Estudando API

Status	Em progresso

Definindo contratos de REST APIs

Introdução

APIs sejam no padrão REST - que a gente vai explorar melhor neste texto - ou outros quaisquer como os variantes de RPC ou GraphQL, já se provaram faz tempo como boas soluções para comunicação entre sistemas.

É praticamente obrigatória em sistemas modernos e muito útil em sistemas legados, pois ajudam a criar uma camada de desacoplamento e abstração da complexidade do que é considerado legado, dando uma "cara" mais aderente com os padrões mais modernos de software.

Nas APIs são definidos contratos (com campos de entrada e saída) que são respeitados pela parte que consome e pela parte que fornece a informação e mesmo que haja alterações internas em qualquer um dos lados, as partes respeitam este contrato não obrigando a outra parte a fazer ajustes no software.

O contrato é algo evolutivo e pode agregar novas funcionalidades com o passar do tempo, principalmente se adotadas as melhores práticas na sua definição e versionamento.

E justamente na definição do contrato que observo que a capacidade de "análise do sistema" parece ter sido colocada em segundo plano por muitos desenvolvedores. Talvez por conta de abrirmos mão da documentação conforme manifesto ágil e não adotarmos as outras práticas de entendimento do problema conforme o próprio ágil define; talvez, porque parar para pensar no sistema sem produzir nenhuma linha de código pode parecer improdutivo para muitos de nós. No entanto, negligenciar esta etapa de análise, traz atrasos, refatorações e posterga alguns tipos de problemas para o momento em que poderíamos estar apenas transformando café em código.

Quando fazemos APIs pensando primeiro no contrato (contract first), trazemos um nível de maturidade muito maior de entendimento da solução como um todo, antes mesmo de começar a programar. Além de iniciar o desenvolvimento com um artefato (contrato) que pode agilizar geração de código fonte, mocks, documentação etc.

O propósito deste guia é trazer em caráter introdutório como é o processo de entender o negócio e montar um contrato de REST API.

> Após a leitura deste guia, recomendo a leitura do Guia de Design REST que aprofunda bem em alguns conceitos e serve como material de referência.

APIs

API significa **A**pplication **P**rogramming **I**nterface, ou seja, é um interface para que um programa possa se comunicar com outro programa diferente. APIs podem ser bibliotecas (.dll), trechos de código, endereços na internet, frameworks etc. O importante é que ela permite que um sistema acesse as suas funções de outro sistema sem conhecer seus detalhes internos.

Hoje, quando nos referimos às APIs, já subentendemos que a comunicação entre os sistemas se dá através de sistemas conectados em rede - ou internet - e, na maioria dos casos, se comunicando através do protocolo HTTP (protocolo da internet).

Além de expor as informações do sistema através de um contrato, que é o foco deste texto, APIs usadas para conectar sistemas em rede costumam trazer consigo padrões arquiteturais que trazem capacidades importantes para tratamento da comunicação. Por exemplo:

- API Gateways que fazem o roteamento das mensagens para diversos tipos de sistemas, garantem a aplicação de políticas de segurança, controlam o volume de acessos etc.
- APIs que podem ser programadas para orquestrar, compor e transformar informações do sistema afim de abstrair algumas complexidades e facilitar o uso pelo sistema cliente.
- Sistemas de cache acoplados na API ou no API Gateway podem armazenar respostas para requisições já feitas anteriormente em ambientes de alta performance e baixo custo em relação ao gasto de ter de fazer todo o processamento novamente.
- Padronização de protocolos que fazem com que o consumidor não se preocupe com detalhes da linguagem e ambiente de execução do sistema que processa as informações. Permitindo até que o sistema que processa a informação possa ter partes do seu ambiente em uma linguagem e partes em outras.

Hoje, é praticamente unânime o uso do HTTP, dado que é um protocolo amplamente conhecido em ambientes distribuídos (web), mas mesmo dentro dele, algumas tecnologias que já tiveram

seu auge e caíram em desuso:

- RPC (**R**emote **P**rocedure **C**all), em que não haviam regras rígidas com relação ao formato da mensagem trafegada e apenas se trafegava qualquer texto via HTTP.
- SOAP (**S**imple **O**bject **A**ccess **P**rotocol), comumente chamado de WebService, endereçou bem questões como segurança e formato de mensagem. Entregou um nível maior de padronizações e permitiu criação de frameworks que facilitavam bastante para o desenvolvedor em utilizar esse padrão. Trafegava XML.

Então, populariza-se o REST com a chegada dos dispositivos móveis com baixo poder de processamento e pouca de banda de internet, trazendo junto o JSON como formato da mensagem, que é um padrão muito mais enxuto. O REST, em relação ao SOAP que usava XML e muitas estruturas para definir a mensagem, foi mais econômico no que diz respeito ao consumo de banda, processador e memória.

REST API

REST API (**Re**presentational **S**tate **T**ransfer Application Programming Interface) é um estilo de arquitetura que define um conjunto de restrições e propriedades baseadas em HTTP fornecendo interoperabilidade entre sistemas de computadores na internet (ou rede local). REST permite que os sistemas consumidores acessem e manipulem representações textuais de recursos - as "coisas" da vida real em REST se chamam recursos - usando um conjunto pré definido de operações.

Por colocar mais restrições do que WebServices SOAP e utilizar mais recursos já nativos do protocolo HTTP, acabou padronizando bastante a forma de comunicação e isso agilizou o desenvolvimento das integrações.

O REST enfatiza::

- **Simplicidade** através de uma comunicação bem definida, com algumas restrições que facilitam a padronização e stateless, ou seja, todas as informações necessárias para o processamento de uma requisição devem estar contidas naquela requisição.
- **Extensibilidade** dado que um determinado recurso pode ser representado em diferentes formatos e até mesmo versões, além da possibilidade de se adicionar novos recursos sem precisar alterar os já existentes.
- **Confiabilidade** porque existe uma definição clara das ações que podem ser feitas e dos seus efeitos colaterais.
- **Performance** porque faz parte dos principais pilares do REST o uso de cache em todas as camadas possíveis através de uma semântica bem definida, além de uma arquitetura baseada em separação de camadas, permitindo que partes diferentes do sistema possam ser escaladas de forma independente e isolada.

