

ARQUITETURA DE REFERÊNCIA - GUIA JAVA

Versão: 1.2

Histórico de Revisões			
Data	Versão	Descrição	Responsável
15/04/2019	1.0	Criação do Guia Específico de JAVA	Harrysson Gilgamesh de Medeiros Nóbrega Thiago Adelino de Melo Valdecy Lourenço de Araújo Júnior
23/04/2019	1.1	Homologação do Guia Específico de JAVA	Alexandre Queiroz de Oliveira Carlos Alberto Rodrigues Santana José Arthur Souza de Macedo José Augusto de Jesus Leonardo Moraes Borges Théo Alves Monteiro Valdecy Lourenço de Araújo Júnior
13/05/2019	1.2	Alteração do quadro de métrica de qualidade no item 4 – Metas e restrições arquiteturais.	Alexandre Queiroz de Oliveira Carlos Alberto Rodrigues Santana José Arthur Souza de Macedo José Augusto de Jesus Théo Alves Monteiro Davi Souza Rafael Jesus Nirian Martins S. dos Santos

Sumário

1	INTRODUÇÃO.....	5
2	PROPÓSITO.....	5
3	ESCOPO.....	5
4	METAS E RESTRIÇÕES ARQUITETURAIS	5
4.1	CAMADA.....	7
4.1.1	CAMADA DE RECURSOS.....	8
4.1.2	CAMADA DE NEGÓCIOS.....	8
4.1.3	CAMADA DE ACESSO A DADOS.....	8
4.2	FRONTEND.....	9
4.3	BACKEND	9
4.4	PADRÕES	10
4.4.1	TECNOLOGIA.....	10
4.4.2	COMPONENTE	10
4.4.3	NOMENCLATURA.....	10
5	VISÃO LÓGICA.....	10
6	VISÃO DE IMPLEMENTAÇÃO.....	10
6.1	MICROSSERVIÇOS	10
6.2	DECOMPONDO APLICAÇÕES EM SERVIÇOS	11
6.3	IMPLANTAR MICROSSERVIÇOS DEPENDE DE UM BOM PARTICIONAMENTO	12
6.4	AS VANTAGENS DA ARQUITETURA DE MICROSSERVIÇOS	13
7	VISÃO DE PROCESSOS.....	13
8	VISÃO DE DADOS.....	13
9	VISÃO DE IMPLANTAÇÃO	13
10	TAMANHO E DESEMPENHO.....	14
11	REQUISITOS DE QUALIDADE	14
12	DEFINIÇÕES, ACRÔNIMOS E ABREVIACÕES	16
13	REFERÊNCIAS	17

14	ANEXO A – TECNOLOGIAS.....	18
15	ANEXO B– COMPONENTES	18

1 Introdução

Este documento é uma especialização do **Guia Geral de Arquitetura** e tem natureza subsidiária ao Guia Geral.

2 Propósito

Esse documento tem por propósito fornecer uma visão arquitetural abrangente para ser usada no desenvolvimento de sistemas. Capturar e transmitir as decisões arquiteturais significativas que foram tomadas em relação ao desenvolvimento dos sistemas da instituição. Assim, não tem a intenção de ser um documento definitivo de arquitetura para cada sistema individual. Ao invés disso, tenta traçar os objetivos de alto nível que levem a uma boa arquitetura de software. Cada sistema em particular deve ser analisado junto com a equipe de desenvolvimento e um dos arquitetos de sistemas da CAPES. Se os tópicos aqui apresentados forem seguidos, as chances de sucesso nos projetos tendem ser maiores.

3 Escopo

Fornecer uma visão arquitetural abrangente para ser usada no desenvolvimento de sistemas. Capturar e transmitir as decisões arquiteturais significativas que foram tomadas em relação ao desenvolvimento dos sistemas da instituição.

4 Metas e Restrições Arquiteturais

Na década de 70 os padrões de projetos era um conceito da arquitetura, descrito no livro *A Pattern Language* escrito pelo Christopher Alexander. Nos anos 80 Kent Beck (o criador do Extreme Programming, Test Driven Development e um dos signatários originais do Agile Manifesto) e Ward Cunningham começaram a aplicar esse conceito na área da computação.

