de aceitação para grandes projetos podem levar horas ou dias para serem executados na íntegra.

- Testes na interface gráfica, muitas vezes utilizados em testes funcionais e de aceitação, tendem a ser menos confiáveis e a parar por motivos não relacionados com problemas de aplicação.

O segredo para limitar problemas como esses não é fazer menos testes, mas fazer menos desses tipos de testes. As equipes devem realizar mais testes de unidade e de integração, pois eles são mais robusta e menos frágeis do que os funcionais e de aceitação.

A figura abaixo mostra como uma abordagem em camadas pode garantir uma aplicação devidamente validada pela execução de uma grande quantidade de testes de unidade e de integração na camada de classe e de componentes, diminuindo assim a quantidade necessária de testes de aceitação e funcionais nas camadas de subsistemas e de sistema.



A camada da base deve ser de classe e componente onde os desenvolvedores escrevem muitos testes de unidade para o projeto. Estes testes são muito eficazes em assegurar que uma classe ou método específico está funcionando corretamente.

Na camada de integração, as equipes testam a lógica de negócios e integração dos componentes. Estes são testes end-to-end que são projetados para testar o software em um ambiente que se aproxima da produção. Em nível de implementação, esses testes usarão o Arquillian para fazer o deploy de um micro deployment (somente com as classes EJB) para um servidor de aplicação. Dessa forma, os testes terão acesso a todos os mecanismos de injeção CDI e de persistência que as classes de negócio. Também será possível realizar testes sem ser necessário subir um servidor de aplicação, quando estes precisarem apenas do contexto CDI.

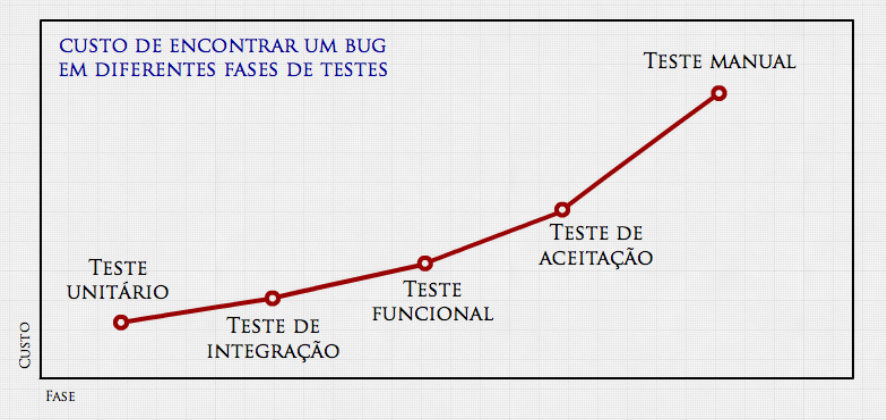
Na camada de testes funcionais, estão os testes de caixa branca, preta e cinza, em cima do deploy completo (EJB e Web) da aplicação. Esses testes existem para garantir que o sistema está de acordo com a especificação de requisitos e que atendem aos requisitos funcionais do sistema (em oposição aos testes de aceitação que servem para garantir que o sistema atende às necessidades dos clientes). Em nível de implementação, esses testes usarão o Arquillian para fazer o deploy de um macro deployment (o mais próximo do deploy de

produção possível) para um servidor de aplicação e posteriormente são rodados os testes em cima dessa infraestrutura.

Os testes de aceitação estão vinculados a uma história de usuário específica e verificam, através de cenários, se as funcionalidades estão corretas e alinhadas às necessidades reais do cliente. Esses testes são um pouco mais lentos, pois, em geral, eles são executados em cima da interface gráfica (caixa preta), mas eles devem ser portáteis o suficiente para serem executados no ambiente do desenvolvedor. Eles podem usar a virtualização de serviços para eliminar dependências em serviços externos, lentos ou difíceis de implantar. Esta camada de teste é um ajuste natural para o desenvolvimento orientado a comportamento (BDD). Como essa camada está mais ligada ao cliente, deve ser utilizado uma linguagem natural para especificação dos cenários (Gherkin - https://github.com/cucumber/cucumber/wiki/Gherkin).

No topo da pirâmide estão os testes especializados: de desempenho e segurança. Estes testes exigem máquinas de produção, com grandes conjuntos de dados e levam mais tempo para ser executado. Devido a isso, eles são executados com menor frequência. Esta camada não é exaustiva, podendo haver mais testes como, por exemplo, teste de usabilidade e acessibilidade.

É importante ressaltar que todas as camadas são importantes para garantir a qualidade do código, não há bala de prata em relação a testes. Em geral, quanto mais alto na pirâmide, maior a quantidade de código que somente um teste cobre e, por isso, é mais difícil de depurar e corrigir erros que ocorrem nesses testes. Em oposição, os testes unitários são os que cobrem a menor porção de código por teste (daí a necessidade de maior número desses testes), porém a correção e depuração são mais fáceis e pontuais. A figura abaixo sintetiza essa ideia.



6.3.1.2. Testes contínuos

A implementação de práticas de testes contínuos permitirá a detecção precoce de defeitos, melhorar a qualidade geral e maximizar a velocidade de entrega. Um conjunto abrangente de testes automatizados irá melhorar a produtividade do desenvolvimento, o que permitirá que os erros sejam encontrados e corrigidos enquanto o código ainda está fresco na mente do

desenvolvedor. Além disso, irá garantir que a base de código permaneça estável em todos os momentos. Isso isola as causas potenciais de um defeito, tornando defeitos muito mais fácil de depurar.

6.3.1.3. Unitário

Framework	**Versão/ Release**	**Responsabilidade**
JUnit	4.12	Framework para execução dos testes unitários.
Hancrest	2.2.9.Final	Framework para escrita de asserções de forma declarativa.
Mockito	2.0.3-beta	Framework para criar objetos mock nos testes unitários.
PowerMock	1.6.1	Permite cria mocks de classes final e métodos estáticos.

6.3.1.4. Integração

Framework	**Versão/ Release**	**Responsabilidade**
Arquillian	1.1.6.Final	Trata de todo o gerenciamento de container, deploy e inicialização para que você possa se concentrar em escrever testes próximos a realidade do código de produção. Permite aos desenvolvedores criar facilmente testes de integração, funcionais e de aceitação automatizados para Java middleware.
Weld-SE	2.2.9.Final	Prover contexto de DI (dependency injection) para os testes que não exigem um ambiente JEE full compliant.
ShrinkWrap	1.2.2	API Java para manipulação de arquivos. Através dele é possível gerar os arquivos de deploy para testes em servidores de aplicação.

Arquillian Transaction Extension: JTA	1.0.1.Final	Extensão do Arquillian que permite controlar transações nos testes de integração, dessa forma é possível, por exemplo, realizar um rollback após a execução de um teste.
Arquillian Suite Extension	1.1.2	Extensão do Arquillian que permite executar diversas classes de testes com o mesmo deployment. No modo de execução padrão do Arquillian, é necessário um deploy por classe de teste.

CDI Only: Testes que não exigem um ambiente JEE full compliant, mas apenas o contexto de DI disponível (weld-se)

Arquillian: Testes de EJB (JEE full compliant)

6.3.1.4.1. Deployment com ShrinkWrap

O ShrinkWrap é a API que permite, em tempo de execução, gerar o arquivo "deployable" que será disponibilizado ao WildFly. No caso dos testes de integração, é gerado um micro-deploy contendo apenas as classes de serviço EJB, sem as classes de managed beans nem as telas .xhtml (o .war).

```
/**
  * Método que se integra ao ciclo de vida do Arquillian para realização do
  * deploy dos arquivos necessários para execução dos testes no servidor de
  * aplicação.
  *
  * @return Arquivo elaborado para realização do deploy da aplicação + testes
  *         no servidor de aplicação.
  */
 @Deployment
 public static Archive<?> createDeployment() {

    JavaArchive ejb = ShrinkWrap.create(JavaArchive.class, "platao.jar")
         // aproveitando classe "raiz" para importar todas as classes do projeto
         .addPackages(true, PlataoAllTestsSuite.class.getPackage())
         .addAsResource("test-arquillian-persistence.xml", "META-INF/persistence.xml");

    EnterpriseArchive ear = ShrinkWrap.create(EnterpriseArchive.class, "platao.ear")
         .addAsResource("test-arquillian-application.xml", "application.xml");
    ear.addAsModule(ejb);

    return ear;
 }
```

6.3.1.4.2. Maven Profile: Remote x Managed

Há três modos de execução do Arquillian: Embedded, Managed e Remote. O Embedded executa em runtime uma "simulação" de um servidor de aplicação. Como o propósito principal do Arquillian é escrever testes "reais", essa forma não é recomendada (vide http://arquillian.org/blog/2012/04/13/the-danger-of-embedded-containers/).

O modo Managed e o Remote são bem parecidos, pois ambos funcionam com servidores de aplicação reais. A diferença é que no Managed o Arquillian é responsável por inicializar e desligar o servidor de aplicação. No modo Remote, assume-se que o servidor já está em execução antes do início dos testes.

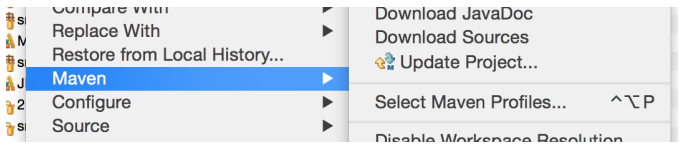
Em resumo, os pontos positivos:

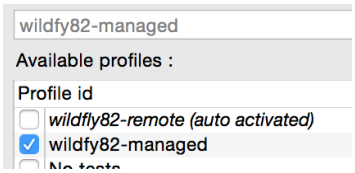
- Managed:
  - Início e download automático do servidor de aplicação (gerenciamento do ciclo de vida do servidor)
- Remote:
  - Velocidade de execução dos testes (não é necessário esperar o servidor iniciar a cada vez que executar o teste)
  - É possível subir o servidor em modo de debug e depurar a aplicação

Diante disso, é recomendável usar o modo Remote em tempo de desenvolvimento, por este ser mais rápido e mais fácil de depurar, e usar o modo Managed em integração contínua, pois facilita o gerenciamento do ciclo de vida do servidor de aplicação automaticamente.

A mudança de modo de execução é determinada por profiles do maven, sendo que o modo Remote é o modo padrão quando não se especifica nenhum profile.

Para mudar o profile para o modo Managed:

- Na linha de comando: mvn test -Pwildfy82-managed
- No eclipse:
  - Clicar com o botão direito no projeto raiz
  - Clicar Select Maven Profiles:
    - 

  - Selecionar o profile wildfy82-managed
    - 

6.3.1.4.3. Controle Transações

Ao executar os testes de integração, é possível indicar que os testes serão envoltos por uma transação JTA. Dessa maneira, pode-se considerar que os dados dos testes vão adotar umas das 4 estratégias:

- ROLLBACK: rollback de transação após cada teste.
- COMMIT: commit de transação após cada teste.
- DISABLED: instrui extensão para não usar transações à execução testes. Útil nos testes de aceitação e funcionais quando estes estão no modo cliente (explicação mais detalhada em tópico dedicado à diferença entre modo cliente e managed de execução de testes Arquillian)
- DEFAULT: mesmo efeito de COMMIT.

A anotação Transactional pode ser definida em um nível de classe ou método, quando esta última tem prioridade se for utilizada.

```
@RunWith(Arquillian.class)
@Transactional(value = TransactionMode.ROLLBACK)
public abstract class DeployableTestBase {
```

6.3.1.4.4. Compartilhamento de Deployables

No modo de execução padrão do Arquillian, é necessário um deploy por classe de teste. Isso muitas vezes é desnecessário e, por conta disso, será usado uma extensão para minimizar o downtime gerado por esse comportamento padrão. A extensão utilizada será a Suite Arquillian Extension que permite executar diversas classes de testes com o mesmo deployment. Para isso basta anotar a classe que tem o método de deploy com @ArquillianSuiteDeployment.

```
@ArquillianSuiteDeployment
public class PlataoDeployableTestBase extends DeployableTestBase {

        @Deployment
        public static Archive<?> createDeployment() {
               …
          }
}
```

Também é possível gerar mais de um deploy e indicar no teste qual deploy deve ser usado no teste.

```
@ArquillianSuiteDeployment
public class Deployments {

        @Deployment(name = "normal", order = 2)
        public static Archive<?> generateDefaultDeployment() {
              …
```

```
        }
        @Deployment(name = "extra", order = 1)
        public static Archive<?> generateExtraDeployment() {
             ...
        }
}
```

Quando se usa o multideployment, é obrigatório usar a anotação @OperateOnDeployment("name") nos testes. Ex.:

```
@Test
@OperateOnDeployment("normal")
public void testNormal() {
       // Teste no deployment normal
}

@Test
@OperateOnDeployment("extra")
public void testExtra() {
       // Teste no deployment extra
}
```

6.3.1.5. Testes Funcionais

Os testes funcionais e de aceitação usarão praticamente os mesmos frameworks e tecnologias, o que diferencia um teste de outro é o foco do teste. Enquanto aquele concentra-se em garantir que a aplicação preenche os requisitos que originalmente impulsionaram o projeto, este concentra-se nos casos de uso/cenários e na validação das reais necessidades dos usuários finais.