A maioria dos softwares nos quais nós interagimos são construídos para atender às necessidades humanas e normalmente referenciam as "coisas" reais das quais esses softwares tratam. Por exemplo, um software de biblioteca vai representar livros, seções, autores etc. e, através de alguma interface, permitir que um usuário interaja com estas representações. Uma API difere-se deste tipo de software no que tange o usuário: quem interage com o software é outro software, não diretamente o usuário final. No entanto, quem desenvolve o software que interage

com a API é um programador e para que a experiência deste programador e do usuário do software que ele desenvolve seja a melhor possível, princípios de design devem ser respeitados.

O assunto que será abordado no próximo capítulo é sobre **como definir os contratos das APIs**. Não entrarei em conceitos avançados de REST, arquitetura REST (camadas), implementação do código fonte, autenticação, entre outros assuntos que permeiam a integração via REST pelo simples fato de que eles já estão muito bem documentados na web. Alguns conceitos serão brevemente explicados para permitir que o processo de definição do contrato seja entendido e conceitos específicos de design pode ser aprofundados neste material: **Design de REST API**.

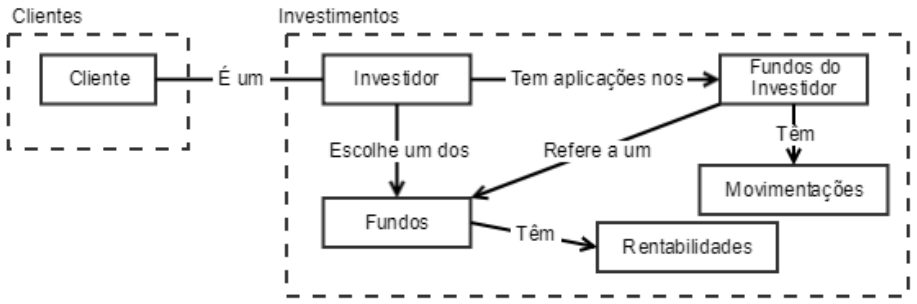
Entendendo e representando o negócio

O primeiro grande desafio ao se construir qualquer software deveria ser entender o negócio sobre o qual o software trata. Muitas vezes, essa etapa de entendimento é ignorada e o software produzido acaba não refletindo o negócio como ele é. Quando isso acontece, as alterações que aos olhos de negócio pareceriam simples, quando são orçadas pela TI, descobre-se que são mais complexas e caras do que o imaginado.

Para ajudar a entender o negócio e representá-lo de uma forma que possa ser entendido desde o analista de negócio até o desenvolvedor, pode ser feito um Domain Model. O Domain Model é um dos muitos conceitos do Domain Driven Design (DDD).

O DDD fornece um conjunto de recursos e padrões que ajudam a documentar e modelar software. O DDD ganhou notoriedade recentemente por facilitar o recorte de Microsserviços.

O Domain Model é uma representação de conceitos do mundo real (do negócio) que são pertinentes ao um ou mais domínios a serem trabalhados em um software. Estes conceitos incluem as entidades e as relações entre elas. No entanto, não faz parte de um Domain Model implementações técnicas como banco de dados ou componentes específicos de software.



Exemplo de um Domain Model suficiente para começar a definir APIs

O Domain Model quando pronto, mesmo com um modelo simplificado como o do exemplo acima, parece simples e deve ser simples, no entanto, a riqueza de se modelar o domínio está mais no processo da criação dele do que no resultado final. Durante o processo de criação do Domain Model, deve-se alinhar o entendimento do time sobre os melhores nomes para representar as entidades de negócio e as relações entre elas. Neste processo muitas divergências ocorrem entre o que é o negócio como a área de negócio entende, o que é o sistema no estado atual e o que algumas pessoas têm em mente como visão futura daquele ambiente. Muitas vezes o analista com mais profundidade nos conceitos do Domain Driven Design precisa orientar o time sobre os conceitos e objetivo do Domain Model.

Alguns conceitos úteis podem guiar o processo da criação do Domain Model:

- **Bounded Context**: São fronteiras (no exemplo acima representadas por tracejado) que separam conceitos que têm alguma relação entre si. Não existem regras rígidas para criação de bounded context, mas o significado desta fronteira deve ser claro para todo o time. Um bounded context pode inspirar divisões para softwares diferentes, contratos de APIs diferentes, componentes diferentes, etc.
- **Ubiquitous Language**: Linguagem ubíqua, em português, é o uso da linguagem falada que seja de conhecimento comum entre todos os envolvidos com um conceito de negócio: o jargão. Um conceito de negócio deve ter um nome único e seu significado deve ser claro para todos. Devemos buscar utilizar estes nomes adotados em todo o software, desde o linguajar usado no dia a dia à telas, programação, bancos de dados, etc. Um dos pontos fortes do DDD é o resgate e reforço do uso de uma notação ubíqua entre os domain experts (quem conhece do negócio) e área de TI e o outro é a busca pela simplificação. Alguém sem conhecimento de um determinado negócio, deveria ser capaz de aprender sobre ele apenas lendo o código do sistema. E alguém com conhecimento de negócio deveria ter facilidade em usar e até mesmo entender a estrutura de dados e funções codificadas apenas reconhecendo os termos lá presentes. Um software que não tem suas raízes plantadas profundamente no domínio, não irá se adaptar bem às mudanças com o passar do tempo.

O Domain Driven Design tem muitos conceitos: é como uma grande caixa de ferramentas. Neste guia para definição de contratos de APIs, somente alguns deles já serão suficientes para atingir o objetivo.

Vamos à prática - Iniciando o Domain Model

Primeiramente, é necessário identificar as **entidades** do negócio. Entidades são os nomes genéricos que damos às coisas, como "carro", "livro", "cartão", "conta", etc.