A arquitetura em camadas é um estilo arquitetural que propõe dividir um sistema em várias camadas de acordo com a responsabilidade de cada parte do software. Este estilo propõe que:

- As camadas tenham um propósito bem definido;
- Cada camada conheça apenas camadas abaixo dela;
- As camadas possam ser reaproveitáveis.

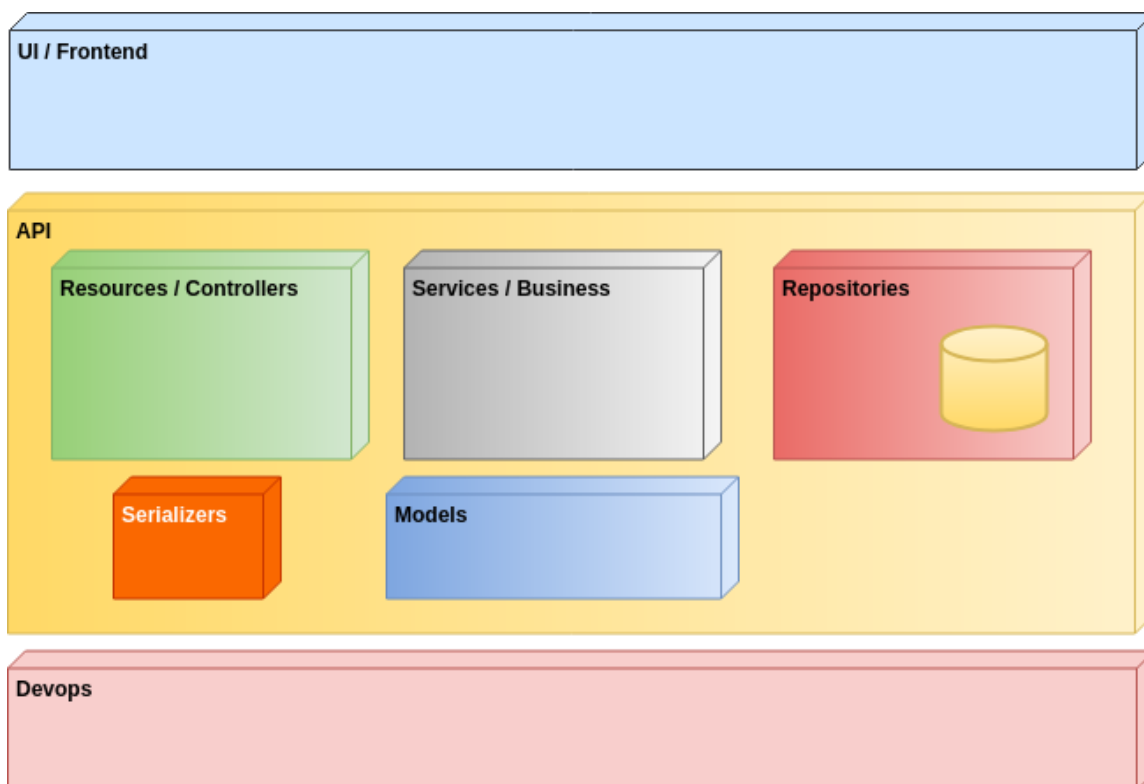


Figura 1 Arquitetura de aplicação Java

A separação de dados e comportamento leva a programação procedural e como consequência a perda de vantagens básicas da orientação a objetos como reutilização de código, encapsulamento, herança e polimorfismo. Esse modelo de programação é conhecido como modelo anêmico e devemos evitá-lo.

É preciso destacar também a importância da cobertura de testes, que traz várias vantagens para o projeto, como a maior qualidade dos sistemas entregues, facilidade de manutenção, diminuição de tempo para encontrar e corrigir problemas entre vários outros benefícios. Todos os novos sistemas devem ter suas principais funcionalidades testadas através de testes unitários ou de integração.

