GuideLine Engenharia de Software

Princípios, padronizações e melhores práticas são métodos utilizados para entrega de software de alta qualidade e agilidade, pois as pessoas que fazem parte do time sempre se orientam e se baseiam nos mesmos conceitos, o que lhes dá poder para focar no que realmente importa: criar as melhores soluções para os clientes, no tempo certo e com baixo custo. Outra grande vantagem dessa documentação é referente ao momento em que o time cresce, visto que haverá uma fonte única para consulta das diretrizes de Engenharia de Software, fazendo com que haja disseminação de conhecimento e unificação da cultura técnica entre as pessoas, o que reflete na integridade conceitual dos sistemas.

Qualidade de Software

A qualidade do software deve ser assegurada pelo time de desenvolvimento que deve presar pelos testes unitários em primeira instância, código limpo e bem organizado segundo o code style definido para padronização do código, bem como é de fundamental importância a aplicação de padrões de projetos e conceitos fundamentais, como por exemplo o SOLID para os paradigmas de orientação a objetos.

Code Style

O code style tem por objetivo padronizar a formatação do código e deve ser utilizado conforme documentação abaixo:

https://google.github.io/styleguide/csharp-style.html

Teste Driven Development

É de suma importância que o teste seja feito junto ao desenvolvimento das regras de negócio, sendo fundamental que os cenários ora definidos durante o refinamento sejam previamente implementados na forma de testes unitários para cobertura de um código de qualidade.

Recomendações para a escrita dos testes

- Para cada cenário de teste escrito na *User Story*, escreva um teste unitário
- É importante que o nome do método de teste seja claro e explicito de acordo com o cenário a ser testado
- Nunca mock uma dependência interna da aplicação

Sonarqube para Cobertura de Qualidade e Segurança de código

O Sonarqube é executado a cada commit enviado ao repositório remoto pela esteira contínua para que haja um feedback rápido em relação a qualidade e segurança do código, uma dica importante é a utilização do Sonarqube localmente (através de uma imagem Docker), para o feedback seja mais rápido e a correção seja feita antes do envio do commit.

Métricas

As métricas sumarizam o potencial de qualidade e segurança da aplicação.

Contexto	Indicador	Valor
Reliability	Bugs	0
Security	Vulnerabilities	0
Maintainability	Code Smells	0
Security Review	Security Hotspots	0
Coverage	New Lines to cover	>= 80%
Duplications	New Lines	0

Documentação do Sonarqube sobre métricas:
https://docs.sonarqube.org/latest/user-guide/concepts/

Versionamento

O Versionamento segue o padrão proposto pela Via pelo time de Governança: Versionamento.

Inclusão da estrutura de pré-release.

Seguindo o modelo de MAJOR, MINOR e PATCH, há inclusão da estrutura de PRÉ-RELEASE no qual seria como exemplo:

Versão: 1.2.0-3. Em que 1 é MAJOR, 2 MINOR, 0 PATCH e após o hífen(-) o número 3 é o PRÉ-RELEASE.

Guia de Design de API

Considerações Gerais

Uma Application Programming Interface (API) é um contrato entre um fornecedor e um ou mais consumidores. Esse contrato deve incluir o protocolo de comunicação, as operações (consultas e atualizações) e os formatos de dados para entradas, saídas e erros.

Uma API consta de um desenho e uma implementação. Existem vários paradigmas de desenho de APIs como SOAP, REST, RPC, etc. e muitas tecnologias de transporte e formatos de dados, mas o REST com HTTP/JSON se converteu no padrão de fato.

Para o desenho da API REST, consideremos as seguintes restrições:

- Protocolo a utilizar. Sempre utilizaremos o protocolo HTTP (RFC2616) / HTTPS (RFC2660).
- Segurança da API. A arquitetura sobre a qual se apoiam as APIs será a encarregada de garantir a segurança da mesma.
- Formato de entrada e saída de dados. Somente será aceito o JSON como representação dos recursos. Além disso, esse JSON será escrito em **lowerCamelCase**.

O uso apropriado de nomes nas URIs dota de contexto às petições que são realizadas garantindo uma API mais legível e utilizável, sendo um dos princípios do desenho de APIs Rest.

Para o desenho de API será usado a especificação **SWAGGER 3.0**, que permite definir de uma forma fácil, e com componentes reutilizáveis, código simples de entender para qualquer perfil utilizando-se o formato **YAML**.

Domínios

Um **Domínio** dentro do contexto de Microserviços é um agrupamento de entidades fortemente relacionadas dentro de um contexto de negócio, como produtos ou serviços que a organização possui/oferece, ou processos internos.

Esta abordagem foi consolidada por Eric Evans no livro Domain Driven Design (DDD) ou Projeto Orientado a Domínio. DDD é uma metodologia para modelagem de Software focada no desenvolvimento de sistemas com baixo acoplamento e altamente escaláveis. Para atingir este objetivo é necessário segregar o sistema em contextos bem delimitados e dentro de cada contexto podem existir vários Domínios Funcionais**.**

Por exemplo, dentro do contexto de inventário de produtos as informações de produtos, fornecedores e estoque podem estar presentes e fortemente relacionadas ao catálogo. Este **contexto** é **delimitado** pela forma como estes domínios se relacionam, porém a venda de um produto não está diretamente ligada a este contexto. Portanto uma API que realiza a efetivação de uma venda não deveria estar dentro do contexto de inventários.