Uma entidade é uma representação de um conjunto de informações sobre determinado conceito do sistema. Estas informações são os **atributos**, que juntos representam a entidade.

Entidades têm **identificadores** e através dos identificadores, distinguimos duas instâncias diferentes da mesma entidade. Por exemplo, se existir uma entidade carro, como se distingue dois Fuscas azuis fabricados no mesmo dia? Neste caso, pode-se usar o número do chassi como identificador da entidade.

Enquanto um analista de sistemas ou de negócio está explicando a ideia do software, já podemos identificar no falar da pessoa algumas possíveis entidades. Durante o falar, já é interessante ir anotando o que for identificado.

Como identificar entidades

O que "tem cara" de ser entidade:

- Recursos (coisas) que têm um identificador. Por exemplo: conta corrente tem um número que a representa, cartão de crédito tem um número, carro tem chassi, aparelho celular tem imei, placa de rede tem MAC adress, pessoas têm documentos, etc.
- Palavras que vêm depois de frases como: "- Aí eu gravo a .", "- Então eu trago uma lista de .", "O usuário escolhe uma .".

- Definições de negócio que não têm um identificador, mas você consegue identificar unitariamente entre todos. Por exemplo: "A fatura do mês 08 de 2017."
- **Substantivos**.

O que não "tem cara" de ser entidade:

- Definições de negócio que não têm ID, nem é possível identificá-las unitariamente entre todos.
- Definições que têm como nome "gravação", "listagem", "detalhes", "dados", "informações" entre outros termos vagos. Esse tipo de nomenclatura provavelmente precisa ser refinada e talvez desse refinamento, surja uma entidade.
- Verbos.

Tendo entendido este conceito de entidade, podemos fazer um primeiro desenho sobre o negócio de cartões representando as principais entidades. Para esse exemplo, usei um caso semelhante ao real em ambiente bancário. As soluções deste guia são para fins didáticos, então vamos chegar em um resultado próximo ao real, mas simplificado.

Domain Model Simplificado de Cartões.

Antes do processo de desenhar as entidades, durante ou depois - a ordem não importa - é preciso entender também quais são as funções de negócio que serão disponibilizadas. Para usar como exemplo para este guia, vamos inventar algumas necessidades de negócio que são semelhantes às reais:

- listar os cartões em que é possível fazer grade (upgrade ou downgrade);
- ofertar possibilidades de grade de cartão;
- listar os cartões cuja fatura pode ser consultada online;
- listar as faturas de um cartão;
- listar os cartões que estão com a opção de fatura digital ativada;
- listar os cartões em que permitem parcelar a fatura em aberto;
- ofertar possibilidades de parcelamento de faturas;
- contratar parcelamento de faturas;

Enquanto para identificar as entidades focamos nos substantivos, da lista de funções precisamos focar nos verbos:

- listar

- ofertar
- contratar

Veremos mais detalhes sobre REST nos capítulos posteriores, mas aqui já precisamos ter em mente que o REST tem um conjunto de verbos limitados para serem usados. São eles: GET, POST, PUT, PATCH e DELETE. Este conjunto básico de verbos do REST representam operações de CRUD (Create, Read, Update, Delete). Podemos entender que o listar é na verdade uma consulta, logo, será substituído pelo verbo GET que representa uma consulta.

Apesar de muitas situações não requererem grande esforço criativo por apenas exporem as suas entidades e operações de CRUD, muitas APIs vão além disso e precisam refletir funções de negócio como reservar, comprar, simular, transferir, calcular, ofertar, cancelar, contratar, etc. Na lista de verbos acima existem alguns casos nesta situação: ofertar e contratar. Encaixar estes verbos (funções) dentro das restrições do REST é o segundo grande desafio da modelagem de APIs.

Mais pra frente neste guia vamos detalhar esta transformação dos verbos do negócio em verbos do REST, por agora, precisamos apenas identificar que eles estão associados com “coisas” que normalmente têm um ID. Estas funções de negócio não são entidades porque não têm IDs, mas podem ser representados também no Domain Model, principalmente para entender a relação deles com as entidades. Por vezes, essas funções também envolverão várias outras entidades para executar a função. Chamamos estas funções de serviços e podemos colocar também no modelo.

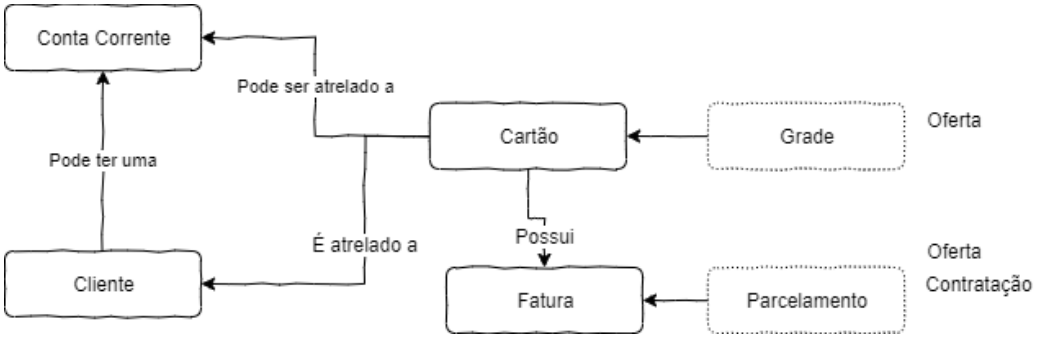
Como identificar serviços

O que "tem cara" de ser serviço:

- Recursos que cuja resposta varia conforme um conjunto de filtros e não são armazenados no banco de dados, como algumas simulações e cálculos.
- Recursos que retornam resultados que não têm ID, como extratos, francesinha, saldo, posição consolidada, etc.
- **Verbos**.