É importante lembrar que cada camada deve ser desenvolvida de forma independente. A camada de negócio não deve mudar seu comportamento, nem deve ser dependente de uma tecnologia que a camada de usuário utiliza. O mesmo não deve ocorrer na comunicação entre cada uma dessas camadas. Isso facilita a implementação dos testes automatizados, visando cada camada em específico.

Os sistemas desenvolvidos deverão utilizar HTTPS nas comunicações com o usuário externo com a finalidade de garantir maior segurança no trâmite das informações.

Para promover a integração entre os sistemas e facilitar a usabilidade os sistemas deverão utilizar o Segurança SSO como mecanismo de login único para as aplicações.

Os indicadores de qualidade que cada sistema deverá apresentar ao ser submetido à avaliação por meio de ferramenta Sonarqube. Cada indicador representa uma meta a ser alcançada. No que tange ao código-fonte, o quadro a seguir relaciona os indicadores e sua respectiva meta.

Grupo	Indicador	Tipo Meta	Meta
PROJETO	Problemas confirmados	Unidades	= 0
	Complexidade	Média total	<= 10
	Métodos	Média total	<= 3
	Índice de manutenibilidade	Nota	A
	Índice de confiabilidade	Nota	A
	Índice de segurança	Nota	A
	Taxa de dívida técnica	%	<= 2,5%
	Classes	Média total	<= 10
	Arquivos	Média total	<= 10
	Linhas duplicadas (%)	%	<= 4%
VIOLAÇÕES	Problemas impeditivos	Unidades	= 0
	Problemas críticos	Unidades	= 0
TESTE	Testes unitários ignorados	Unidades	= 0
	Sucesso em testes unitários (%)	%	= 100%
	Cobertura (camada de negócio)	%	>= 70%

Em alinhamento com o Guia Geral, a arquitetura de aplicações Java é projetada para o desenvolvimento em microsserviços e é agrupada em dois segmentos: API e DevOps.

- API é o agrupamento dos pontos de consumo da aplicação que podem ser consumidos por outros sistemas ou interfaces de usuários especializadas. Constitui a fronteira entre o *backend* e *frontend*.
- DevOps é o conjunto de scripts e artefatos necessários à automação do processo *build/deploy* da aplicação usando, no contexto da CAPES, o OpenShift.

4.1 Camada

Conforme mostrado na *Figura 1 Arquitetura de aplicação Java*, o primeiro seguimento da arquitetura, API, agrupa as camadas que compõe às aplicações nessa linguagem. Nesse contexto, cada camada exerce papel específico e bem definido no plano de codificação. Suas responsabilidades são agrupadas em 3 grupos:

- **Resources Layer / Controllers / Recursos:** Essa camada é responsável por separar a estrutura de acesso dos dados, da camada de apresentação do sistema. Ela centraliza a complexidade requisições e retornos da API Rest. Esta camada adere ao item 3.2.1.1.9 do guia geral.
- **Business Layer / Services / Serviços:** Responsável por centralizar e validar as regras negociais utilizadas para atingir o objetivo da aplicação. Esta camada também tem como objetivo centralizar o acesso a serviços externos, sendo estes em sua maioria serviços web Rest e SOAP. Como exemplo de serviço externo consumido, está o Cadastro de Pessoas, o

Serviço de Documentos e o Serviço de Pessoas da Receita. Esta camada adere ao item 3.2.1.2.3 do guia geral.

- **Repositories Layer / Camada de banco de dados:** Representa o SGBD, ou local de persistência dos dados. Podendo ser um banco de dados relacional (Oracle, Postgresql) ou não relacional (CouchDB, MongoDB). Esta camada adere ao item 3.2.1.1.4 do guia geral.

4.1.1 Camada de recursos

Nessa camada devem ser implementadas o conjunto de rotinas e padrões estabelecidos e documentados da API, ela é o ponto de partida das funcionalidades do microserviço.

Deve ser utilizado o Java EE, ou equivalente no Spring Boot, para descrever a API e expor sua documentação corretamente utilizando o Swagger.