Um contexto delimitado deve ser aplicado com o intuito de delimitar o comportamento de um domínio funcional, fornecendo um entendimento claro do que é necessário ser desenvolvido isoladamente e o que deve ser compartilhado.

Entidades sob um domínio compartilham cenários funcionais, estruturas de dados e comportamentos comuns. Além disso, poderão utilizar estruturas de dados e comportamentos comuns a todas as entidades, ainda que se deva revisar que estas não sejam funcionalidade de negócio. Alguns exemplos poderiam ser: Dinheiro (valor + moeda corrente), endereço ou pessoa.

As entidades que se repetem em diferentes domínios (por exemplo: Accounts) deveriam ser copiadas para evitar dependências e administrar para que uma delas seja a "principal" (por exemplo: Accounts no domínio Accounts) e que nas diferentes cópias (por exemplo: Accounts no domínio Cards) sejam sempre sub-conjuntos da principal.

A nomenclatura de um domínio deveria consistir em um **substantivo** no plural em maiúsculas: Accounts, Cards, Products, etc.

Para a definição de domínios, será realizada uma API em SWAGGER para cada um deles. Isto permitirá administrar também ciclos de vida diferentes por domínio funcional.

Subentidades

As subentidades como /accounts/v1/loans são projeções parciais da entidade principal para consultar ou atualizar.

No exemplo anterior, a entidade accounts pode ter um array de loans e expor um end-point /accounts/v1/loans para facilitar sua leitura ou atualização parcial.

A nomenclatura de uma subentidade deveria consistir em um substantivo no plural ou singular dependendo se é uma lista de entidades ou de uma entidade.

O **SWAGGER** permite a definição de recursos, que é traduzida ao endpoint neste caso de /accounts/v1/loans. Os recursos dispõem de todas as características próprias de um endpoint, como os cabeçalhos, o método HTTP, etc.

Formato URI (RFC 3986)

URI ou Identificador de Recurso Uniforme é a sequência de caracteres que identifica um recurso físico ou abstrato.

Uma URI tem um formato similar a este: “URI = scheme "://" authority "/" path ["?" query] por definição.

É muito importante que esta estrutura cumpra uma hierarquia que proporcione sentido próprio à URI. Deve ser de fácil entendimento, de modo que ao lê-la se obtenha a informação concreta sobre a qual recurso afeta.

Devem cumprir-se uma série de restrições:

1. Os nomes dos **recursos são indicados no plural**. Semanticamente é o correto porque definem coleções. Ex.: cards, users, customers, etc
2. Para **palavras compostas**, será utilizado o caractere dash ‘-’ como elemento de separação, conhecido como “lower-dash-case”. Ex.: user-account
3. As partes **variáveis** da URI devem ser **especificadas entre chaves “{ }” e em dash-case**. Isto corresponde aos identificadores concretos das coleções; além disso, esses identificadores devem ser prefixados. Ex.: /cards/v1/{card-id}
4. Os parâmetros de paginação, filtros e critérios de ordenação serão especificados como query params. O uso de cada um dos mesmos é detalhado em suas seções correspondentes.
5. Adicionalmente as URIs conterão todos os caracteres em minúsculas, embora a prioridade das URIs não sejam case-sensitive mas existe uma RFC, a 3986, que define URIs case-sensitive.
6. Não deve conter **verbos** na URI e os recursos devem ser sempre **substantivos**

Nas URIs teremos 2 tipos de parâmetros (ambos serão escritos como lower-dash-case):

- **path-parameters**
- **query-parameters**

Os path parameters são destinados exclusivamente para identificá-los para os recursos dentro das coleções mediante seu identificador único. Serão especificados como “{resource-name-id}” nos endpoints; isto será substituido pela ID do recurso em tempo de invocação.

Exemplo:

> Definição: /cards/v1/{card-id}
>
> Invocação: /cards/v1/1234

No SWAGGER, os path parameters são definidos como recurso:

Estrutura dos endpoints

A estrutura dos endpoints deve seguir o seguinte padrão:

Estrutura

```
{host}/{context?}/{domain?}/{service-version}/{resources}?{parameters?}
```

Onde:

Parâmetro	Descrição	Exemplo
host	é o domínio onde está API	api.viavarejo.com.br api.casasbahia.com.br api.extra.com.br api.pontofrio.com.br
context	Opcional. É o contexto de negócio ao qual o serviço irá atender. Este campo deve ser utilizado quando as APIs estão alocadas dentro de visões macros dentro da companhia. O valor padrão é vazio, indicando que os serviços contidos fazem parte do contexto da companhia como um todo.	marketplace viamais
domain	Opcional se um contexto estiver definido. Área de negócio a qual o serviço está atendendo. Em alguns casos, o recurso pode ser associado diretamente ao contexto.	catalog (catálogo) inventory (estoque) sellers (vendedores)
service-version	a versão do serviço que está sendo utilizado, levando em consideração sempre o campo MAJOR do versionamento.	v1
resources	Recursos do serviço	items
parameters	campos opcionais utlizados para definir critérios de busca (veja GET)	