6.3.1.5.1. Testes caixa-branca

Os testes de caixa-branca funcionais e de aceitação funcionam da mesma forma que os testes de integração, com a única diferença que aqueles tem acesso as classes do projeto web (managed beans action e REST). Todos os teste caixa-branca são executados no modo In-Container, como explicado na seção abaixo.

6.3.1.5.2. Testes caixa-cinza

Framework	**Versão/ Release**	**Responsabilidade**
Arquillian Warp	1.0.0.Alpha7	Permite implementar métodos de teste caixa-cinza.

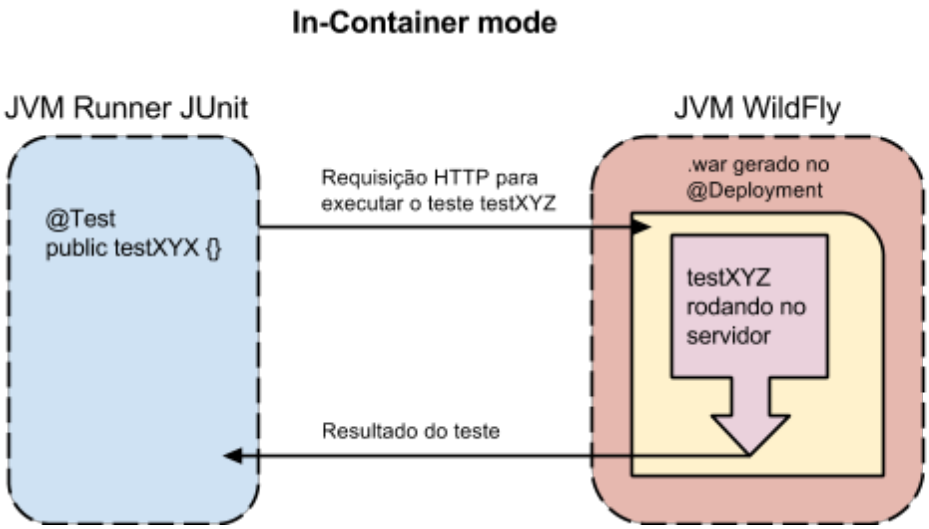
Os testes de caixa-cinza são possíveis de se realizar usando a extensão Arquillian Warp. Ela é responsável por, em um único método de teste, realizar cliques na tela e o assert no servidor.

Antes de entrar no detalhe de utilização do Warp, é necessário explicar os modos de teste cliente e container do Arquillian.

6.3.1.5.2.1. Diferença entre modos de execução Client e In-Container

Quando se executa um teste Arquillian são possíveis 2 tipos de testes: Client e In-Container.

No modo In-Container, os testes são realizados na JVM do próprio servidor, como exemplificado na imagem abaixo.

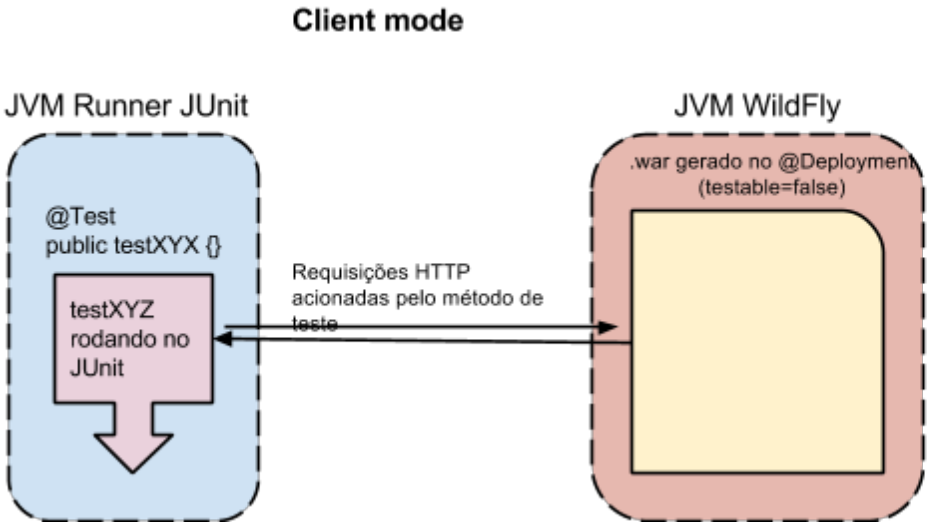


O fluxo de execução de execução de um método de teste nesse modo é o seguinte:

1. É solicitado ao JUnit para se executar um teste com Arquillian.
2. O Arquillian faz o deploy do war no servidor.
    a. Isso é feito pelo runner JUnit do Arquillian que identifica o método com a anotação @Deployment e o executa.
    b. Quando esse método é finalizado, o Arquillian adiciona ao .war gerado as classes de suporte para permitir a execução remota dos testes no servidor. É importante notar que, como os testes executam no servidor, todas as classes e bibliotecas de testes devem ser inseridas no war.
3. Após gerar e enriquecer o war, o deploy é realizado no servidor.
4. Ao fim do deploy, a JVM do JUnit faz uma requisição ao servidor solicitando que o teste seja executado.
5. Ao final do teste, o resultado é retornado à JVM do JUnit através do HTTP response.
6. O último passo é fazer o undeploy do war.

Esse é o modo padrão de execução do Arquillian, não é necessário explicitá-lo no @Deployment, pois ele é equivalente a escrever @Deployment(testable = true).

O segundo modo é o Client. Ao contrário do modo In-Container que reformula e substitui a execução do teste, o modo Client faz o mínimo possível. Ele não remonta seu Deployment nem encaminha a execução do teste para um servidor remoto. Seu caso de teste está sendo executado na JVM do JUnit como esperado e você está livre para testar o sistema do lado de fora, como seus clientes o vêem. A única coisa que faz é Arquillian é controlar o ciclo de vida de seu Deployment.



O fluxo de execução de um método de teste nesse modo é o seguinte:

7. É solicitado ao JUnit para se executar um teste com Arquillian.
8. O Arquillian faz o deploy do war no servidor.
    a. Isso é feito pelo runner JUnit do Arquillian que identifica o método com a anotação @Deployment e o executa.
    b. Não é necessário colocar as classes e bibliotecas de teste nesse deploy, pois os testes não serão executados no servidor.
9. Após gerar o war, o deploy é realizado no servidor.
10. Ao fim do deploy, o JUnit realiza o teste na sua própria JVM, possivelmente acionando a aplicação no servidor através de requisições HTTP.
    a. No método de teste é possível receber por injeção a URL em que foi feito o deploy. Para isso basta injetar a URL:
        i. @ArquillianResource
        private URL urlBase;
11. O último passo é fazer o undeploy do war.

Para ativar o modo Client, há 2 formas:

1. Ativar para todos os testes daquele Deployment:

a. `@Deployment(testable = false)`

2. Ativar por método de teste:
   a.
```
@RunAsClient
public void testXYZ(){...}
```

6.3.1.5.2.2. Arquillian Warp

Com a extensão Arquillian Warp, é possível realizar, em um único método de teste, ações client-side e inspeções server-side.

```
@Test
@RunAsClient
public void pesquisarUsuarioInexistente() {
    Warp
        .initiate(new Activity() {
             @Override
             public void perform() {
                   // WebDriver Selenium
                   servidorCrudPage.pesquisarPorNome("Augusto 123456");
             }
        })
        .inspect(new Inspection() {
             private static final long serialVersionUID = 1L;

             @ManagedProperty("#{servidorAction.resultList}")
             private List<Servidor> resultado;

             @AfterPhase(Phase.RENDER_RESPONSE)
             public void verifyFromEjb() {
                   // assert no servidor
                  Assert.assertTrue(resultado.size() == 0);
             }
    });

}
```

No exemplo, apesar de estar sendo utilizado o modo Client, a classe anônima Inspection será executada no servidor.

6.3.1.5.3. Testes caixa-preta

Os testes de caixa-preta são realizadas usando o modo Client descrito acima. A proposta inicial é de se utilizar os frameworks desenvolvidos pelo time Arquillian para prover os métodos de interação via Browser no sistema. Também é recomendado usar o padrão PageObject para melhorar a manuntenção dos codigos de teste.

Framework	Versão/ Release	Responsabilidade
Arquillian Graphene	2.1.0.Alpha1	API construída em cima do Selenium WebDriver que adiciona diversos recursos para a escrita de testes funcionais reutilizáveis e sustentáveis.
Arquillian Drone	1.3.1.Final	É uma extensão que gerencia o ciclo de vida dos navegadores. Drone simplifica a configuração inicial de teste e prepara testes para integração contínua em uma variedade de navegadores web.
Arquillian Screenrecorder	1.0.0.Alpha1	Extensão para Arquillian que fornece gravação de vídeo e foto da execução dos testes.

6.3.1.6. Aceitação (JBehave/Cucumber-JVM)

Framework	Versão/ Release	Responsabilidade
Cukespace	1.5.10	Permite implementar e executar features do Cucumber no servidor de aplicação usando o framework de teste Arquillian.
Cucumber-JVM	1.1.8	É uma ferramenta para execução de testes automatizados escritos em linguagem natural.

6.3.1.7. Cobertura de código Jacoco Eclemma

Ao rodar os testes de integração, funcionais ou de aceitacão, será gerado a métrica de cobertura de código. Essa métrica será feita pelo Jacoco Eclemma e o relatório da cobertura de código pelo plugin Maven do Jacoco.

O Jacoco funcionará como javaagent do servidor Managed do Arquillian, por isso, só será possível gerar cobertura ao se rodar com o profile Managed. A configuração do javaagent é realizado no arquivo arquillian.xml, na linha <property name="javaVmArguments">.

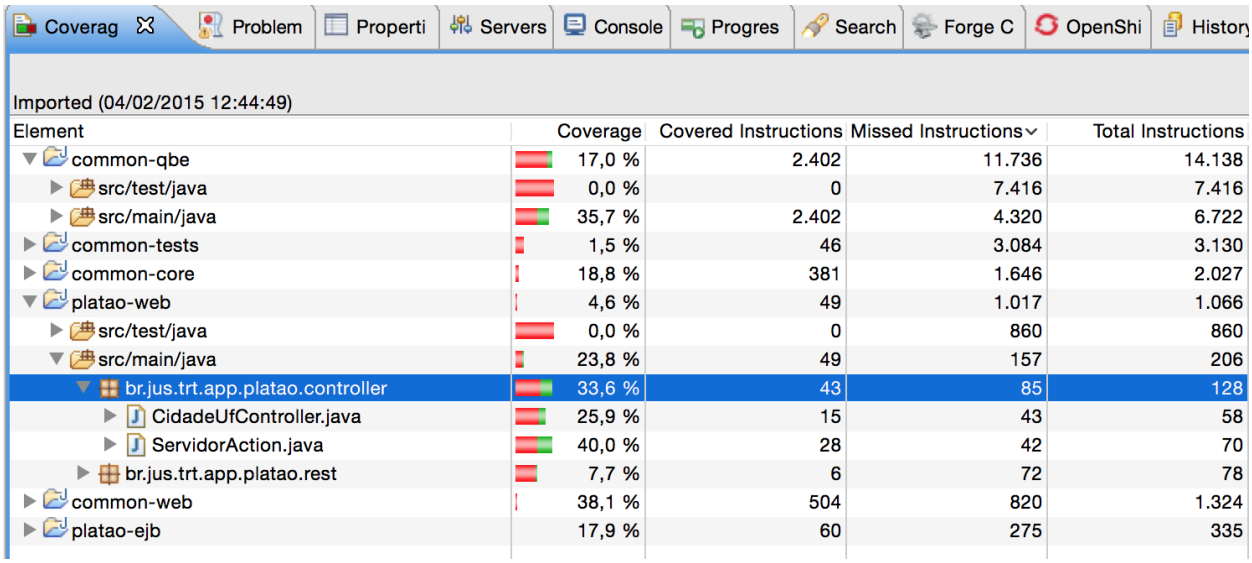
O relatório gerado pelo maven será armazenado na pasta /target/site/jacoco-ut/ e terá o visual parecido com a imagem abaixo:

ServidorAction

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
preLoad(Servidor)		0%		n/a	1	1	5	5	1	1
configLoad(Servidor, Filter)		0%		n/a	1	1	4	4	1	1
getCidadeUfController()		0%		n/a	1	1	1	1	1	1
configSearch(Filter)		100%		n/a	0	1	0	4	0	1
init()		100%		n/a	0	1	0	4	0	1
ServidorAction()		100%		n/a	0	1	0	1	0	1
Total	42 of 70	40%	0 of 0	n/a	3	6	10	19	3	6

Created with JaCoCo 0.7.2.201409121644

A métrica de cobertura também poderá ser importada pelo Eclipse, pelo atalho "Import -> Other -> Coverage Session", escolhendo o arquivo /target/jacoco.exec.