O que "não tem cara" de ser serviço:

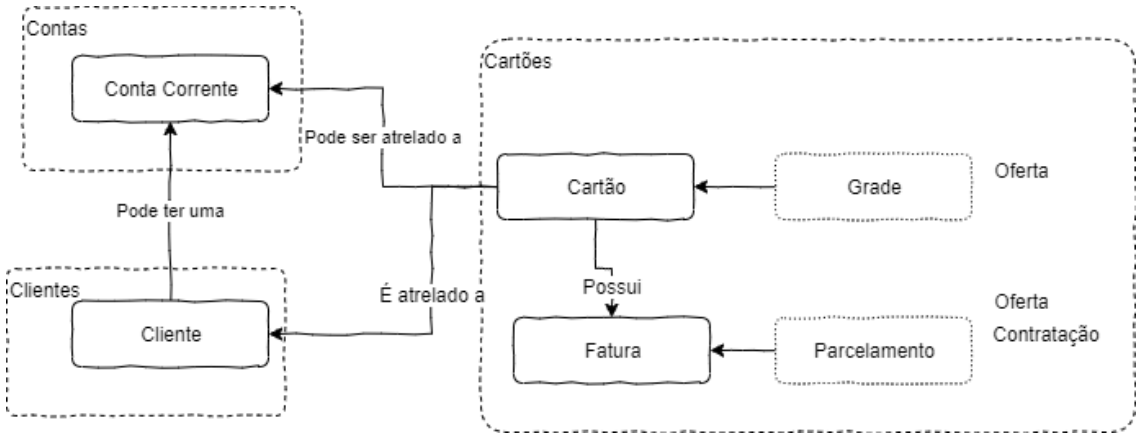
- Verbos que representam CRUD, como listar, gravar, apagar etc.
- Recursos que têm ID e são armazenados em banco de dados.
- Substantivos.



Domain Model de Cartões com alguns serviços.

Agora, precisamos pensar em dividir os assuntos que estamos trabalhando. Mesmo o modelo não estando 100% refinado, já podemos recortar os assuntos semelhantes através de bounded contexts. O Domain Driven Design não define rigidamente os critérios para executar estes recortes, no entanto, uma coisa é regra: o recorte deve ser consensual entre os analistas do negócio. Não sendo, deve-se buscar o alinhamento.

No modelo abaixo, separamos os assuntos de cartões de outros assuntos que fazem parte do banco, mas não diretamente do negócio de cartões. Por vezes, este recorte será relacionado aos domínios, subdomínios e/ou grupos funcionais, outras vezes será um produto específico.



Domain Model de Cartões com bounded contexts.

Até o ponto que chegamos podemos observar que já temos um diagrama que facilita a explicação do negócio para qualquer pessoa dentro ou fora do contexto desse negócio. O principal ponto é que até para que diagramas simples como este muita discussão acontece e com ela alinhamentos e compartilhamento de conhecimento.

Para a apresentação do Domain Model não há um formato rígido: já trabalhei com times que fizeram em ferramentas de UML, em apresentação de slides, em folha de caderno e a grande maioria foram rabiscados em mesas de vidro e registrados com foto no celular. Isso já foi suficiente para geração de algumas centenas de boas modelagens de contratos de API durante minha carreira.

Por ora, paramos por aqui com Domain Driven Design para absorver outros conceitos novos.

Conceitos básicos de REST

Recursos

Recursos no REST são os "substantivos" do seu sistema. São as entidades que serão expostas para outros sistemas. Os recursos são representados através de URLs (Uniform Resource Location) como a dos sites na web.

Exemplos de recursos representados por URLs:

- Representação das artistas do Spotify: **https://api.spotify.com/v1/artists/**
- Representação dos álbuns do Spotify: **https://api.spotify.com/v1/albums**
- Representação de um álbum específico do Spotify: **https://api.spotify.com/v1/albums/{id}**

- Representação das trilhas de um álbum do Spotify: **https://api.spotify.com/v1/albums/{id}/tracks**

Estes substantivos representados pelas URLs representam o "S" de "State" do RE**S**T. Isso significa que existe uma representação do estado de um recurso. Estado é como um recurso se encontra em um momento: é a situação atual dos dados de uma entidade em um determinado instante.

Ações

Clientes de REST APIs alteram o estado dos recursos, ou seja, alteram suas propriedades através de 5 ações. Existem outras, mas essas são as principais:

- Consultar, que utiliza o verbo GET do HTTP;
- Criar Novo, que utiliza o verbo POST do HTTP;
- Substituir, que utiliza o verbo PUT do HTTP;
- Atualizar parcialmente, que utiliza o verbo PATCH do HTTP;
- Apagar, que utiliza o verbo DELETE do HTTP.

Estas ações também são chamadas de verbos ou métodos. Elas representam o "T" de "transfer" do RES**T**. Isso significa que você está transferindo o recurso de um estado para outro.

Outro conceito importante no que tange os verbos é sobre ser seguro e/ou idempotente.

Consumidores de REST APIs quando utilizam verbos seguros, entendem que eles não alteram o estado dos recursos quando fazem uma requisição. Ou seja, podem fazer uma tentativa de requisição e, se falhar, simplesmente tentar de novo, pois sabem que não estão fazendo alterações no sistema.

Consumidores de REST APIs quando utilizam verbos idempotentes, entendem que eles podem executar várias vezes a requisição e terão o mesmo resultado inicial.

Por exemplo:

- GET é seguro, ao usá-lo, é feita uma consulta no backend da e nada é alterado.
- GET é idempotente, posso chamá-lo várias vezes e terei o mesmo recurso como retorno.
- POST não é seguro, pois quando é feita uma requisição com POST, ele cria coisas novas no backend.
- POST não é idempotente, pois a cada vez que eu faço uma requisição com POST, ele cria um recurso novo.
- PUT não é seguro, pois quando é feita uma requisição com PUT, ele altera do estado do recurso no backend.
- PUT é idempotente, pois a cada vez que eu faço uma requisição com PUT, ele altera da mesma forma o recurso.

O mais importante sobre segurança e idempotência é que na modelagem de contratos de APIs sejam respeitados os propósitos dos verbos, porque por se tratar de um padrão, os consumidores têm uma expectativa de funcionamento destes verbos. Logo, se você cria uma API em que um GET altera estado de recursos, um consumidor, poderá estar alterando o estado deste recurso de forma não intencional.