Deve-se criar um ResourceS, que descreve e implementa o acesso à camada de negócios, processa os dados, serializa e retorna um Response contendo o conteúdo do modelo serializado solicitado ou gera uma exceção como Http404.

4.1.2 Camada de negócios

Todo acesso à camada de Banco de dados deve ser feito a partir da camada de negócio, é nela também que são verificadas e validadas as regras negociais, podendo emitir exceções se for o caso. Devem ser consumidos todos os tipos de serviços externos, independente se são serviços web usando Rest, ou SOAP. Essa camada também tem como objetivo abstrair o consumo desses recursos, deixando visível para as outras camadas apenas os parâmetros de requisição e o retorno esperado.

Para as operações que envolvem manipulação de dados é nesta camada que devem ser controladas as transações.

4.1.3 Camada de acesso a dados

Implementação do modelo de dados no Java é definido em conjunto com a equipe de banco de dados e suas boas práticas. Na CAPES os sistemas são totalmente isolados, cada um tem um esquema de banco de dados diferente, um sistema não deve ter acesso à dados de outro sistema via banco de dados, a maneira que estes dados devem ser compartilhados estão descritos no item "Camada de negócio" deste documento.

Todo acesso ao banco de dados no esquema do sistema é feito através da camada de acesso a dados. As consultas DML de recuperação, inclusão, remoção e modificação de informações em bancos de dados devem feitas através de uma técnica de mapeamento objeto relacional que permite fazer uma relação dos objetos com os dados que os mesmos representam (Object Relational Mapper) utilizando a especificação JPA2 implementada pela *framework* Hibernate.

O objetivo da especificação JPA é facilitar o mapeamento dos atributos entre uma base tradicional de dados relacionais e o modelo objeto de uma aplicação, utilizando-se de anotações.

JPA 2 permite também ao desenvolvedor realizar a criação das queries utilizando-se de sua própria API de Criteria orientada a objetos, facilitando a abstração da lógica de consulta, além de implementar a JPA Criteria API, o Hibernate também possui sua própria API de Criteria. JPA 2 também especifica uma DSL mais

próxima do SQL chamada de JPQL. JPQL lida acessando as propriedades do objeto e não as colunas na tabela. O Hibernate além de implementar JPQL tem o HQL: sua DSL que estende SQL onde assemelha-se ao próprio SQL. E por fim o SQL, consultando-se diretamente usando a estrutura presente no banco de dados.

A biblioteca assume o trabalho de automatizar tarefas redundantes enquanto o desenvolvedor permanece no controle de como o banco de dados é organizado e como o SQL é construído.

4.2 Frontend

Não se aplica.

4.3 Backend

Sem prejuízo à definição do Guia Geral 3.1.3. A seguir, é definido como uma aplicação Java será executada em ambiente CAPES.

Com o advento da automatização e devida inteligência na manutenção das aplicações, hoje, é esperado que a aplicação possa atender a picos de demandas com inicialização automática de novos processos, sem afetar seu comportamento.

A boa prática indica que processos de aplicações são *stateless* (não armazenam estado) e *share-nothing*. Quaisquer dados que precisem persistir devem ser armazenados em serviço de apoio *stateful* (armazena o estado), normalmente é usado uma base de dados. O objetivo final dessa prática não faz distinção se a aplicação é executada na máquina do desenvolvedor ou em produção, pois, nesse caso, o que muda é a quantidade de processos iniciados para atender as respectivas demandas.

É importante salientar: ao seguir a prática, uma aplicação não assume que, qualquer item armazenado em cache de memória ou no disco, estará disponível em futura solicitação ou *job* – com muitos processos de cada tipo rodando, são altas as chances de futura solicitação ser servida por processo diferente, até mesmo em servidor diferente. Mesmo quando, rodando em apenas um processo, um *restart* (desencadeado pelo *deploy* de um código, mudança de configuração, ou o ambiente de execução realocando o processo para localização física diferente) geralmente vai acabar com o estado local (memória e sistema de arquivos, por exemplo).