6.3.1.8. Vulnerabilidade

Serão feitos estudos futuros sobre a viabilidade de utilizar o Vega (https://subgraph.com/vega/)

6.4. Apêndice D. QBE - Consulta Dinâmica de Entidades

A consulta de dados em um SGBD pode ser realizada de algumas maneiras utilizando as soluções disponibilizadas pelas tecnologias envolvidas na Plataforma JEE7. As principais são Sql (último caso), HQL e Criteria. As duas últimas opções promovem uma visão orientada a objetos do modelo de dados para a realização de Queries.

Visto que todas estas alternativas possuem estreito laço com os frameworks ORM (*JPA*, *Hibernate*), é estabelecido, nesta proposta arquitetural, que todos os seus recursos devem ser encapsulados e mantidos em elementos da camada de persistência, preferencialmente nos objetos *Repository*. Entretanto, esta abordagem acaba por promover uma considerável proliferação de métodos para consulta de dados na aplicação, dado que, na maioria dos casos, estas operações são criadas para atender a necessidades específicas de funcionalidades e telas do usuário. No trecho de código abaixo considere que todas as operações são requisitos de algumas interfaces com o usuário:

```
interface ServidorFacade {
        List<Servidor> consultarServidores();
        List<Servidor> consultarServidores(Setor setor);
        List<Servidor> consultarServidores(boolean carregarDependentes);
        List<Servidor> consultarServidores(String nome, boolean carregarDependentes);
}
```

É importante diferenciar operações de consultas para interface de operações de consultas que implementam regras de negócio. Estas últimas também existem na fachada de negócio da aplicação, mas implementam uma regra de forma independente de uma interface com o usuário, enquanto aquelas existem apenas para atender a necessidade de uma página ou serviço.

A proposta para a otimização de consultas é a utilização de um recurso que possibilite a construção de queries dinamicamente, separando-a em dois conceitos complementares:

- **Configuração:** Representa as restrições e condições de comportamento que devem ser aplicadas à consulta. São termos genéricos como: Operações e operadores, ordenação, carregamento de associações, etc. Os elementos que implementam este conceito são independentes dos frameworks ORM, portanto é permitida sua utilização em qualquer camada da aplicação.

- **Execução:** Realiza a transformação e aplicação da configuração em objetos concretos do framework ORM estabelecido, garatindo seu funcionamento e integração à camada de persistência, obedecendo aos requisitos arquiteturais.

Estes conceitos permitiriam a utilização de um código mais coeso na camada de negócio, com uma (ou poucas) operação capaz de atender a inúmeras requisições diferentes. O requisitante seria responsável por configurar adequadamente a *query* a ser realizada:

```
interface ServidorFacade {
        List<Servidor> consultar(Configuracao config);
}
```

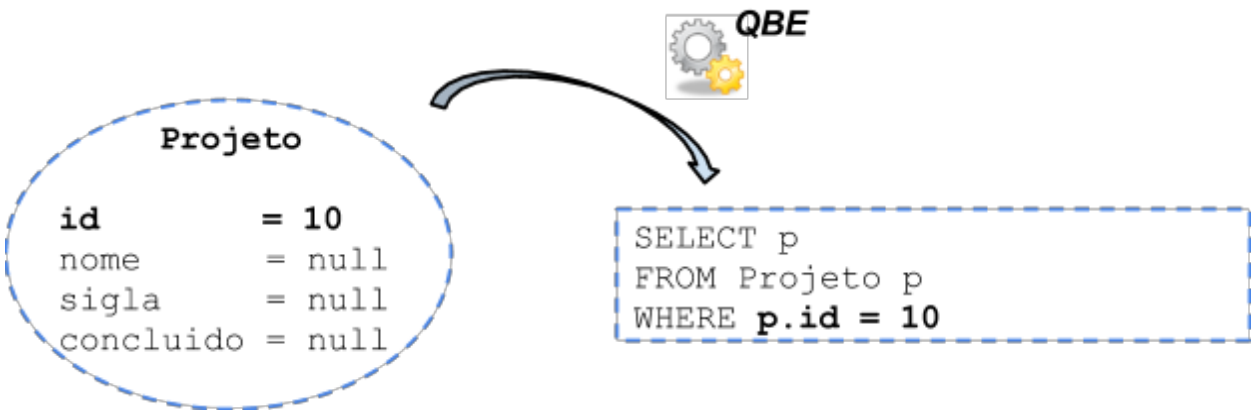
Realizando este conceito, está disponível no framework que suporta esta arquitetura o módulo ***common-qbe***.

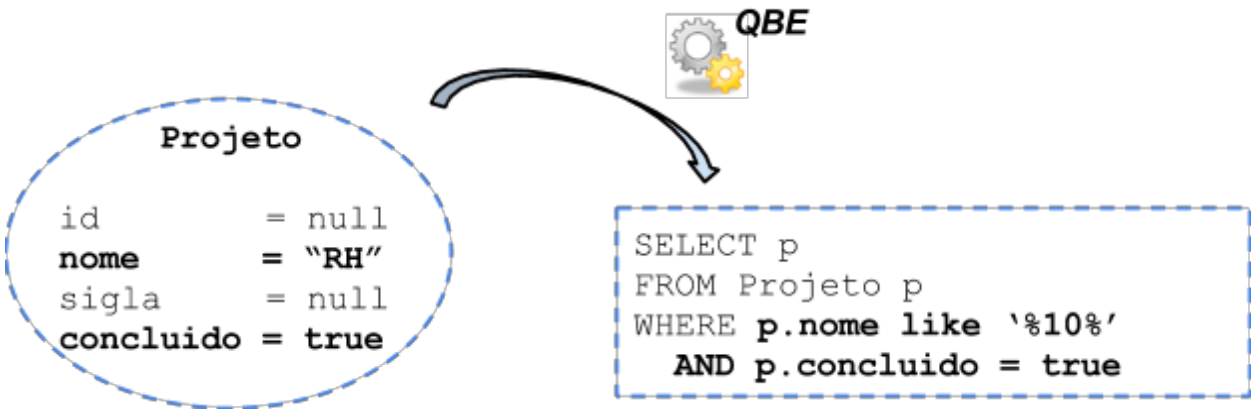
6.4.1. D.1. Princípios Básicos do QBE

O mecanismo do *QBE - Query By Example*, trabalha sobre o conceito de que os parâmetros de uma consulta são dinâmicos, e devem ser definidos o mais próximo possível de sua execução. Até este momento, os parâmetros são considerados como parte da "configuração", e poderiam ser descartados ou adequados antes de se tornarem parte da execução.

A maneira mais simples de se construir uma consulta dinâmica é através de um objeto "***exemplo***", que se trata de uma entidade preenchida, cujos valores dos seus atributos representarão os parâmetros da consulta. Estes atributos são transformados em restrições em tempo de execução, durante a elaboração da Query.

Nas imagens abaixo, é possível verificar a transformação de um objeto *exemplo,* do tipo Projeto, em uma query SQL. Apenas os parâmetros informados no objeto *exemplo* são considerados pelo mecanismo de *QBE*:

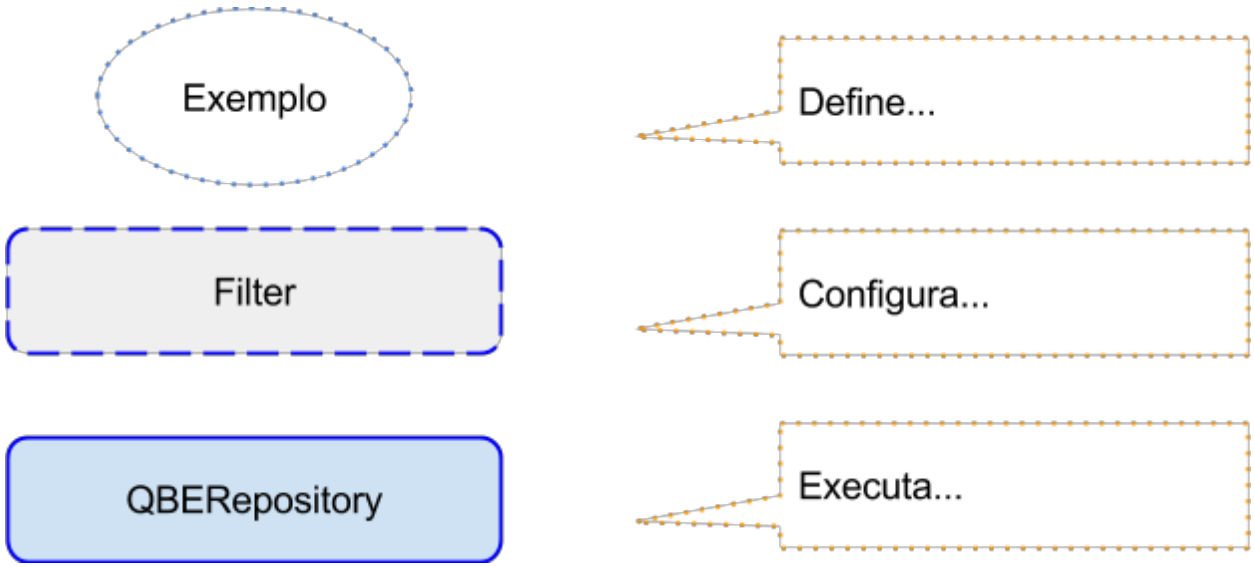




6.4.2. D.2. Implementando QBE: Principais Elementos

Para qualquer consulta a ser realizada com QBE, existem 3 elementos principais a serem conhecidos:

- O ***Exemplo***: Objeto (entidade) com dados para construir a consulta. Seus atributos ***preenchidos*** serão transformados em restrições;
- O ***Filtro***: Encapsula o exemplo, além de outros parâmetros configuráveis para a construção da Query: *Filter.java*
- O ***Motor QBE***: Recebe o filtro e realiza toda a lógica de transformação dinâmica e execução da *query*: *QBERepository.java*



6.4.3. D.2.1. Exemplo

Um *exemplo* é representado por qualquer entidade de domínio, seja concreta ou abstrata. Suas principais responsabilidades são *definir o tipo do objeto de consulta* e *carregar valores para construção das restrições*.

6.4.4. D.2.2. QBERepository

Esta interface é a fachada para as operações de consultas disponibilizadas pelo mecanismo de QBE, e o local mais adequado para sua utilização é encapsulado na camada de persistência, implementando as operações de consultas que se adequam a este conceito. Em casos específicos, onde não existem outras operações de persistência, ela poderia ser utilizada como a representação da própria Camada de Persistência, respeitando todos os princípios desta camada.

As operações disponibilizadas em *QBERepository* são:

- **List<TIPO> search(Filter<? extends TIPO> filter)**: Realiza uma consulta dinâmica a partir da análise das propriedades do filtro recebido, retornando todos os objetos encontrados (do tipo determinado pelo filtro).

- **long count(Filter<? extends TIPO> filter)**: Realiza consulta dinâmica a partir da análise das propriedades do filtro recebido, retornando o total de registros encontrados.

```
Filter<Projeto> filtro = new QBEFilter<Projeto>(new Projeto(nome, sigla));
// o mesmo filtro é utilizado em ambas as operações
List<Projeto> projetos = qbeRepository.search(filtro);
Long numProjetos = qbeRepository.count(filtro);
```

6.4.5. D.2.3. Filter

O *Filter* disponibiliza todas as opções de configuração dinâmica das consultas, sendo a peça mais importante da solução porque é aquela que o desenvolvedor deverá interagir com maior frequência.