Faz parte também do protocolo HTTP o uso de códigos de retorno para entender quando uma requisição foi bem ou mal sucedida. Os códigos são compostos de 3 dígitos e de forma genérica:

- 200 significa que a requisição foi bem sucedida
- 300 significa que a requisição foi redirecionada para outra URL
- 400 significa que o usuário cometeu um erro
- 500 significa que ocorreu um erro no servidor

E cada um deles pode se desdobrar em outros mais específicos, por exemplo:

- 201 (sucesso) significa que um recurso foi criado
- 206 (sucesso) significa que a resposta está paginada
- 403 (erro do requisitante) significa que o usuário não tem acesso naquele recurso
- 404 (erro do requisitante) significa que a URL (recurso) não existe
- 503 (erro do servidor) significa que o servidor não está disponível

Então, é parte do processo executar uma ação em uma API através de um dos verbos e receber um código de retorno indicando o status daquela requisição. Como os códigos são padrões, a aplicação poderá implementar comportamentos padrões conforme os códigos de retorno.

Representações

Representações é a forma como os clientes da API vêem os recursos. Os clientes de APIs enxergam estas representações do recurso.

Por exemplo, uma pessoa para um sistema está armazenada em uma tabela no banco de dados, mas a representação deste recurso para o cliente se dá através de um JSON com alguns campos, talvez nem todos os campos da tabela e talvez com alguns filtros, mas o acesso direto à tabela (recurso), ele não tem. Nesta mesma linha, a imagem da pessoa pode estar em um file server dentro da empresa, e através da API esta imagem é exposta como JPG. O usuário tem acesso à representação da imagem original, talvez com resolução menor, talvez com restrições de acesso e não tem direto acesso ao recurso original no file server.

Estas representações são o "RE" de "representational" do **RE**ST.

O funcionamento de uma chamada em REST

Suponha a existência de uma API que informe o clima. Ela estaria exposta em uma URL semelhante a essa **http://conhecasaopaulo.com/temperatura/** e o servidor responde a temperatura através de uma representação em JSON.

Em HTTP a comunicação se dá através de um Request com a requisição de alguma informação e um Response com a resposta da requisição. Por se tratar de uma consulta, utiliza-se um dos verbos do HTTP, o GET. Como a requisição foi bem sucedida, foi retornado o HTTP status code 200.

Ex:

```
Request
GET http://conhecasaopaulo.com/temperatura/
```

```
Response
HTTP 200 Ok
{"celsius": 14, "fahrenheit": 57}
```

Para envio desta requisição e recepção da resposta em HTTP, quase todas as linguagens de programação têm funções que permitem fazê-la.

Para transformar a representação em JSON em um objeto a ser manipulado, as linguagens de programação também têm funções que fazem esta conversão de forma automática.

Filtrando Resultados

Imagine a existência de um recurso que represente os restaurantes de São Paulo. Uma busca por eles seria algo como:

```
Request
GET http://conhecasaopaulo.com/restaurantes

Response
HTTP 200 Ok
{
   [
      {
         "id":1,
         "nome":"Restaurante A",
         "bairro":"Pinheiros",
         "endereço":"Rua XPTO, 123",
         "telefone":"11 912345678"
      },
      {
         "id":2,
         "nome":"Restaurante B",
         "bairro":"Mooca",
         "endereco":"Rua ABCD, 234",
         "telefone":"11 345678922"
      },
      ...
   ]
}
```

No entanto, esta lista pode ser realmente grande e tornar lenta uma possível busca onde queremos apenas alguns restaurantes conforme algum critério, não todos.

Para fazer filtragem usamos query parameters. Query parameters fazem parte da URL do HTTP e são representados por conjuntos chave/valor iniciados com interrogação e separados por "e" comercial.

Ex: http://.../...?chave1=valor1&chave2=valor2&chave3=valor3

Logo, caso você queira filtrar a lista de restaurantes por bairro, por exemplo, utilizamos os query parameters para fazer o filtro no recurso.

```
Request
GET http://conhecasaopaulo.com/restaurantes?bairro=Pinheiros

Response
HTTP 200 Ok
{
  [
      {
         "id":1,
         "nome":"Restaurante A",
         "bairro":"Pinheiros",
         "endereço":"Rua XPTO, 123",
         "telefone":"11 912345678"
      }
      ...
   ]
}
```

Transformando o modelo em recursos e verbos de REST APIs

Tendo aprendido algumas das características do REST, já podemos começar a transformar nossa especificação de negócio em um contrato de API.

E para cada uma dos serviços ou entidades identificados no Domain Model, devemos fazer os seguintes questionamentos:

1 - Qual é o mapeamento entre os verbos HTTP e os verbos das funcionalidades do meu negócio?

Os “listar” são mais óbvios e representam consultas, logo, para consultas no REST usamos o verbo GET.

Pensando na chamada REST, o servidor está parado e recebe uma requisição, logo, o “ofertar” não é uma ação iniciada pelo servidor, mas pelo cliente da API que “buscará” uma oferta de algo. Assim, essa busca é na verdade uma consulta a um conjunto de informações, logo, usaremos também o verbo GET.

Para o “contratar”, não temos uma ação segura, pois eu crio um estado novo no lado do servidor através da gravação da contratação, neste caso, usaremos o verbo POST.

Fazendo a devida substituição:

- listar: GET dos cartões em que é possível fazer grade (upgrade ou downgrade);
- ofertar: GET de possibilidades de grade de cartão;
- listar: GET das faturas de um cartão;
- listar: GET dos cartões que estão com a opção de fatura digital ativada;
- listar: GET dos cartões em que permitem parcelar a fatura em aberto;

- ofertar: GET de possibilidades de parcelamento de faturas;
- contratar: POST parcelamento de faturas;

2 - Que recursos em formato de URL serão criados para cada entidade ou serviço do meu negócio?