Durante o processo de desenvolvimento de uma aplicação é difícil imaginar o volume de requisição que ela terá no momento que for colocada em produção. Por outro lado, um serviço que suporte grandes volumes de uso é esperado nas soluções modernas. É frustrante solicitar acesso a uma aplicação e ela não estar disponível. Sugere falta de cuidado e profissionalismo, na maioria dos casos. Quando a aplicação é colocada em produção, normalmente é dimensionada para determinada carga esperada, porém é importante que o serviço esteja pronto para escalar. A solução deve ser capaz de iniciar novos processo da mesma aplicação, caso necessário, sem afetar o produto.

A CAPES adotará as seguintes opções de *Deployment* para aplicações Java utilizando-se das soluções de servidor embutido do Spring Boot, como é o caso do Tomcat, Jetty e Undertow. Essa abordagem é recomendada na utilização de *containers*, visto que o servidor de aplicação passa a ser uma parte da aplicação.

4.4 Padrões

Definições adotadas pela CAPES com base em suas necessidades para criação, manutenção e evolução de seus sistemas.

4.4.1 Tecnologia

O Anexo A - Tecnologias elenca as tecnologias adotadas pela CAPES de forma não exaustiva para a pilha de desenvolvimento Java. A inclusão ou remoção de itens nele, deverá ser definida pela CAPES.

4.4.2 Componente

O Anexo B- Componentes elenca os componentes adotados pela CAPES para a pilha de desenvolvimento Java. A inclusão ou remoção de itens nele, deverá ser definido pela CAPES.

4.4.3 Nomenclatura

Nas referências deste documento existe um link para o guia de estilo do Java, definido e utilizado atualmente pela Google. Existe a padronização da codificação do código (UTF-8) e regras de estilo afim de manter o código organizado e legível.

5 Visão lógica

Vide correlação no item 4.1.

6 Visão de implementação

Java é não só uma linguagem de programação, mas uma plataforma de desenvolvimento. Ela possui estruturas de dados de alto nível eficientes, bem como adota uma abordagem simples e efetiva para a programação orientada a objetos. Os grupos de usuário Java são muito utilizados, além de ser possível obter materiais de estudo de forma fácil. Esses grupos são fortes e bem distribuídos, tanto em língua inglesa, quanto na língua portuguesa. Além disso a comunidade de desenvolvimento Java possui uma vasta gama de *frameworks* e bibliotecas. Acelerando o processo de desenvolvimento, utilizando-se do que os outros já implementaram e testaram, para a resolução de problemas.

O Spring Boot é um *framework* vasto que possui várias soluções já implementadas, com a opção de utilização parcial. Engloba desde a parte de autenticação, operações de banco de dados, até a parte de automação de testes.

O Spring Boot também possui solução de servidor de aplicação embutido, permitindo que a aplicação seja executada como uma aplicação Java convencional, sem a necessidade de se realizar o *deploy* através do servidor de aplicação externo.

6.1 Microsserviços

Tradicionalmente, softwares é construído como uma estrutura fechada, com começo, meio e fim. Ou melhor, *back-end* e *front-end*.

Essa filosofia acaba gerando peças muito grandes (Arquitetura Monolítica), normalmente focadas em resolver grandes necessidades e problemas organizacionais. Se você quiser usar somente as funções que

fazem gráficos do Excel para alguma outra aplicação, terá que abrir o Excel e carregar outras funcionalidades desnecessárias naquele momento.

A Arquitetura de Microsserviços, ao contrário das anteriores de Arquitetura Monolítica ou até SOA, destaca-se por explorar a ideia de granularidade, o que facilita a execução do próprio serviço e a adaptação às mudanças.

A ideia é dividir um determinado sistema em serviços acionáveis e modulares, de modo que a união de pequenas partes realize um trabalho maior. Assim, os microsserviços permitem a integração entre vários serviços e a inserção de vários componentes no sistema.