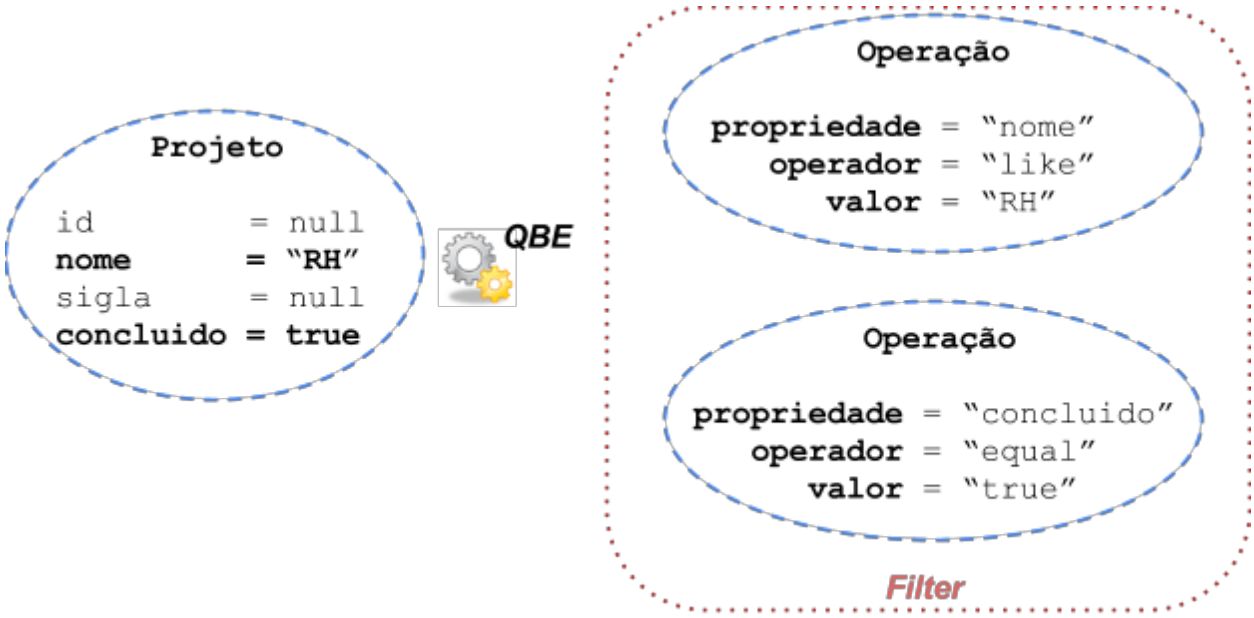
Um filtro pode ser instanciado a partir de um objeto exemplo ou apenas do tipo da entidade exemplo. Quando associado a um objeto, os atributos deste serão utilizados na definição das restrições:

```
// baseado em um exemplo
Projeto exemplo = new Projeto();
Filter<Projeto> filtro = new QBEFilter<Projeto>(exemplo);
```

```
// baseado em um tipo
Filter<Projeto> filtro = new QBEFilter<Projeto>(Projeto.class);
```

6.4.5.1. Configurando Operações e Operadores

Qualquer atributo preenchido em um objeto exemplo é transformado em uma ***restrição***, que, por sua vez, determina uma **Operação** a ser aplicada à consulta na base de dados. Por sua vez, todas operação é composta por 3 elementos básicos: *Propriedade, **Operador**, Valor:*



As operações são implementadas pela classe *Operation*, enquanto os operadores devem implementar a interface *Operator*. Inicialmente, são disponibilizados diversos operadores nativos para utilização nas consultas, e são facilmente acessíveis través da classe Operators:

A customização de uma operação pode se dar facilmente de duas formas:

- ***Customizando o Operador***: O valor considerado na restrição será encontrado no exemplo. Caso o atributo não esteja preenchida, a restrição é descartada:

```
// o exemplo contém o valor
Projeto exemplo = new Projeto();
exemplo.setSigla("RH");

// o filtro configura apenas o operador pera uma determinada propriedade
QBEFilter<Projeto> filtro = new QBEFilter<Projeto>(exemplo);
filtro.filterBy("sigla", Operators.equal());
```

- ***Customizando Operador e Valor:*** Quando não há exemplo ou não há valores preenchidos no exemplo, pode-se incluir uma operação diretamente no filtro:

```
QBEFilter<Projeto> filtro = new QBEFilter<Projeto>(Projeto.class);
filtro.filterBy("sigla", Operators.equal(), "RH");
filtro.filterBy("servidores", Operators.isNotEmpty());
filtro.filterBy("concluido", Operators.equal(), true);
```

Também é possível trabalhar operações de propriedades aninhadas facilmente, seja por exemplo ou programaticamente:

```
// preparando o exemplo com um valor de propriedade aninha: cidade.uf.sigla
Servidor exemplo = new Servidor();
exemplo.setCidade(new Cidade(new UF("PE")));
Filter<Servidor> filtro = new QBEFilter<Servidor>(exemplo);

// configurando a operação diretamente para a propriedade aninhada
Filter<Servidor> filtro = new QBEFilter<Servidor>(Servidor.class);
filtro.filterBy("cidade.uf.sigla", Operators.equal(), "PE");
```

6.4.5.2. Paginação da Consulta no Servidor

A paginação da consulta pode ser configurado através dos métodos:

- paginate(Integer firstRegistryIndex, Integer maxNumberRegistries)

```
QBEFilter<Projeto> filtro = new QBEFilter<Projeto>(Projeto.class);
filtro.paginate(0, 50);

//acessando 2 página
QBEFilter<Projeto> filtro = new QBEFilter<Projeto>(Projeto.class);
filtro.paginate(50, 50);
```

- setPagination(Pagination paginacao)

```
QBEFilter<Projeto> filtro = new QBEFilter<Projeto>(Projeto.class);
filtro.setPagination(new Pagination(0, 50));
```

6.4.5.3. Otimização de Consultas

Considerando que é boa prática no contexto do *JPA* mapear as associações com método de carregamento *fetch=LAZY*, o mecanismo QBE disponibiliza uma configuração dinâmica para o forçar o carregamento de associações no modo *EAGER*. Isto traz uma importante vantagem em consultas onde há um pré-requisito que prevê a utilização de dados de atributos completos associados. Esta configuração é alcançada através do método ***addFetch(String...property)***:

```
// carrega todos os servidores de um projeto
QBEFilter<Projeto> filtro = new QBEFilter<Projeto>(new Projeto());
filtro.addFetch("servidores");

// permite o carregamento de diversas coleções
QBEFilter<Servidor> filtro = new QBEFilter<Servidor>(new Servidor ());
filtro.addFetch("projetos", "dependentes");

// Permite carregamento aninhado.
QBEFilter<Servidor> filtro = new QBEFilter<Servidor>(new Servidor ());
filtro.addFetch("cidade.uf");

// Permite carregamento aninhado sobre coleções.
QBEFilter<Projeto> filtro = new QBEFilter<Projeto>(new Projeto());
filtro.addFetch("servidores.servidor.cidade");
```

Por default, a operação *addFetch* realiza uma operação lógica do tipo *Lef Join*, mas é possível configurá-la para funcionar no modo *Inner Join:*

```
QBEFilter<Projeto> filtro = new QBEFilter<Projeto>(new Projeto());
filtro.addFetch(new FetchMode("servidores", JoinType.INNER));
filtro.addFetch("servidores.servidor.cidade");
```

6.4.5.4. Ordenação do Resultado

A ordenação do resultado da consulta pode ser configurado através dos métodos:

```
sortBy(SortConfig... sortings);
sortBy(boolean priority, SortConfig... sortings);
sortAscBy (String... properties);
sortDescBy(String... properties);
```

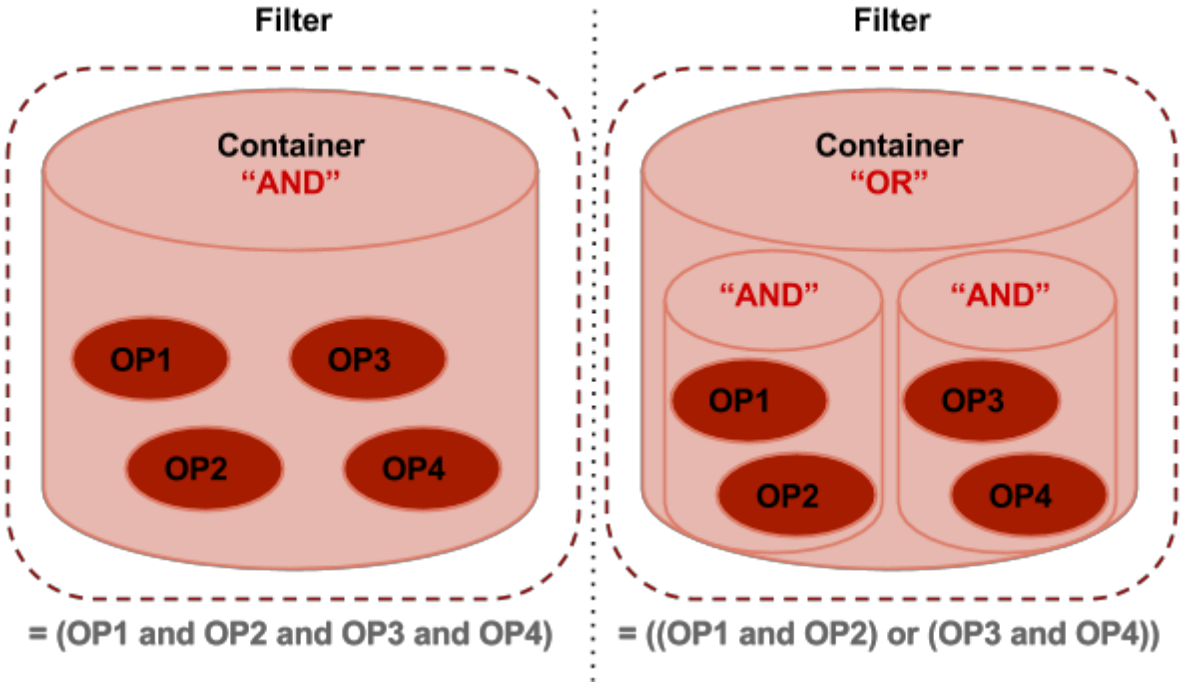
Exemplos:

```
// A ordem dos parâmetros determinam a precedência
QBEFilter<Servidor> filtro = new QBEFilter<Servidor>(new Servidor());
filtro.addAscOrder("dataNascimento", "nome");

QBEFilter<Servidor> filtro = new QBEFilter<Servidor>(new Servidor());
filtro.addAscOrder("dataNascimento");
filtro.addDescOrder("nome");

QBEFilter<Servidor> filtro = new QBEFilter<Servidor>(new Servidor());
filtro.addOrder(OrderConfig.ASC("dataNascimento"), OrderConfig.DESC("nome"));

// Aninhamento também é permitido, mas não em coleções
QBEFilter<Servidor> filtro = new QBEFilter<Servidor>(new Servidor());
filtro.addAscOrder("cidade.uf.sigla", "cidade.nome");
filtro.addDescOrder("nome");
```

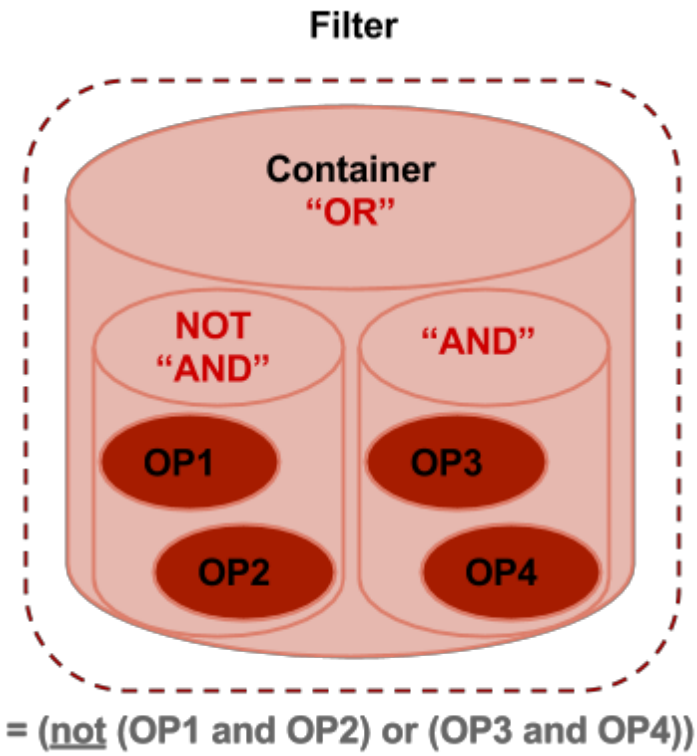
6.4.5.5. Container - Agrupando Operações

Todas as operações configuradas em um *Filter*, quando entram na fase de transformação, são dispostas linearmente utilizando-se o conector *SQL* “AND”. No entanto, em diversas situações, esta solução não poderá atender às necessidades de variações da lógica a ser aplicada em uma consulta. Sendo assim, este mecanismo de *QBE* realiza o agrupamento de todas as operações através de um elemento chamado ***Container***:

Um *Container* é composto por um conjunto de *Operations* e um conjunto de *Containers* (denominados de *subcontainers)*, formando uma estrutura em árvore, determinando a precedência das operações. Cada container também permite a configuração de um operador lógico do tipo AND, OR ou um tipo Customizado. Este operador será aplicado à todas as operações encapsuladas no container:



Um container também pode ser negado. A união destes 3 operações lógicas possibilita a configuração de qualquer operação lógica matemática sobre um conjunto:



Este conceito de *Container* é implemento pela classe *OperationContainer*, juntamento com alguns métodos para manipulação encontrados em *Filter*. Seguem alguns exemplos:

- ***Container Raiz:*** A classe concreta *QBEFilter* possui um container que representa a raiz de toda a hierarquia:

```
// Alterando operador lógico: nome OR sigla
Projeto projeto = new Projeto("RH", "SAAD");
QBEFilter<Projeto> filtro = new QBEFilter<Projeto>(projeto);
filtro.setRootContainerType(ContainerType.OR);
```