Analisando o Domain Model, transformamos as funcionalidades e os verbos definidos na questão anterior em recursos na URL:

- GET dos cartões http://.../cartoes em que é possível fazer grade (upgrade ou downgrade)

A entidade a ser retornada são os cartões.

- GET de possibilidades de grade de cartão http://.../cartoes//ofertas-upgrade

O recurso a ser retornado são ofertas de upgrade de cartão. Essas ofertas de cartão são feitas sempre à partir de um cartão existente (). Por isso, ela é relacionada a um cartão e aparece na URL como um sub-recurso.

- GET das faturas de um cartão http://.../cartoes//faturas/

O recurso a ser retornado são as faturas de um cartão específico () e existe uma fatura por mês. Cada uma dessas faturas terá o seu ID.

- GET dos cartões http://.../cartoes que estão com a opção de fatura digital ativada

O recurso a ser retornado são os cartões que têm uma característica de disponibilizar a fatura apenas online.

- GET dos cartões http://.../cartoes em que permitem parcelar a fatura em aberto

O recurso a ser retornado são os cartões que têm uma característica de permitir o parcelamento da fatura.

- GET de possibilidades de parcelamento de faturas http://.../cartoes//ofertas-parcelamentos-fatura

O recurso a ser retornado é um serviço que faz ofertas de parcelamentos de faturas pré calculadas. As ofertas são para um cartão específico por isso, o recurso é relacionado com um cartão já existente ().

- POST parcelamento de faturas http://.../cartoes//faturas//parcelamento/

O recurso é um serviço que grava uma escolha de parcelamento. O parcelamento é de uma fatura específica, por isso, o recurso é relacionado com uma fatura já existente ().

3 - Para cada verbo + URL, quais são os parâmetros de entrada?

- GET http://.../cartoes?upgrade=true

A entidade a ser retornada são os cartões que tenham uma característica de permitir upgrade. No caso, dentre todos os existentes, faremos um filtro através do query parameter upgrade=true.

- GET http://.../cartoes//ofertas-upgrade

O recurso a ser retornado são ofertas de cartões. Oferta de cartão não é a entidade de cartão que representa os cartões já existentes de um cliente e as ofertas são feitas sempre à partir de um cartão existente (). Por isso, ela é relacionada a um cartão e aparece na URL como um sub-recurso.

- GET http://.../cartoes//faturas/

O recurso a ser retornado são as faturas de um cartão específico () e existe uma fatura por mês. Cada uma dessas faturas terá o seu ID.

- GET http://.../cartoes?elegivelFaturaDigital=true

O recurso a ser retornado são os cartões que têm uma característica de disponibilizar a fatura apenas online.

- GET http://.../cartoes?elegivelParcelamentoFatura=true

O recurso a ser retornado são os cartões que têm uma característica de permitir o parcelamento da fatura.

- GET http://.../cartoes//ofertasParcelamentosFatura

O recurso a ser retornado é um serviço que faz ofertas de parcelamentos de faturas pré calculadas. As ofertas são para um cartão específico, por isso o recurso é relacionado com um cartão já existente ().

- POST http://.../cartoes//faturas//parcelamento

Por se tratar de um POST, os parâmetros vão no body da mensagem, ao contrário das consultas que são via queryparameter. Assim, precisa identificar quais são esses parâmetros que o consumidor da API envia. Estes parâmetros devem ser relacionados apenas ao serviço de parcelamento e, independente de como é o backend, devem ser escritos de forma a ser o mais auto-explicativo possível, além de ser apenas os campos necessários para o consumidor. Ou seja, nesta camada de API, devemos abstrair a complexidade do backend afim de dar a melhor experiência possível para quem consome e não conhece no detalhe o sistema.

```
{
   "idConta": "0004321098765432109",
   "valorPagamamentoMinimo": 123456.78,
   "idOferta": “12P1”,
   "quantidadeParcelas": 12,
   "valorParcela": "234.56",
   "dataPrimeiroVencimento": "2018-06-20",
   "dataPagamentoEntrada": "2018-05-20",
   "valorEntrada": 1000.20,
   "valorUltimoPagamento": 134.67,
   "seguroPrestamista": false
}
```

3.1 - Pergunta especial: E o ID do usuário?

Talvez você tenha percebido que não é passada identificação do usuário em nenhuma das APIs acima. Isso é porque a identificação do usuário deve vir via token. O uso de token é explicado nas tecnologias **OAUTH**, **Open Id Connect** e **JWT**.

Basicamente, todas as chamadas de API passam no header um texto que pode ser uma chave de busca para recuperar as informações do usuário em um servidor de autenticação, ou um texto em formato base64 que pode ser convertido para JSON e neste JSON está a identificação do cliente e aplicação que chamou a API. Este texto chama-se token.

Então, podemos considerar as informações do usuário logado (cliente) também é um parâmetro de entrada, mas não precisamos definí-lo na API, pois ele é parte do token. Sendo parte do token, quem recebe a informação tem meios de garantir que ela não foi adulterada e foi gerada por um serviço confiável.

> Para conhecer melhor sobre esse assunto, confira estes materiais:https://www.youtube.com/watch?v=h4A8HytL5tshttps://oauth.net/2/https://jwt.io/https://openid.net/connect/

4 - Para cada verbo + URL, quais são os parâmetros de saída?