6.2 Decompondo aplicações em serviços

A forma mais usual para escalar uma aplicação é executando várias cópias idênticas de aplicação (por meio de um balanceador de cargas), realizando um processo chamado Decomposição.

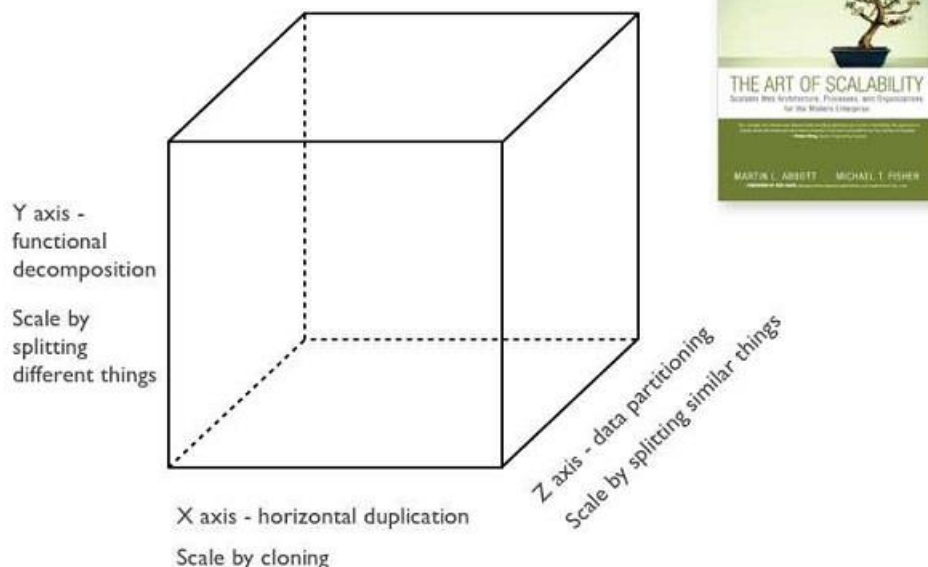
A finalidade da decomposição é resolver problemas comuns na arquitetura monolítica, implicando em que alguns serviços poderão ser muito pequenos enquanto outros serão significativamente maiores.

No conceito de três dimensões, é possível entender como os Microserviços são úteis no particionamento e escalabilidade de uma aplicação monolítica. Temos três eixos de escalabilidade:

- X: referente à escalabilidade horizontal, para ampliar a capacidade e disponibilidade da aplicação (cada servidor executa uma cópia idêntica do código);
- Z: semelhante à do eixo X, mas requer a presença de um componente que se responsabilize pelo roteamento das requisições ao servidor adequado;
- Y: é a terceira dimensão da escalabilidade, a horizontal, denominada decomposição funcional e é responsável por dividir a aplicação em uma série de serviços. A cada serviço corresponde um conjunto de funções (gerenciamento de pedidos, gerenciamento de clientes e assim por diante).

Lembrando que, enquanto o eixo Z divide elementos semelhantes, o eixo Y divide elementos distintos.

3 dimensions to scaling



6.3 Implantar Microsserviços depende de um bom particionamento

A divisão, ou particionamento, do sistema deve ser bem elaborada, segue as abordagens que adotamos para implantar a Arquitetura de microsserviços:

- **Por verbos/casos de uso:** o caso *checkout*, por exemplo, em que um serviço de conclusão de pedidos, ou *Checkout UI service*, implementa a interface com as pessoas que usam o sistema.
- **Por sinônimos/recursos:** como exemplo, considere que, para gerenciamento do catálogo de mercadorias, a empresa pode ter o *Catalog Service*. Nesse caso, o serviço se responsabiliza por todas as atividades que envolvem os recursos/entidades relacionados.

O importante é que cada serviço possua poucas responsabilidades, de acordo com o Princípio de Responsabilidade Única (SRP, Single Responsibility Principle). Ou seja, um serviço exposto deve ter um único e claro papel dentro da Arquitetura.

Se for o caso em que um serviço possui mais de uma responsabilidade, deve-se aplicar algum particionamento, como citado acima.