- ***Adicionando Sub-Containers:*** Através dos métodos *addOr()* e *addAnd()*

```
/*
 * Neste exemplo 2 propriedades são preenchidas no exemplo: nome, sigla,
 * concluido. As propriedades "nome" e "sigla" são configuradas para
 * pertencer ao container OR, enquanto o restante é mantido no container
 * raiz. Resultado: (concluido AND (nome OR sigla))
 */
Projeto projeto = new Projeto("RH", "SAAD", true);
QBEFilter<Projeto> filtro = new QBEFilter<Projeto>(projeto);
filtro.addOr(new Operation("sigla", Operator.like(true)),
              new Operation("nome", Operator.like(true)));


/*
 * Também é possível criar containers isoladamente e adicioná-los ao
 * Filter. O resultado será o mesmo : (concluido AND (nome OR sigla))
 */
Projeto projeto = new Projeto("RH", "SAAD", true);
QBEFilter<Projeto> filtro = new QBEFilter<Projeto>(projeto);

OperationContainer or = OperationContainer.or();
or.addOperation(new Operation("sigla", Operator.like(true)));
or.addOperation(new Operation("nome", Operator.like(true)));

filtro.addContainerOperation(or);
```

- ***Negando um Container:*** A negação é realizada diretamente no container

```
/*
 * A negação pode ser configurada em qualquer container.
 * O Resultado: (concluido AND NOT(nome OR sigla))
 */
Projeto projeto = new Projeto("RH", "SAAD", true);
QBEFilter<Projeto> filtro = new QBEFilter<Projeto>(projeto);

OperationContainer or =
       filtro.addOr(new Operation("sigla", Operator.like(true)),
                            new Operation("nome", Operator.like(true)));
```

```
or.negate();

/*
 * Também é possível negar o container raiz.
 * Resultado: NOT (nome AND sigla AND concluido)
 */
Projeto projeto = new Projeto("RH", "SAAD", true);
QBEFilter<Projeto> filtro = new QBEFilter<Projeto>(projeto);
filtro.getRootContainer().negate();
```

- ***Customizando um Container:*** É possível customizar um container para encapsular uma lógica booleana (geralmente quando esta é complexa ou repetitiva na aplicação) através da sobrescrita do método ***transfom()***. Exemplo:

```
/*
 * O código abaixo implementa a seguinte lógica booleana: A and (B or C)
 */
QBEFilter<Projeto> filtro = new QBEFilter<Projeto>(new Projeto());
filtro.addOperation("concluido", Operator.equal(), true);
filtro.addOr(new Operation("sigla", Operator.like(true),"SAAD"),
                     new Operation("nome", Operator.like(true), "RH"));

/*
 * Caso esta lógica seja bastante comum em uma aplicação (ou até
 * complexa o suficiente), pode-se encapsulá-la em Container customizado:
 */
public class ABCContainer extends OperationContainer {
       // ... atributos com operadores + construtor
       @Override
       protected OperationContainer transform() {
       // transformando a lista simples de operadores
       // em um container do tipo A AND (B OR C)
       OperationContainer abcContainer = OperationContainer.and();
              abcContainer.addAnd(op1);
                     abcContainer.addOr(op2, op3);
                     return abcContainer;
       }
}

// utilizando o Container
QBEFilter<Projeto> filtro = new QBEFilter<Projeto>(new Projeto());

ABCContainer abc = new ABCContainer();
       abc.addOperation(new Operation("concluido", Operator.equal(), true),
                  new Operation("sigla", Operator.like(true),"SAAD"),
           new Operation("nome", Operator.like(true), "RH"));

filtro.addContainerOperation(abc);
```

6.5. Apêndice E. CRUD - Padronização de Solução para Cadastros

Versões futuras deste documento irão contar com regras e guias para reaproveitar código na criação de CRUDs. Além disso, um documento de Padrão de Interface com o usuário está sendo preparado.

SIGEP-SGRH	Versão 1.2
Plano de Evolução da Arquitetura - Roadmap Arquitetural	Data: 28/09//2015

Plano de Evolução da Arquitetura (Roadmap Arquitetural) SGRH-SIGEP

Histórico de Revisões

Data	Versão	Descrição	Aprovado por
20/02/2014	1.0	Versão inicial do roadmap de Arquitetura para o SGRH.	
23/02/2014	1.1	Revisão pelos participantes do grupo de arquitetura	
28/09/2015	1.2	Adequações referentes à 3ª reunião de trabalho do gtPessoalJT	

Classificações do Documento

Meio de Distribuição	(x) Impresso (x) Eletrônico
Reprodução	( ) Parcial (x) Total
Uso	(x) Interno ( ) Público

Sumário

Introdução e objetivos
- Desvantagens de um ambiente heterogêneo
  - Visão de TI
  - Visão de Negócios
- Necessidade de convergência para uma Arquitetura única

Plano de Evolução da Arquitetura - Roadmap Arquitetural - SGRH-SIGEP
- Análise da Situação Atual
- Visão Geral do Plano de Evolução da Arquitetura
- Fase 1
- Fase 2
- Etapa 2
- Etapa 3

Conclusão

Introdução e objetivos

Este documento tem como objetivo guiar a definição de um Plano de Evolução da Arquitetura (*Roadmap* Arquitetural) de longo prazo para o projeto SGRH-SIGEP, buscando a melhor forma de conciliar as necessidades de negócio e as necessidades da TI (Tecnologia da Informação).

Tendo como base a experiência dos envolvidos e a complexidade do sistema de RH, este é um projeto que ocorrerá em várias etapas. Ainda que já exista um sistema pronto, há funcionalidades que ainda não foram contempladas e precisam ser desenvolvidas. Há funcionalidades desenvolvidas pelos Regionais que precisam ser integradas ao sistema, no entanto, foram construídas apoiadas em tecnologias distintas, observando outras arquiteturas internas. A situação atual remete a um ambiente heterogêneo que apresenta desvantagens significativas tanto sob a ótica da TI, como também do negócio.

Desvantagens de um ambiente heterogêneo

Visão de TI

- Maior esforço de gerência, administração e monitoramento da Infraestrutura;
- Possível uso de soluções de mercado antigas com nenhum ou pouco suporte, como versões antigas do servidor de aplicações JBoss. Além disso, essas versões não contam mais com *patches* de segurança, o que gera riscos adicionais.
- Custos adicionais com Licenciamento e Treinamento, bem como soluções e *frameworks* proprietários utilizados na arquitetura interna das aplicações e/ou nos ambientes de desenvolvimento (JCompany, Oracle Forms, Oracle ADF, Borland Delphi 7);
- Dificuldade em otimizar o uso da força de trabalho disponível nacionalmente
    - É possível que um regional tenha força de trabalho disponível, mas não seja capaz de apoiar a evolução de um módulo por não ser capacitada na tecnologia ou arquitetura utilizada por ele.
- Dificuldade em manter um programa de treinamento consistente para as áreas de TI.

Visão de Negócios

- Processos de negócio não otimizados, potencializando o risco de retrabalho;
- Redundância de informações em vários sistemas com grande probabilidade de produzir inconsistências nas informações.
- Dificuldade de interação entre os processos de negócios;
- Risco dos módulos determinarem o fluxo de trabalho, quando na verdade os sistemas informatizados devem apenas auxiliar na execução dos fluxos de trabalho já definidos pelas áreas de negócio;
- Interface com usuário distintas, dificultando a usabilidade, acessibilidade e treinamento;

    - Interface Desktop Windows;
    - Interface Web com layout e componentes de tela distintos entre módulos;
- Necessidade de diferentes logins em módulos diferentes;
- Auditoria e estatística descentralizadas, dificultando a consolidação de informações.

Necessidade de convergência para uma Arquitetura única

Uma arquitetura única é o primeiro passo para a solução dos problemas levantados acima. Ela é vantajosa para o desenvolvimento e, por consequência, para o usuário final, que obterá um produto coerente e intuitivo. Além disso, uma arquitetura única e provada possibilita que a equipe de desenvolvimento tenha maior produtividade, conseguindo entregar aos usuários uma maior número de funcionalidades em um prazo menor.

Plano de Evolução da Arquitetura - Roadmap Arquitetural - SGRH-SIGEP

O Plano de Evolução da Arquitetura, como o próprio nome diz, consiste dos passos para evoluir a arquitetura do SGRH-SIGEP, objetivando uma arquitetura única.
Obter uma arquitetura única a partir de módulos heterogêneos é extremamente complexo. A fim de se reduzir essa complexidade se torna necessário dividir este trabalho em fases e etapas. Nisso consiste o plano de evolução da arquitetura (Roadmap Arquitetural).

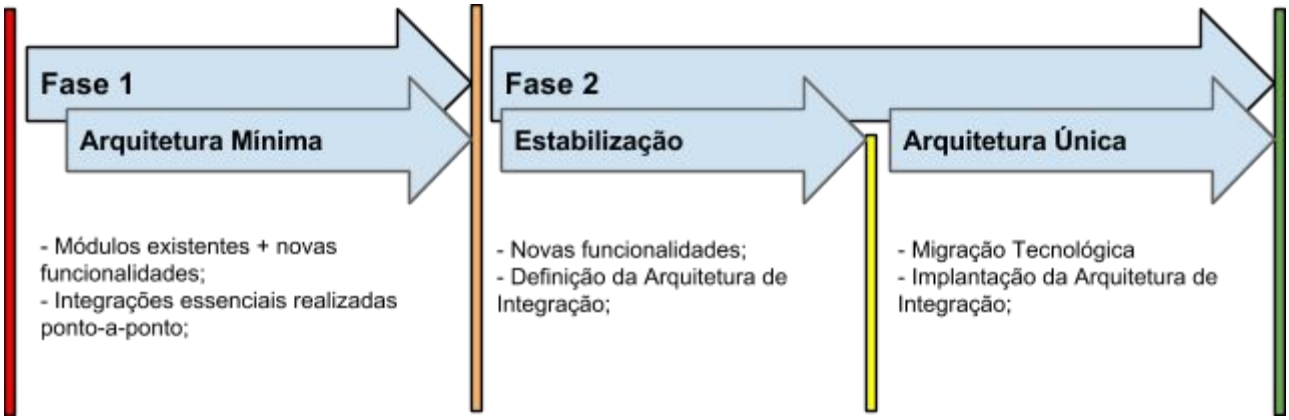
Análise da Situação Atual

Embora ainda não tenha sido realizado um levantamento detalhado de cada módulo que se pretende integrar ao SGRH é de conhecimento que o ambiente e as aplicações são bastante heterogêneas:

- Banco de Dados Oracle 10g e 11g
- Delphi 7
- PL/SQL Web
- Diferentes versões do Java Enterprise Edition (J2EE, JEE5, JEE6, JEE7)
- Diferentes versões do servidor de aplicações JBoss (EAP 4.2, EAP 5.1, EAP 6.3, WildFly 8)
- Diferentes frameworks (Java Struts, Primefaces, Richfaces, Seam 2)

Visão Geral do Plano de Evolução da Arquitetura

O Plano de Evolução da Arquitetura está dividido em duas fases cada qual com objetivo e foco bem definido. A primeira fase foca na integração dos módulos existentes enquanto a segunda fase foca em obter uma arquitetura única. Por tratar se de um conjunto de atividades extremamente complexas e de elevado esforço, consideramos uma fase intermediária para estabilização da solução, que deverá focar nas entregas para o negócio seguindo a arquitetura de referência proposta para a solução.



Fase 1

Nesta primeira etapa, o foco será a integração dos módulos já existentes. A arquitetura interna de cada módulo não será alterada, porém será necessário prover e consumir serviços nas tecnologias definidas pela arquitetura mínima. As integrações serão feitas de forma ponto-a-ponto, todavia as tecnologias adotadas suportam uma evolução futura para utilizar um barramento de serviços.

Não será permitida a integração direta via banco de dados (*shared database*), exceto no caso da utilização de views.

Deve ser minimizada a duplicação de dados entre as aplicações, se escolhendo uma fonte de dados canônica para certos domínios de negócio e a acessando para atualizações e consultas via serviços.

Toda a infraestrutura deverá seguir o que está definido na arquitetura mínima, especialmente versões do banco de dados devem ser padronizadas, assim como a versão mínima do servidor de aplicação.

As novas implementações devem ser preferencialmente feitas na arquitetura de referência. Apenas pequenas correções devem ser executadas nos sistemas existentes.

Fase 2

Estabilização

O foco da fase 2 é intensificar a construção de novas funcionalidades, utilizando a arquitetura de referência. Novas funcionalidades, nesta etapa, devem obrigatoriamente usar a arquitetura de referência.

Será definida uma evolução na arquitetura de integração, com os mecanismos e protocolos que todos os módulos deverão utilizar para o consumo e prestação de serviços internos. Inicialmente, se prevê a adoção de um barramento de serviços, mas deverá ser observado o padrão definido pelo grupo gtIntegraJT, responsável pelas definições de integração no âmbito do GAe (Gestão Administrativa Eletrônica).

As integrações construídas na Fase 1 devem ser migradas para a nova arquitetura de integração.

Um dos serviços prestados pela infraestrutura de integração será o Single Sign On (SSO). Ele deverá integrar os mecanismos de autenticação dos módulos, para que os usuários possam logar-se apenas uma vez e ter suas credenciais compartilhadas entre os demais módulos. Essa característica poderá ser também utilizada nas demais aplicações da solução GAe.