Neste ponto, a gente pensa em cada entidade a ser exposta por API envolvida no nosso sistema e define o conjunto de atributos que a representa. Vamos representá-la em JSON, pois é o padrão que será utilizado na hora de programar a API. O exercício de definir estes campos já servirá como base para usar de exemplo na documentação (contrato).

```
Request
GET http://.../cartoes

Response
{
   "id": "44723873284",
   "numero": "879879879898798798",
   "nome": "Cartão XPTO",
   "nomePortador": "José da Silva",
   "dataValidade": "12/25",
   "bandeira": "Mastercard",
   "parceiro": "Ipiranga",
   "cvv": "789",
   "dataMelhorCompra": "2016-02-28",
   "bloqueado": false
}
```

```
Request
GET http://.../cartoes/{idCartao}/ofertas-upgrade

Response
{
    "id": "90909090909",
    "nomeProduto": "CARTAO TOP MULTIPLO MATERCARD PLATINUM",
    "valorAnuidade": 480,
    "quantidadeParcelasAnuidade": 12,
    "valorParcelaAnuidade": 40,
    "beneficios": [
        {
            "id": "1",
            "tituloBeneficio": "20% descontos em diversos cinemas, teatro
```

```
s e shows",
            "descricaoBeneficio": "Curta diversos shows, teatros e cinema
s com 20% de desconto!"
        },
        {
            "id": "2",
            "tituloBeneficio": "Compras parceladas",
            "descricaoBeneficio": "Parcele o pagamento sem pagar juros ou
encargos até o máximo de vezes que o estabelecimento permitir."
        }
    ]
}
```

```
Request
GET http://.../cartoes/{idCartao}/faturas

Response
{
    "id": "123123123",
    "status": "atual",
    "totalFatura": 1000.12,
    "dataVencimento": "2017-01-29",
    "dataFechamento": "2017-01-27",
    "pagamentoMinimo": 100.00,
    "taxaJurosFinanciamentoAm": 1.2,
    "taxaJurosFinanciamentoAa": 1.3,
    "elegivelFaturaDigital": true,
    "elegivelPagamentoFatura": true,
    "lançamentos": [
        {
            "data ": "2017-10-19",
            "descricaoLancamento": "SHOPPING ABC",
            "valorLancamento": 72.8
        },
        {
            "data": "2017-10-23",
            "descricaoLancamento": "POSTO AGUIA",
            "valorLancamento": 128.70
        }
    ]
}
```

```
Request
GET http://.../cartoes/{idCartao}/ofertasParcelamentosFatura

Response
```

```
{
    "id": "12P1",
    "quantidadeParcelas": 12,
    "valorParcela": 230.10,
    "taxaJuros": 1.54
}
```

```
Request
POST http://.../cartoes/{idCartao}/faturas/{idFatura}/parcelamento

Response
{
    "idConta": "00043210987654321O9",
    "valorPagamamentoMinimo": 123456.78,
    "idOferta": “12P1”,
    "quantidadeParcela": 12,
    "valorParcela": "234.56",
    "dataPrimeiroVencimento": "2018-06-20",
    "dataPagamentoEntrada": "2018-05-20",
    "valorEntrada": 1000.20,
    "valorUltimoPagamento": 134.67,
    "seguroPrestamista": false,
    "dataHoraContratacao": "2018-05-15T21:00:00"
}
```

5 - Para cada verbo + URL, quais são os códigos de retorno HTTP a serem utilizados?

- GET http://.../cartoes?upgrade=true
- GET http://.../cartoes//ofertas-upgrade
- GET http://.../cartoes//faturas/
- GET http://.../cartoes?elegivelFaturaDigital=true
- GET http://.../cartoes?elegivelParcelamentoFatura=true
- GET http://.../cartoes//ofertasParcelamentosFatura

Em todos os casos acima temos apenas consultas (GET) e os códigos de retorno seguirão o mesmo padrão:

- Quando a resposta é processada com sucesso, trazendo ou não resultados, o código de retorno é um 200 representando sucesso.
- Caso seja passado algum parâmetro de URL apontando para um recurso que não exista () ou (), deve ser retornado um 404 indicando que a URL (recurso) não existe.
- POST http://.../cartoes//faturas//parcelamento

Quando se faz um POST e um recurso é criado, o código de retorno é 201. Se a requisição falhar por culpa de algum atributo mal informado, utiliza-se um 400. Se os parâmetros forem todos

devidamente informados, mas alguma regra de negócio impediu que a contratação do parcelamento ocorresse com sucesso, utiliza-se um 422.

Escrevendo o contrato

A maior parte do trabalho para modelagem de um contrato foi o levantamento que fizemos até este ponto. Daqui pra frente vamos apenas formalizá-lo em um formato que seja interpretado por ferramentas de automação, frameworks, linguagens, catálogos, gateways etc.

Existem algumas linguagens diferentes para declarar contrato de APIs, no entanto, o Swagger ou OAS (**Open API Specification**) é o mais conhecido e suportado.

Como a escrita do contrato em si é a parte mais simples do processo e é amplamente documentada na web, vamos fazer apenas um dos casos que levantamos neste guia e sem entrar nos detalhes da linguagem.

O exemplo a ser escrito em OAS será o `GET http://.../cartoes?upgrade=true` e o `POST http://.../cartoes/{idCartao}/faturas/{idFatura}/parcelamento`. Com o conhecimento gerado para montar estas duas URLs, você já será capaz de entender como fazer as outras e terá o conhecimento mínimo para explorar os materiais na web e fazer seus próprios contratos. Detalhes como autenticação e OAUTH também não serão trabalhados neste exemplo.

Vamos percorrer cada trecho do contrato e entender a sua estrutura.

Observe o código abaixo, repare que a escrita está em **YAML** (que é uma notação que organiza as declarações aninhando com espaços). O OAS permite que o contrato seja escrito também em JSON.

Ali estão definidas a versão do OAS, informações básicas do que se trata o documento e o endereço do servidor onde as URLs deste contrato estão respondendo. Neste guia, até agora usamos as reticências para encurtar as URLs e não ficar repetindo o endereço do servidor o tempo todo, no entanto, no OAS ele deve ser declarado no servers.url.

```
openapi: '3.0.0'
info:
  title: API do Sistema de Cartões do Banco Dourado.
  version: '1.0'
servers:
  - url: http://www.bancodourado.com.br/cartoes/v1
```

Abaixo, preencho os Paths com as URLs, verbos, descrições e parâmetros de entrada e saída que fazem parte da URL.

Observe aninhado abaixo do paths, a presença do **/cartoes** e seu verbo **get** e do **/cartoes//faturas//parcelamento** e seu verbo **post**; cada um dos path parameters (parâmetros que vão no path ou URL) e cada um dos query parameters.