Alguns exemplos de endpoint

		host	context	domain	service-version	resources	parameters
Endpoint com context o padrão e parâme tros	api.viavarejo.com.br/inventory/v1/items/?category-id=100	api.viavarejo.com.br		inventory	v1	/items	/?category-id=100
Endpoint com context o padrão sem parâme tros	api.viavarejo.com.br/catalog/v1/items	api.viavarejo.com.br		catalog	v1	/items	
Endpoint com context o padrão sem parâme tros	api.viavarejo.com.br/catalog/v1/items/1234	api.viavarejo.com.br		catalog	v1	/items/1234	

com indicação de id do resource							
Endpoint com contexto específico sem parâmetros	api.viavarejo.com.br/marketplace/catalog/v1/showcase	api.viavarejo.com.br	marketplace	catalog	v1	showcase	
Endpoint com contexto específico, porém sem domínio. O recurso é valido dentro do contexto.	api.viavarejo.com.br/marketplace/v1/sellers	api.viavarejo.com.br	marketplace		v1	sellers	

HTTP Headers

Content Negotiation

- Mecanismos que permitem diferentes versões do mesmo documento a existir na mesma URL, assim o client-serve conseguem determinar qual a melhor versão que melhor se encaixa em suas capacidades.

Caching

- Um modo de armazenar e obter dados, assim os próximos requests serão mais rápidos sem precisar realizar a operação novamente (calcular, request para outra API, etc).

ETags

- É uma abordagem para não trafegar todo o payload a cada request
    1. Client faz um request
    2. Server responde e cria um ETag baseado no estado do recurso
    3. Client faz outro request com o verbo HEAD (mesmo request anterior, apenas com verbo HEAD)
    4. Se o dado não mudou, o server envia o mesmo ETag
    5. Se o dado mudou, o server envia outro ETag
    6. Se o ETag mudou, o client faz outro request novamente com o verbo correto
- Existe um pequeno delay devido ao request extra, mas para aplicações mobile com conexão ruim, é uma ótima solução pois não precisa trafegar payloads se a resposta é sempre a mesma

Contratos

Essa documentação tem por objetivo propor uma solução padronizada para criação de contratos de API.

Definição de Endpoint (Path e Query Parameters)

Estrutura dos endpoints para seguir.

O que é URI?

Segundo a documentação da Via:
http://confluence.viavarejo.com.br/pages/viewpage.action?pageId=60926674

Exemplo de paths:

/v1/antifraude/transacoes

/v1/antifraude/transacoes/1

/v1/antifraude/transacoes/1/regras/1234

/v1/antifraude/perfilRegras

Definição do Body

As respostas HTTP seguem dois padrões, um para cenários de sucessos, outros para cenários de erros.

Para cenário de sucesso.

Significado das propriedades:

meta: propriedade obrigatória, objeto responsável por conter todas as informações referente a meta dados da requisição;

version: propriedade obrigatória, representa a versão completa da API;

limit: propriedade opcional, representa o limit quando o mesmo é passado como parâmetro;

offset: propriedade opcional, representa o offset quando o mesmo é passado como parâmetro;

total: propriedade obrigatória, representa o total de registros para o recurso;

records: propriedade obrigatória, representa os registros retornados em um tipo array;

Para cenário de erro.

Significado das propriedades (todas são obrigatórias):

errors: conjunto de elementos no qual possui os erros de uma requisição;

code: código de erro de negócio;

message: objeto responsável por conter as mensagens de erro de todos os níveis;

system: mensagem de erro de nível de sistema;

user: mensagem de erro de nível de usuário (cliente);

Catálogo de Erros HTTP

O catálogo de Erros HTTP de APIs é baseado nos status codes e refere-se a padronização as falhas que podem ocorrer nas APIs, seja no lado do cliente, seja no lado do servidor. As excessões devem ser devolvidas ao cliente com as seguintes informações:

Status Code	Código	Mensagem de Sistema	Mensagem de Usuário
400	0001	Bad Request	${field} deve ser enviado
400	0002	Bad Request	${field} deve ser do tipo inteiro
400	0003	Bad Request	${field} deve ser do tipo texto
400	0004	Bad Request	${field} deve ser do tipo decimal
400	0005	Bad Request	${field} deve ser do tipo data
400	0006	Bad Request	${field} deve ser do tipo timestamp
401	0007	Unauthorized	Você não está autorizado
402	0008	Payment Required	Seu pagamento infelizmente foi declinado
403	0009	Forbidden	Você não está autorizado a acessar esse recurso
404	0010	Not Found	Seu recurso não foi encontrado