A centralização das informações de auditoria e estatística possibilitará maior segurança, rastreabilidade e gestão dos dados manipulados pelo sistema.

Arquitetura única

Todos os módulos que ainda não estiverem na arquitetura de referência terão sua arquitetura interna migrada para o padrão definido na arquitetura arquitetura de referência. Ao fim do processo, teremos uma *arquitetura única*, integrada. Serão padronizados todos os aspectos técnicos incluindo uma versão única de banco de dados e de servidor de aplicação.

Conclusão

Este documento expôs a sequência de etapas que devem ser seguidas para alcançarmos uma arquitetura única e integrada para o SGRH-SIGEP, obtendo os seguintes benefícios:

- Infraestrutura padronizada, permitindo melhor gerência e otimização do uso de recursos;
- Arquitetura única para o desenvolvimento dos sistemas;
- Redução da barreira para o desenvolvimento colaborativo entre os Tribunais, permitindo melhor uso dos recursos humanos;
- Maior aderência e padronização, em nível nacional, dos processos de negócio dos Tribunais;
- Incremento na usabilidade e produtividade para os usuários finais.

Relatório Técnico

Análise da versão Java EE adotada na Arquitetura de Referência

Histórico de Revisões

Data	Versão	Descrição	Autor	Aprovado por
21/01/2014	1.0	Versão inicial		
21/01/2014	1.1	Correções de Gramática e Autoria		
09/02/2014	1.2	Adequações gerais		

Autoria

Nome	Cargo	Orgão	E-mail
Alexandre A. Drummond	Analista Judiciário, TI	TRT3	alexadb@trt3.jus.br
Augusto Breno de F Lima	Analista Judiciário, TI	TRT6	augusto.breno@trt6.jus.br
David Henrique Sales Vieira	Analista Judiciário, TI	TRT6	david.vieira@trt6.jus.br
Fabiano G. Souza	Analista Judiciário, TI	TRT3	fabianog@trt3.jus.br
Juliano Amorim de Oliveira	Analista Judiciário, TI	TRT3	julianoo@trt3.jus.br

Classificações do Documento

Meio de Distribuição	(x) Impresso (x) Eletrônico
Reprodução	( ) Parcial (x) Total
Uso	(x) Interno ( ) Público

Sumário

- Histórico de Revisões
- Autoria
- Classificações do Documento
- Sumário
- Introdução
  - Breve Histórico Java EE
- Adoção Java EE
  - Ganhos com a adoção da Java EE 7
    - Java Persistence API (JPA 2.1)
    - Java Server Faces (JSF) 2.2
      - Suporte a HTML 5
      - FacesFlow
      - Stateless views
    - CDI 1.1 (Contexts and Dependency Injection for Java EE)
    - JMS 2.0
    - Bean Validation 1.1
    - Java API for RESTful Web Services (JAX-RS) 2.0
    - Batch Applications for the Java Platform 1.0 (JSR 352)
- Servidores de Aplicação
  - Política de ciclo de vida de suporte Red Hat
- Análise de Riscos
  - Adoção de Java EE 6
    - Risco EE6.01: Menor ganho em produtividade em relação ao Java EE 7
    - Risco EE6.02: Necessidade de migração futura para Java EE 7
    - Risco EE6.03: Alto custo e esforço de migração
    - Risco EE6.04: Alto custo e esforço no período de manutenção do sistema durante
  - Adoção de Java EE 7
    - Risco EE7.01: Necessidade de Implantação no WildFly 8.2
    - Risco EE7.02: Problemas no WildFly
    - Risco EE7.03: Liberação de versão EAP do servidor de aplicação
- Conclusões
- Referências

Introdução

A arquitetura dos sistemas do TRT3 e TRT6 é baseada na plataforma JEE - Java *Enterprise Edition*.

A versão atual da especificação da plataforma é a Java EE 7 e a sua antecessora a Java EE 6. A versão Java EE 8 já está em processo de construção no JCP (*Java Community Process*) e tem previsão de ser finalizada até meados de 2016.

O objetivo deste documento é analisar questões relacionadas à adoção da Java EE 7, apontando vantagens da versão atual e vantagens da Java EE 6 e os riscos envolvidos na adoção de uma ou outra versão.

Breve Histórico Java EE

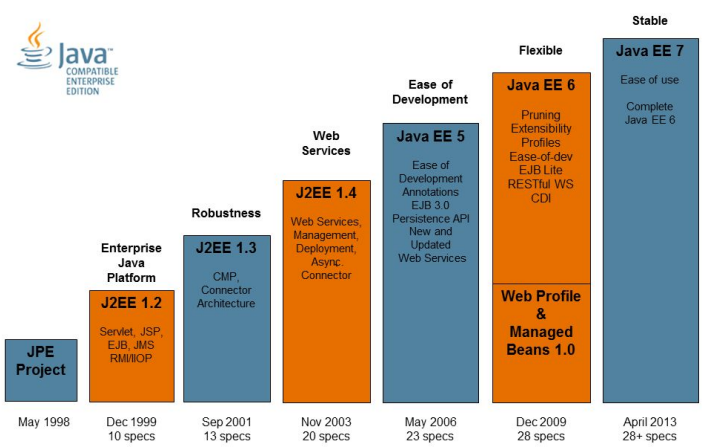
Java EE foi inicialmente focada em prover uma plataforma robusta, com suporte a Web Services e facilidades de implantação. A idéia do Java Enterprise Edition é que o desenvolvedor possa focar nas regras de negócio e que questões de infraestrutura comuns a maioria dos projetos sejam fornecidas pela plataforma (segurança, transações, concorrência, etc.)

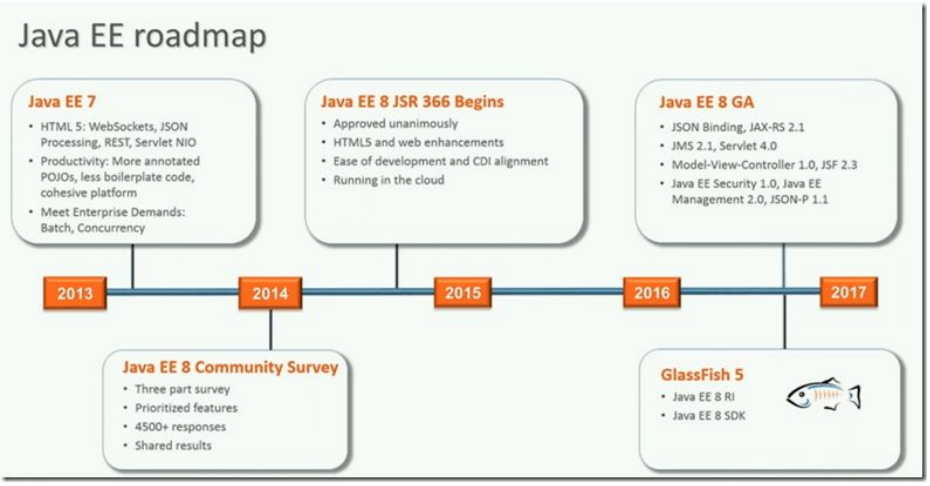
A partir da especificação Java EE 5 o foco se voltou para o aumento da produtividade do desenvolvedor com a introdução de anotações, a nova especificação EJB 3.0 para o desenvolvimento de componentes de negócio, novas especificações para Web Services e melhorias no modelo de persistência.

Java EE 6 racionalizou ainda mais o processo de desenvolvimento e aumentou a flexibilidade da plataforma, permitindo atender melhor aplicações Web leves. Foi introduzido o perfil Web como um subconjunto da especificação Java EE. Além disso, Java EE 6 incluiu frameworks de código aberto a fim de melhorar a integração e criar um ambiente mais padronizado com mais recursos.

Precisamente no dia 12 de junho de 2013, a Oracle lançou oficialmente a versão 7 da Java Enterprise Edition. A Java EE 7 é mais evolucionária do que revolucionária, e conta com atualizações importantes em praticamente todas as especificações que a compõem (Servlets, JPA, Expression Language, CDI, EJB, JMS, JTA, JSP e outras) e também com o lançamento de outras inéditas, como a API para o protocolo Web Sockets, novidade do HTML5; a API para processamento de conteúdo JSON e a API para processamento Batch. A evolução da Java EE 7 é caracterizada especialmente por ganhos em produtividade, desempenho e escalabilidade.

Na visão de *roadmap* da Java EE, a Java EE 7 completa e resolve questões ainda em aberto na Java EE 6. A figura ao lado ilustra esse *roadmap*.





Adoção Java EE

É indiscutível a adoção da plataforma Java EE na arquitetura. As questões ainda a serem discutidas se referem à adoção da versão Java EE 6 ou Java EE 7 como base da arquitetura.

A discussão é fundamentada nas seguintes questões

- Ganhos com a adoção da Java EE 7;
- Servidores de Aplicação Disponíveis;
- Riscos decorrentes da adoção de Java EE 6;
- Riscos decorrentes da adoção de Java EE 7.

Ganhos com a adoção da Java EE 7

A Java EE 7 traz atualizações importantes em suas APIs além de novas APIs como a API para o protocolo Web Sockets do HTML5, a API para processamento de conteúdo JSON e a API para processamento Batch.

São muitas novidades. Num dos maiores eventos de Java, o Java One edição 2013, umas das apresentações discutiu 50 novidades (http://pt.slideshare.net/glassfish/fifty-feature-of).

Apesar de serem muitas e todas elas serem significativas, nesta seção do documento vamos discutir apenas algumas, com o enfoque de analisar o ganho da adoção de Java EE 7. Dessa forma, os itens que serão apresentados são aqueles que objetivam maior produtividade e maior facilidade do desenvolvedor e aqueles que permitem melhor gerenciamento da aplicação em seu ambiente de infra-estrutura em termos de desempenho, escalabilidade e tolerância a falhas.

Um aspecto geral das novidades do Java EE 7 se refere ao fato da nova especificação completar as lacunas deixadas pelo Java EE 6. Muitas especificações que compõem a Java EE 7 são evoluções que proporcionam maior padronização, menos códigos e melhor gerenciamento do

container. Essas características significam maior produtividade em termos de desenvolvimento e menores custos de manutenção dos projetos.

Java Persistence API (JPA 2.1)

JPA irá suportar nativamente chamadas a procedures de maneira transparente, sem a necessidade de usar código específico de nenhuma implementação.

```
//prepara a procedure
StoredProcedureQuery query = EntityManager.createStoredProcedureQuery("processaBoletos");
//registra os parametros da procedure
query.registerStoredProcedureParameter(1, String.class, ParameterMode.OUT); //saída
query.registerStoredProcedureParameter(2, Integer.class, ParameterMode.IN); //entrada
query.setParameter(2, 001); //seta o parametro
query.execute();
String resposta= query.getOutputParameterValue(1);
```

Podemos também invocar qualquer função que esteja disponível no banco de dados, através do novo comando FUNCTION da JPQL.

```
SELECT c
FROM Cliente c
WHERE FUNCTION('EhClienteEspecial', c.id)
```

Os *EntityListeners* da JPA agora também estão integrados com o CDI, permitindo receber por injeção qualquer dependência usando o @Inject:

```
class PedidoListener{
  @Inject
  private GeredorDeNumero gerador;

  @PrePersist
  public void geraNumero (Object o){
    Pedido p = (Pedido) o;
    P.setNumero(gerador.novoNumero());
  }
}
```

Também foi incluído o suporte a operações de *update* e *delete* em lote usando Criteria, além de outras funcionalidades.

As novidades da JPA 2.1 representam ganho em produtividade com menos código e padronização, com código compatível com a plataforma, reduzindo o risco de problemas futuros relativos às interfaces e a dependência direta de implementações de terceiros.

Java Server Faces (JSF) 2.2

Java Server Faces é o framework padrão para a construção de interfaces com o usuário em aplicações Web. Ele já é conhecido por maximizar a produtividade enquanto minimiza a

complexidade de manutenção. As novidades da especificação JSF 2.2 são significativas no sentido de continuar provendo maior produtividade e padronização no desenvolvimento de aplicações Web. Vamos discutir as principais novidades.

Suporte a HTML 5

O HTML 5 é o novo padrão para aplicações web e navegadores. Entre as novidades do HTML5 estão os novos atributos para elementos já existentes. Por exemplo o elemento <input> agora suporta valores *search, email*, *url, range, number, date* no seu atributo *type*, o atributo placeholder foi adicionado.

Para um framework baseado em componentes como o JSF, isso pode ser um problema, visto que os componentes JSF existentes precisariam ser atualizados para reconhecer os novos atributos. Para componentes que precisam explicitamente suportar tais atributos, uma atualização faz sentido. Mas em muitos casos, um componente não precisa fazer nada além de incluir o novo atributo no HTML que produz, ou seja, deixando diretamente a cargo do cliente. Os atributos pass-through do JSF 2.2 fazem exatamente isso, com uma abordagem padronizada, sem a necessidade de componentes de terceiros.

As vantagens do HTML5 e, por consequência, do JSF que o suporta, são indiscutíveis: maior padronização e suporte dos navegadores, incremento do uso de componentes nativos dos navegdores, melhor usabilidade, melhor acessibilidade, suporte a dispositivos móveis.