Abaixo de cada rota + verbo, podem existir responses ou requestBody. No caso do /cartoes, preencheremos o response do GET indicando que poderá haver uma resposta HTTP 200 e cujo conteúdo é um application/json com um schema que especifica os atributos da entidade. Vamos ver schemas mais pra frente.

Faremos o mesmo para a rota de parcelamento: declararemos que a resposta será um HTTP 200 representando que a contratação ocorreu com sucesso e referenciaremos o schema que

criaremos a seguir.

```
paths:
  /cartoes:
    get:
      summary: Lista de cartões de um cliente.
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/get-request-cartao'
      parameters:
        - schema:
            type: boolean
          in: query
          name: upgrade
          description: Indica se o cartao tem ofertas de upgrade.
  '/cartoes/{idCartao}/faturas/{idFatura}/parcelamento':
    parameters:
      - schema:
          type: string
        name: idCartao
        in: path
        required: true
      - schema:
          type: string
        name: idFatura
        in: path
        required: true
    post:
      summary: Contratação de uma opção de parcelamento para uma fatura.
      tags: []
      responses:
        '200':
          description: Opção de fatura contratada com sucesso.
          content:
            application/json:
              schema:
                type: object
                properties: {}
      requestBody:
        content:
          application/json:
            schema:
```

```
              $ref: '#/components/schemas/post-request-parcelamento-fatur
a'
```

Agora vamos definir a entidade (ou no OAS: schema) do cartão para que seja usada na rota `GET http://.../cartoes?upgrade=true` que nomeei como get-request-cartao. E faremos o mesmo para objeto enviado no `POST /cartoes/{idCartao}/faturas/{idFatura}/parcelamento` que nomeei como post-request-parcelamento-fatura.

```
components:
  schemas:
    get-request-cartao:
      title: cartao
      type: object
      description: Cartão de crédito
      properties:
        id:
          type: string
          description: Identificador do cartão
          example: '9784364237'
        numero:
          type: string
          description: Numero do cartão mascarado.
          example: '5467********1234'
        nome:
          type: string
          description: Nome do cartão
          example: Cartão XPTO
        nomePortador:
          type: string
          description: Nome do portador do cartão.
          example: José da Silva
        dataValidade:
          type: string
          description: Mês e ano da validade do cartão.
          example: 12/25
        bandeira:
          type: string
          description: Nome da bandeira do cartão de crédito.
          example: Mastercard
        parceiro:
          type: string
          description: Nome do parceiro
          example: Supermercado Grandes Compras
        cvv:
          type: integer
          description: Card Verification Value do cartão.
          example: 987
```

```
        dataMelhorCompra:
          type: string
          description: 'Retorna a data referente a melhor data de compra,
com referência à data de corte do cartão.'
          example: '2016-02-28'
        bloqueado:
          type: boolean
          description: Indica se o cartão está bloqueado.
          example: false
    post-request-parcelamento-fatura:
      description: Parcelamento de fatura
      type: object
      title: parcelamento-fatura
      properties:
        idConta:
          description: Identificador da conta de débito.
          type: string
          example: '4321098765432109'
        idOferta:
          description: Identificador da oferta selecionada na simulação.
          type: string
          example: 12P1
        valorPagamentoMinimo:
          description: Valor do pagamento mínimo da fatura.
          type: number
          example: 123456.78
        quantidadeParcela:
          description: Número de parcelas a serem pagas
          type: integer
          example: 12
        valorParcela:
          description: Valor de cada parcela.
          type: number
          example: 234.56
        dataPrimeiroVencimento:
          description: Data do vencimento da primeira parcela.
          type: string
          example: '2018-06-20'
        dataPagamentoEntrada:
          description: Data do pagamento da entrada.
          type: string
          example: '2018-05-20'
        valorEntrada:
          description: Valor da entrada.
          type: number
          example: 1000.2
        valorUltimoPagamento:
```

```
          description: Valor da última parcela.
          type: number
          example: 134.67
        seguroPrestamista:
          description: Indica se foi contratado o seguro prestamista que
garante a quitação de uma dívida no caso de impossibilidade de pagamento
por motivos maiores.
          type: boolean
          example: false
      required:
        - idConta
        - idOferta
        - valorPagamentoMinimo
        - quantidadeParcela
        - valorParcela
        - valorEntrada
        - valorUltimoPagamento
        - seguroPrestamista
```

Note que na definição do post-request-parcelamento-fatura alguns campos foram declarados na lista do required, pois são obrigatórios no envio.

É isso! Seguindo este padrão, daí pra frente, bastaria criar cada uma das rotas, verbos, requests e responses que especificamos durante a análise.

Palavras finais

O processo de definir um bom contrato de API envolve conhecer e estudar REST e mais importante ainda, envolve conhecer o seu negócio. Minha experiência com times definindo estes contratos mostra que a maior parte do tempo gasto foi em entender o negócio e identificar cada entidade e como se relacionam. Quanto à escrita do contrato, depois de fazer o primeiro, os analistas tiram de letra.

REST não é apenas uma forma de montar um contrato de API, mas um padrão arquitetural que envolve alguns conceitos e algumas restrições. Nem todas as empresas de partida implementam APIs respeitando todas as restrições. É chamada de RESTfull uma API que respeita todas estas restrições. É chamada de REST a API que respeita todos ou pelo menos alguns dos padrões e restrições. Podemos resumir que RESTfull seria o estado da arte na implementação de uma API REST.

Com frequência teremos que fazer sistemas para expor funcionalidades de sistemas legados, aí surge uma grande dificuldade: equilibrar o que é entendido como ideal com o que já está construído sem as melhores práticas. Logo, nem sempre vamos conseguir atingir o RESTFull.

Assim, conhecer REST conceitualmente é muito importante: quando se junta a dificuldade de entender o negócio com a falta de maestria em REST, as coisas ficam um pouco difíceis. Então, deixo como sugestão que busquem bastante conhecimento em REST primeiro e aproveite as oportunidades dos projetos para na prática aprender a entender o negócio.