As funcionalidades Unix constituem outro exemplo de modelagem de serviços, em que cada funcionalidade realiza somente uma operação definida (podendo, no entanto, ser combinada, por meio de *shell script*, com outras funcionalidades a fim de realizar atividades mais complexas). Ao longo dos anos, esse baixo acoplamento facilitou que diversas variações do sistema operacional fossem lançadas, como Ubuntu, Fedora, Solaris e tantas outras.

6.4 As vantagens da Arquitetura de microsserviços

- Os desenvolvedores usufruem de liberdade maior para o desenvolvimento de serviços de modo independente;
- Implantação automática através de ferramentas de integração contínua e código aberto, como Hudson, Jenkins e outras;
- O contêiner web tem inicialização mais rápida;
- Possibilidade de utilizar códigos escritos em linguagens diferentes para diferentes serviços, usando uma “língua franca” para comunicação entre eles (como Json ou XML);
- Oportunidade para os desenvolvedores usarem as tecnologias mais atuais;
- Arquitetura de fácil compreensão e bastante adaptável às mudanças, o que favorece o aprendizado dos profissionais novatos, contribuindo para maior produtividade da equipe;
- Fácil ampliação e integração dos microsserviços com serviços terceirizados, através de APIs, por exemplo;
- Código organizado em função de capacidades de negócio, dando mais visão das ofertas e necessidades dos clientes;
- Mudanças necessárias poderão ser aplicadas somente sobre o serviço específico, sem necessidade de modificar todo o aplicativo. Atualizações de funcionalidades também passam a ser menos complexas;
- Gerenciamento otimizado das falhas (por exemplo, caso um serviço venha a falhar, os outros continuarão trabalhando).

7 Visão de processos

Não se aplica.

8 Visão de dados

As aplicações desenvolvidas na CAPES utilizam como mecanismo de persistência os SGBs relacionais. As aplicações em Java, fazem uso do *framework Hibernate* para acessar os dados dos bancos. São necessários alguns cuidados a fim de garantir que a aplicação seja estável e não venha a causar problemas no acesso aos dados:

- Preferir utilizar JPA Criteria API, ou JPQL, ao SQL puro (evitando dependência de banco). A API de Criteria ou HQL do Hibernate não são encorajados, mas podem ser notados como disponíveis.
- Usar corretamente o Lazy Loading, para trazer a maior quantidade de dados possíveis que realmente serão usados, com a finalidade de evitar 1+n consultas ao banco.
- Sempre que possível usar consultas paginadas para melhorar o desempenho.

9 Visão de implantação

Atualmente, os ambientes não produtivos e produtivos da Capes estão disponibilizados via Openshift, de modo que é possível baixar uma imagem de um contêiner Docker, homologada pela equipe de infraestrutura e iniciar o desenvolvimento da mesma, sem a necessidade que seja provido um ambiente específico para desenvolvimento ou servidor pré-instalado.

Deste modo, o desenvolvedor não necessita configurar ambiente, instalar servidor, ou configura-lo. Tudo está disponível direto na imagem Docker nos repositórios da Capes

Para a geração do pacote de deploy da aplicação, a Pipeline do GitLab se integra com o Jenkins. No próprio GitLab é feita a criação de uma TAG que represente o código usado nessa versão do sistema. A cada execução dessa tarefa é feita uma análise estática de código através do Sonar. E finalmente, o pacote gerado é disponibilizado no contêiner que seguirá todos os passos de aceitação para ser disponibilizado em ambiente produtivo.

Na *wiki*, existe uma página que detalha as configurações que devem ser feitas pelo desenvolvedor na sua máquina e no projeto. A página está disponível em [https://wiki.capes.gov.br/index.php/DTI:Politica de Geracao de Builds para Deploy](https://wiki.capes.gov.br/index.php/DTI:Politica_de_Geracao_de_Builds_para_Deploy).

Cada projeto deverá ter um pacote chamado *devops* em seu repositório, mantido pela equipe de arquitetura e pelo líder técnico da equipe. Neste pacote deve conter os arquivos de CI e CD da aplicação.