FacesFlow

Na maioria das aplicações existe um conceito de fluxo que leva o usuário por uma série de telas, por exemplo um assistente, uma subscrição, um cadastro de currículo. Nesse tipo de aplicação é um ponto inicial e uma sequência de passos até atingir um objetivo, ou seja, um fluxo.

A implementação desses fluxo em aplicações Web tem utilizado diferentes técnicas, desde a simples ligação entre páginas utilizando o escopo de sessão para repassar dados entre as requisições. As principais desvantagens dessa abordagem são a dificuldade de reuso e o risco do usuário abrir mais de uma janela (visto os dados estarem gravados no escopo de sessão). Apesar de CDI já possuir o escopo de conversação (@ConversationScoped), essa solução é só parte de um problema maior, solucionado pelo Faces Flow, introduzido na JSF 2.2.

Faces Flow, foi elaborada em cima dos melhores recursos existentes em outros frameworks (exemplos: Spring Web Flow e Oracle ADF) também orientados a fluxos e já consolidados no mercado.

A partir do JSF 2.2 e da Java EE 7, será possível encapsular um conjunto de views/páginas em um pacote independente de todos os demais, totalmente reutilizável e que, sozinho, será capaz de representar uma funcionalidade do sistema. Desta maneira, as aplicações deixam de ter um único e grande fluxo, e essas pequenas unidades passam a ser combinadas para compor uma solução final.

Vale salientar que reusabilidade e que poder dividir a complexidade das tarefas em porções menores, mais gerenciáveis, gera ganhos em produtividade num time de desenvolvimento.

Stateless views

Uma das principais críticas ao JSF desde sua criação foi ser um framework *statefull view*, ou seja, toda a árvore de componentes é mantida na memória do servidor. O JSF 2.x introduziu o *Partial State Saving* no qual apenas mudanças no estado são atualizadas. Apesar disso ser bastante eficiente e utilizar uma quantidade de memória bem pequena para armazenar estado, ainda existem vantagens em **não** manter estado.

Em um ambiente clusterizado, evita-se a necessidade de replicar o estado por meio de múltiplos nós de um cluster ou de usar *sticky-sessions* para garantir que todas as requisições retornem ao mesmo nó de onde se originou a conversação. Isso é importante visto que grande parte das aplicações corporativas (por exemplo as aplicações do Tribunal) devem utilizar redundância, seja para melhorar o desempenho e a escalabilidade da aplicação ou simplesmente para ser tolerante à falhas. Outra vantagem é o desempenho e o consumo de memória menor.

A implementação stateless do JSF é objetiva. O *TagHandler* para o componente **<f:view>** agora tem o atributo booleano **transient**, passado para o método **UIViewRoot#setTransient**. Se a página estiver configurada para ser transiente, o *StateManager* do JSF não armazenará nenhum dado relacionado àquela view. E durante a fase de restauração da view, a mesma é criada sem nenhum estado aplicado a ela.

Componente nativo para Upload

Desde a primeira versão do JSF uma das críticas era a falta de um componente e suporte nativo para upload. Para isso era necessário utilizar componentes de terceiros com comportamento que nem sempre se integravam perfeitamente ao restante da aplicação. O JSF 2.2 trouxe esse componente nativo.

Visto que JSF 2.2 se integra melhor com as demais APIs da Java EE agora é possível utilizar a interface javax.servlet.http.Part, introduzida na Servlet 3.0 para realizar o upload. Existe ainda a novidade de realizar upload utilizando AJAX.

Ainda existem diversas novidades do JSF 2.2 que poderiam ser discutidas, mas não é objetivo desse documento. Para finalizar a seção sobre o framework Web vale destacar que a exemplo da melhor integração com Servlets 3.0, JSF 2.2 se integra melhor com CDI e EJB, pois agora é possível injetar recursos em praticamente qualquer componente JSF, como Conversores e Validadores.

WebSocket 1.0

O Java EE 7 introduziu uma nova API para trabalhar com o protocolo WebSocket, o qual é uma das várias tecnologias que compõem o HTML5.

O protocolo WebSocket muda a forma como os servidores web reagem frente às requisições de seus clientes: em vez de encerrar a conexão, o servidor devolve um status 101 e mantém a

conexão aberta para que ele mesmo ou o cliente possam escrever novas mensagens no fluxo de comunicação. Enquanto o HTTP opera no modo requisição-resposta, o protocolo WebSocket suporta comunicação full-duplex, permitindo que o cliente (normalmente um navegador web) e o servidor possam trocar mensagens simultaneamente.

Para ficar mais claro o ganho obtido com WebSocket podemos pensar numa aplicação de tempo real, multiusuário. As implementações antigas precisavam de mecanismos no cliente que ficassem constantemente verificando se houve mudança no servidor. Com a introdução de WebSockets o próprio servidor é capaz de notificar o cliente. Isso significa menos desperdício de recursos do servidor e de consumo de banda.

CDI 1.1 (Contexts and Dependency Injection for Java EE)

Mesmo com poucas novidades, o CDI 1.1 significa um amadurecimento importante na especificação. CDI 1.1 passa a permitir a injeção de dependência de objetos estáticos e a criação de contextos personalizados de acordo com a necessidade da aplicação;

JMS 2.0

Desde a versão 1.1 do JMS, lançada em 2003, a API não teve mudanças. A versão 1.1 é considerada altamente verbosa devido as capacidades reduzidas do Java SE e Java EE na época do lançamento. O envio de uma mensagem que no JMS 1.1 exigia 13 linhas de código, agora no JMS 2.0 é feito em apenas uma linha. Além disso a nova API elimina a responsabilidade do desenvolvedor em liberar vários recursos, reduzindo os erros programáticos.

```
@Inject
JMSContext context;

@Resource(lookup = "java:global/jms/demoQueue")
Queue demoQueue;

context.createProducer().send(demoQueue, payload); // de 13 linhas para 1
```

Bean Validation 1.1

Com essa atualização os recursos de validação de beans poderão estar integrados com outras especificações da Java EE, como JAX-RS e JAXB, além das anotações de validação que poderão ser usadas nos parâmetros de métodos, como em: public void validar (@NotNull String nome);

```
public void validar (@NotNull String nome){
    //Implementação
}
```

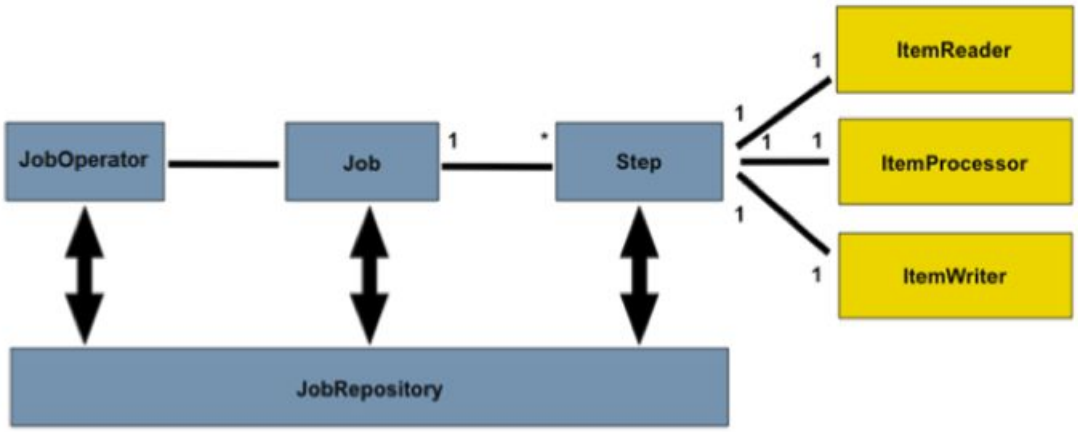
Java API for RESTful Web Services (JAX-RS) 2.0

Passa a dar suporte completo a serviços RESTful (Representational State Transfer). Isso significa que:

- suporta padrão HATEOAS (Hypermedia as the Engine of Application State):
    - a que não é mais necessário programar manualmente a criação de cabeçalhos de resposta HTTP com meta-informações sobre o conteúdo da resposta para atender ao protocolo HATEOAS
- suporta negociação de conteúdo através de melhorias nas anotações @Accepts e @Produces permitindo criação facilitada de views plugáveis:
    - não é mais necessário converter programaticamente requisições JSON (JavaScript Object Notation) em objeto do modelo da aplicação e também não é mais necessário converter cada objeto retornado ao cliente do web service novamente para o formato JSON, o que também economiza algumas linhas em cada método de cada serviço;
    - no caso da implementação da Red Hat, o mesmo vale para outros formatos (XML, HTML, TXT, RSS, etc), com compressão opcional (para menor ocupação de banda), sem que precise ser implementada a conversão para cada um desses formatos;
    - possibilita criação muito simplificada de conversores para os mais diferentes tipos de dados (ex: PDF, XLS, ODS, ODT, etc.);
- suporta HTTP assíncrono (Comet):
    - permitindo que a aplicação faça menos uso de CPU e memória e consequentemente aumentando a carga que um dado servidor/container suporta em relação à implementação JEE6;
- suporta a EJB (Enterprise Java Beans) para controle transacional e lógica de negócios;
- suporta Bean Validation 1.1 integrada aos web services:
    - validação de parâmetros usando anotação (suportando validação de tipo, máscaras, etc, de forma declarativa), em vez de lógica de validação programada dentro do serviço;
- melhorias no suporta a web services seguros:
    - permite, sem uso extensivo de código, criar serviços que não precisam guardar estado (nem para informações de sessão, caso seja usado *token-based authentication*), o que possibilita escalabilidade simples (distribuição de carga pode ser feito até por algumas linhas de configuração em um servidor Apache);
    - além do *token-based authentication* genérico para suporte a Single Sign-On (SSO) em web services, a implementação JAX-RS 2.0 da Red Hat dá suporte a OAuth 2.0;
    - melhor suporte à segurança de forma declarativa, com uso de anotações (@RolesAllowed, …) e a serviços de autenticação customizados (SecurityContextauthenticate, ...);
    - a implementação da Red Hat suporta assinatura digital e criptografia com S/MIME e DOSETA nos web services de forma simplificada, garantindo boa segurança aos web serviços com economia de código;
- configuração centralizada e simplificada dos serviços em relação ao JEE6;

- melhor integração do JAX-RS com CDI;

Batch Applications for the Java Platform 1.0 (JSR 352)



As especificações anteriores ao JEE 7 não possuíam nenhum recurso para controle de *jobs*/processamento *batch*, é portanto uma novidade no JEE7. O que esta nova especificação disponibiliza é um conjunto de recursos para criação de aplicações batch em Java com as seguintes características:

- criação de JobRepositories para cadastramento de trabalhos (*jobs*) a serem executados;
- disponibilização de filas de processamento batch controladas por serviços chamados JobOperators, que podem ser distribuídos;
- JobOperators podem receber solicitação de execução de Job diretamente das aplicações ou de serviços como o Quartz (um dos frameworks de agendamento mais conhecidos pela comunidade Java);
- os passos de um job (steps) podem ser agrupados em um workflow (flow e split) configurados via um schema xml chamado JSL (Job Specification Language);
- existe o recurso de herança de job, step, flow e split para reuso/especialização de tarefas, passos ou partes de um workflow;
- passos podem ser paralelizados (particionamento/concorrência na execução de steps);
- controle transacional incluido e customizável, que pode ser por registro, por grupo de registros (chunks de tamanho determinado), por step, ou por critério de negócio (regras de controle de transação criadas pela aplicação);
- recursos para recuperação automatizada de falhas na execução de um Job;

Servidores de Aplicação

Segundo a Oracle existem atualmente 18 servidores de aplicação em conformidade com o *Full Profile* do Java EE 6 e 4 servidores de aplicação em conformidade com o Full Profile Java EE 7 (Fonte: http://www.oracle.com/technetwork/java/javaee/overview/compatibility-jsp-136984.html).

Tendo em vista que os Tribunais Regionais do Trabalho adotam soluções JBoss EAP da RedHat ou JBoss AS da comunidade JBoss, a discussão aqui se restringirá a tais servidores de aplicação.

As versões mais recentes que suportam o Java EE 6 são JBoss AS 7.1-Final e JBoss EAP 6.3. A diferença básica é que o JBoss EAP conta com suporte da Red Hat por meio de um programa de subscrição pago.

A versão mais recente que suporta o Java EE 7 é o WildFly 8.2-Final, também conhecido como JBoss AS 8. Até o momento não existe versão do JBoss EAP que suporte o Java EE 7 e também não existe previsão oficial da Red Hat para liberação de uma versão compatível. Na *issue* de número 471793 publicada pela Red Hat (https://access.redhat.com/solutions/471793) há vários questionamentos relativos ao Java EE 7, mas nenhuma garantia de que a próxima versão do servidor de aplicação seja conforme com Java EE 7 ou de que sua base seja o WildFly 8 (JBoss AS 8), muito embora esse seja o caminho natural e histórico.