10 Tamanho e Desempenho

Caso a aplicação tenha processamento em lote ou execuções agendadas, essas operações devem, preferencialmente, ser realizadas em servidores separados dos servidores destinados a atender a requisições dos usuários. Devido aos custos de processamento destes recursos a resposta do usuário pode ser onerada caso estejam no mesmo servidor.

As requisições devem ser atendidas o mais rápido possível. Processamentos de pedidos de usuários que gerem demora perceptível na request deve ter o seu processamento efetuado em *background* e o usuário deve ser notificado da conclusão da tarefa.

Preferir a separação do *front-end* e do *back-end*, visando facilitar a escalabilidade de recursos e o desacoplamento entre tecnologias com fins diferentes.

11 Requisitos de Qualidade

Todos os sistemas passarão por análise estática de código através do Sonar. Alguns pontos que serão mais cuidadosamente analisados com base nos relatórios do sonar são: cobertura de testes, complexidade ciclomática, acoplamento e coesão.

O acesso ao Sonar é livre para todos os desenvolvedores e deve ser incentivado dentro das equipes como meio de adequação e qualidade de código.

Com a finalidade de garantir uma boa qualidade do código desenvolvido, os desenvolvedores devem usar o BDD (Design/Desenvolvimento guiado por comportamento), que em resumo é uma técnica voltada para o comportamento da aplicação e é comumente usada para testes.

Como forma de auxiliar no desenvolvimento dos códigos, devem ser utilizadas, bibliotecas que facilitem a atividade de teste, entre elas:

- **JUnit**- usado para escrever os testes de forma que permita a sua execução automática.

- **Selenium** - ferramenta usada para mapear a navegação do usuário, usada para escrever os testes de aceitação.
- **Arquillian** - ferramenta usada para executar os testes de integração, auxiliando em testes que precisem de um banco ou recursos de container.
- **Mockito**- usado para realizar mocks e stubs dos objetos, auxiliando o isolamento do código, para testes.
- **Hamcrest**- biblioteca utilizada para auxiliar na escrita de asserções dos testes escritos, melhorando assim a legibilidade do teste.

12 Definições, Acrônimos e Abreviações

- Jenkins - <http://jenkins.capes.gov.br>
- Nexus - <http://nexus.capes.gov.br>
- Sonar - <http://sonar.capes.gov.br>
- GITLab- <http://git.capes.gov.br>
- Wiki - <http://wiki.capes.gov.br>
- Openshift - <http://openshift.capes.gov.br>
- Spring Boot- <http://spring.io>
- JUnit- <http://junit.org>
- Mockito - <https://site.mockito.org/>
- Swagger - <https://swagger.io/>
- Angular - <https://angular.io/>
- React - <https://reactjs.org/>
- Vuejs - <https://vuejs.org/>

13 Referências

- https://wiki.capes.gov.br/index.php/DTI:Politica_de_Geracao_de_Builds_para_Deploy
- https://wiki.capes.gov.br/index.php/DTI:Visao_geral_tecnologias_integradas
- Google Java Style Guide: <https://google.github.io/styleguide/javaguide.html>
- The Art of Computer Programming: D. E. Knuth, Addison-Wesley (volumes 1--3, 4A), 1998.
- DEITEL, Paul J.; DEITEL, Harvey M. Java: como programar. 10. ed. São Paulo: Pearson Education do Brasil, 2017.
- Spring Guides: <https://spring.io/guides>
- Hibernate Docs: <http://hibernate.org/orm/documentation/5.0/>
- Docker para desenvolvedores: Rafael Gomes, Leanpub, 2017

14 Anexo A – Tecnologias

Tecnologia	Versão
Java	>=1.8
Spring-boot	>=2.0
JUnit	>=5
Mockito	>=2
Arquillian	>=1.4.0
Hibernate	>=5

15 Anexo B– Componentes

NOME	VERSÃO	DESCRIÇÃO
-	-	-