Política de ciclo de vida de suporte Red Hat

A fim de guiar o planejamento das empresas que adotam soluções Red Hat, ela divulga sua Política de Suporte e Atualizações de Produtos - https://access.redhat.com/support/policy/updates/jboss_notes.

Para o servidor de aplicações JBoss EAP 6.x as tabelas são as seguintes:

	Life-Cycle Phase		
Description	**Full Support**	**Maintenance Support**	**Extended Life Support**
Unlimited-incident technical support1	Yes	Yes	Yes
Asynchronous Security Patches	Yes	Yes	Yes4
Asynchronous Bug-Fix Patches2	Yes	Yes	Yes
Minor Releases	Yes	No	No
Software Enhancements	Yes3	No	No
New Certifications (JVMs, DBs, etc.)	Yes	Yes	No

Red Hat JBoss Enterprise Application Platform								
			Full Support		Maintenance Support		Extended Life Support	
Family	**GA**	**Life Cycle**	**Start**	**End**	**Start**	**End**	**Start**	**End**
6.x	Jun 2012	Long-life	Jun 2012	Jun 2016	Jul 2016	Jun 2019	Jul 2019	Jun 2022
5.x	Nov 2009	Long-life	Nov 2009	Nov 2013	Dec 2013	Nov 2016	Dec 2016	Nov 2019
4.3	Jan 2008	Standard	Jan 2008	Jan 2011	Feb 2011	Jan 2013	Feb 2013	Nov 2016
4.2	Jun 2007	Standard	Jun 2007	Jun 2010	Jul 2010	Jun 2012	Jul 2012	Jun 2015

Análise de Riscos

Tendo em vista todas as questões levantadas anteriormente neste documento há duas linhas a serem seguidas no momento para a Arquitetura.

- Adotar Java EE 6 como base da arquitetura, podendo ser implantado em JBoss AS 7.1 ou JBoss EAP 6.3 ou superior;
- Adotar Java EE 7 como base da arquitetura, podendo ser implantado em JBoss WildFly 8.2.

As seções seguintes tem o objetivo de listar riscos relacionados a cada abordagem, a fim de nortear a decisão, que foge ao escopo meramente técnico. Questões relacionados à gestão, produtividade, prazos, custos, esforço e ambiente de infra-estrutura são destacadas.

Vale ressaltar desde já que não estão sendo levantados riscos relativos à adoção de Java EE 7 para a Arquitetura em si. Java EE 7 é uma tecnologia madura. Como frisado anteriormente neste documento ele representa muito mais uma evolução em relação ao Java EE 6. Os riscos que surgiram foram mitigados com provas de conceito. A experiência da equipe de arquitetura envolvida (TRT3 e TRT6) mostra que as novidades do Java EE 7 trarão ganhos de produtividade e que não há grandes novidades que exijam treinamentos da equipe além daqueles voltados ao Java EE 6. A equipe de Arquitetura já prepara treinamento e acompanhamento suficiente para orientar a equipe nas questões relativas às definições arquiteturais e aquelas referentes ao Java EE 7, que porventura se mostrem necessárias.

Adoção de Java EE 6

Risco EE6.01: Menor ganho em produtividade em relação ao Java EE 7

Probabilidade: [ ] Baixa [ ] Média [X] Alta

ID	Dano	Impacto
1	Maior tempo de desenvolvimento e necessidade de soluções de contorno para as lacunas da especificação JEE 6 (completadas pelo JEE 7). Menor padronização do código por ausência de especificações como a API de Batch.	Alto

ID	Ação de Mitigação e Contingência	Responsável
1	Ampliar prazo de entrega do projeto	Gerência de Projeto
2	Aumentar quantidade de desenvolvedores e/ou número de equipes disponíveis	Gerência do Projeto

Risco EE6.02: Necessidade de migração futura para Java EE 7

Probabilidade: [ ] Baixa [ ] Média [X] Alta

ID	Dano	Impacto
1	O Suporte completo do EAP 6.x conforme a Política de Suporte e Atualizações de Produtos da Red Hat termina em junho / 2016. Isso significa o fim de melhorias no produto e o início do ciclo de manutenção em que é recomendada a migração das aplicações Java EE para um novo EAP, que provavelmente será conforme com Java EE 7.	Alto

ID	Ação de Mitigação e Contingência	Responsável
1	Migração da aplicação para Java EE 7	Toda Equipe
2	Migração de toda a infra-estrutura de suporte ao desenvolvimento e ambiente de produção para suportar a aplicação em Java EE 7	Equipe de Infra-estrutura

Risco EE6.03: Alto custo e esforço de migração

Probabilidade: [ ] Baixa [X] Média [ ] Alta

ID	Dano	Impacto
1	Apesar da migração de Java EE 6 para Java EE 7 ser possível, o custo e o esforço necessário para migração cresce à medida que há mais código para migrar.	Alto

ID	Ação de Mitigação e Contingência	Responsável
1	Realizar a migração o mais breve possível, tão logo exista EAP conforme com Java EE 7.	Gerência de Projeto com o Apoio da equipe de Arquitetura
2	Alocar equipe e cronograma do projeto para uma possível migração ainda na fase de desenvolvimento.	Gerência de Projeto

Risco EE6.04: Alto custo e esforço no período de manutenção do sistema durante

Probabilidade: [ ] Baixa [ ] Média [X] Alta

ID	Dano	Impacto

1	Maior tempo para implementação de soluções na fase de manutenção (devido a necessidade de soluções de contorno para as lacunas da especificação, entre outros fatores)	Alto

ID	Ação de Mitigação e Contingência	Responsável
1	Ampliar prazo de entrega de cada manutenção solicitada (seja ela adaptativa, evolutiva ou mesmo corretiva)	Gerência de Projeto
2	Manter uma maior quantidade de desenvolvedores e/ou número de equipes disponíveis para a manutenção do sistema do que a necessária para manutenção do mesmo sistema caso seja utilizada a especificaço JEE7	Gerência do Projeto

Adoção de Java EE 7

Risco EE7.01: Necessidade de Implantação no WildFly 8.2

Probabilidade: [ ] Baixa [ ] Média [X] Alta

ID	Dano	Impacto
1	Para os tribunais que adotam EAP ainda não há versão disponível. Para o desenvolvimento e testes será necessário, portanto o WildFly. Na ocasião da implantação em produção, caso não haja EAP conforme com Java EE 7 disponível ou o sistema que adota a arquitetura ainda não tenha sido homologado para EAP, será necessário implantar em WildFly 8.2.	Alto

ID	Ação de Mitigação e Contingência	Responsável
1	Realizar um conjunto abrangente de testes no WildFly 8.2, incluindo testes de carga e estresse, que possam demonstrar alguma instalabilidade do ambiente.	Equipes de Arquitetura e Qualidade
2	Especializar equipe na administração, configuração e monitoramento do WildFly	Gerência de Projeto

Risco EE7.02: Problemas no WildFly

Probabilidade: [X] Baixa [ ] Média [ ] Alta

ID	Dano	Impacto
1	Qualquer software/sistema está sujeito à problemas e bugs de código. O WildFly é certificado Java EE 7 e está em versão estável, portanto a probabilidade de problemas é baixa.	Alto

ID	Ação de Mitigação e Contingência	Responsável
1	Especializar equipe na administração, configuração e monitoramento do WildFly e até mesmo em detalhes internos da sua implementação, a fim de agilizar a busca por soluções, que também podem contar com o apoio da comunidade JBoss.	Gerência de Projeto

Risco EE7.03: Liberação de versão EAP do servidor de aplicação

Probabilidade: [ ] Baixa [ ] Média [X] Alta

ID	Dano	Impacto
1	Atraso para homologar um novo servidor de aplicação, caso a Red Hat quebre a compatiblidade com o WildFly (JBoss AS 8)	Baixo

ID	Ação de Mitigação e Contingência	Responsável
1	Homologar a aplicação no novo servidor de aplicação, com equipe específica constituída por aqueles que adotam JBoss EAP.	Gerência de Projeto
2	Homologar dois servidores de aplicação.	Time de Desenvolvimento
3	Manter a Arquitetura o mais fiel possível às especificações da JAVA EE 7, minimizando a adoção de frameworks de terceiros.	Time de Arquitetura

Conclusões

A adoção de Java EE 6 é tecnicamente viável como base da Arquitetura. Nenhum requisito funcional e não-funcional deixa de ser atendido numa Arquitetura baseada nesta versão. Como área de visão estratégica para o negócio, o time de Arquitetura irá sempre prezar por maior produtividade e melhor obtenção dos níveis de serviço requeridos pela aplicação, com o menor custo e o menor esforço possível. Na visão dessa área a adoção de Java EE 7 propicia esses ganhos em comparação com Java EE 6.

Apesar de Java EE 6 ser a versão da plataforma Java mais utilizada pelo mercado neste momento, as versões mais novas tendem a levar alguns anos para alcançar um nível considerável de adoção. É esperado que a versão Java EE 7 se torne maioria no mercado corporativo em cerca de 3 anos (MARK DRIVER). Considerando a fase atual de desenvolvimento dos sistemas que adotam a arquitetura (Definição de visão arquitetural; Negociações iniciais para termos do projeto), e o plano de ação para implantação elaborado pela Justiça do Trabalho (2016 até meados de 2020), o Java EE 7 deverá ter alcançado um nível de maturidade alta no início da implantação, e maturidade plena até o final da mesma, sendo a versão Java EE 6 tendendo ao desuso. O mesmo se estende aos Servidores de Aplicação.

As vantagens da adoção do Java EE 7 são claras, como verificado ao longo do documento, se mostrando o rumo natural para um projeto com a duração prevista.

A questão central, que limita a adoção do Java EE 7, se restringe à disponibilidade de servidor de aplicação EAP em conformidade com a especificação. Apesar disso, o uso do WildFly (JBoss AS 8) como servidor de aplicação, inclusive para ambiente de produção, é bastante aceitável. Vale salientar que órgãos públicos, inclusive da própria Justiça do Trabalho, tem feito uso do JBoss AS sem maiores problemas. O impacto, portanto, da indisponibilidade de servidor de aplicação EAP (com suporte), atinge aqueles tribunais que adotam essa solução.

A decisão quanto à adoção de Java EE 7 vai além do escopo técnico do projeto, cabendo ao time de Arquitetura, dentro deste cenário, apenas destacar os riscos de cada solução. Foram destacados os riscos da adoção de Java EE 6 e os riscos de adoção de Java EE 7. Visto que não existe opção sem risco, cabe analisar a solução que apresenta riscos com menor probabilidade de ocorrência, com menor impacto e com ações de mitigação e contingência que estejam aderentes às necessidades do negócio e às estratégias das instituições envolvidas no projeto.

Numa análise técnica, a adoção de Java EE 7 seria, portanto, aquela que oferece menor impacto para o projeto como um todo.

Referências

- JSR-000339 The JavaTM API for RESTful Web Services https://jcp.org/aboutJava/communityprocess/final/jsr339/index.html
- Java EE 7 and JAX-RS 2.0 www.oracle.com/technetwork/articles/java/jaxrs20-1929352.html
- RESTEasy Supported Media Types https://access.redhat.com/documentation/en-US/JBoss_Enterprise_Application_Platform/6.3/html/Development_Guide/RESTEasy_Supported_Media_Types.html
- RESTEasy JAX-RS http://docs.jboss.org/resteasy/docs/3.0.9.Final/userguide/html/index.html
- JSR-000352 Batch Applications for the JavaTM Platform 1.0 Final Release for Evaluation http://download.oracle.com/otndocs/jcp/batch-1_0-fr-eval-spec/
- An Overview of Batch Processing in Java EE 7.0 http://www.oracle.com/technetwork/articles/java/batch-1965499.html
- JSR 356, Java API for WebSocket www.oracle.com/technetwork/articles/java/jsr356-1937161.html
- CON4105 - Java Batch for Cost-Optimized Business Efficiency https://oracleus.activeevents.com/connect/sessionDetail.ww?SESSION_ID=4105
- Java EE Compatibility http://www.oracle.com/technetwork/java/javaee/overview/compatibility-jsp-136984.html
- Conheça as principais novidades no JavaEE 7 http://blog.caelum.com.br/novidades-javaee7-2/
- Going Stateless with JSF https://blog.oio.de/2013/10/02/going-stateless-with-jsf/
- Introduction do Java Platform, Enterprise Edition 7 http://www.oracle.com/technetwork/java/javaee/javaee7-whitepaper-1956203.pdf
- Bean Validation 1.1: validando dados com anotações http://www.devmedia.com.br/articles/viewcomp.asp?comp=30070
- A renovação do JSF - Parte 1 http://www.devmedia.com.br/articles/viewcomp.asp?comp=28879
- A renovação do JSF Parte 2 http://www.devmedia.com.br/articles/viewcomp.asp?comp=29054
- Java JPA: A evolução da persistência em Java http://www.devmedia.com.br/articles/viewcomp.asp?comp=29694
- MARK DRIVER. Hype Cycle for Application Development, 2014. Java Enterprise Edition Version 7. 29/07/2014