API REST: PRINCÍPIOS E BOAS PRÁTICAS PARA SERVIÇOS RESTFUL

15 de agosto de 2018 Paulo CezarPaulo Cezar Internet, Software
API, API REST, Arquitetura de Software, Engenharia de Software, REST, RESTful

O objetivo principal deste artigo é conhecer o necessário para criar uma **API REST** e descrever os princípios da engenharia de software que regem este estilo arquitetural. Vamos entender seus principais conceitos, como funciona, quais são suas regras, decisões de projeto importantes a serem levadas em consideração e algumas recomendações e boas práticas.

Muito ouvimos falar sobre API, mas o que é uma API? A definição mais simples, afirma que **API** (*Application Programming Interface*) é uma interface de usuário que é consumida por uma aplicação e não por uma pessoa, permitindo que funcionalidades sejam utilizadas sem conhecimento da implementação de software. Algo que deve ser lembrado sempre que falamos sobre API é que existe um contrato bem definido entre o provedor do serviço e quem fará seu consumo. Este contrato, na verdade, é um conjunto de especificações técnicas que definem a estrutura para realizar requisições e receber as respostas desejadas (*Request/Response*).

Uma API pode ser desenvolvida pensando em três tipos de público alvo e, dependo do tipo, deverá ter diferentes preocupações. Uma API pode ser: (i) Privada: interna para uma organização; (ii) pública: disponível para qualquer pessoa ou entre (iii) parceiros, onde a API é disponibilizada para alguém de confiança. Seja qual for o tipo de API escolhida, algumas considerações básicas deverão ser levadas em consideração, como: segurança, documentação, requisitos de acesso, gerenciamento da API, entre outros.

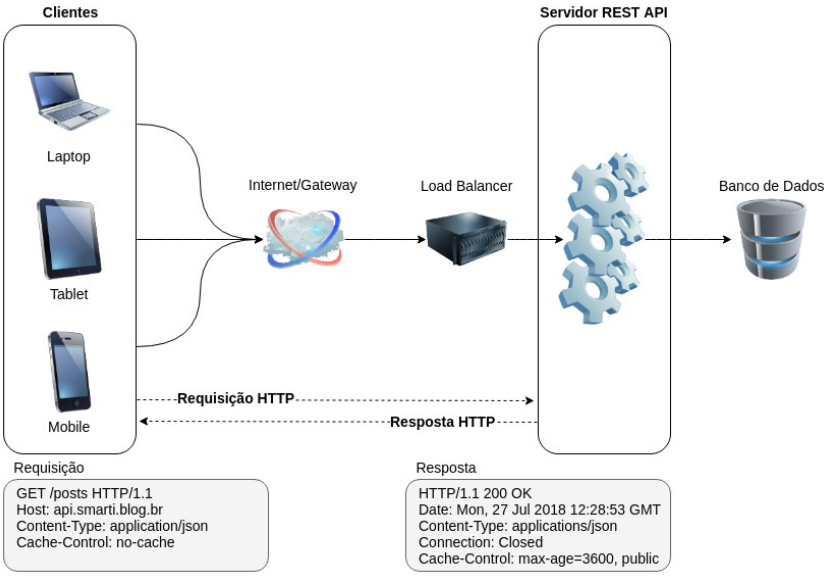
REST

Representational State Transfer (**REST**) é um estilo arquitetural para desenvolvimento de *Web Services* baseado no protocolo HTTP. Isso mesmo, REST, ao contrário do que você ouve falar por aí, não é uma tecnologia específica, um padrão de desenvolvimento ou uma biblioteca. REST é um estilo arquitetural que utiliza determinados padrões, como: HTML, XML, JSON e outros! É um engano pensar que o fato de utilizar o protocolo HTTP com o padrão JSON torna a sua arquitetura RESTful. De acordo

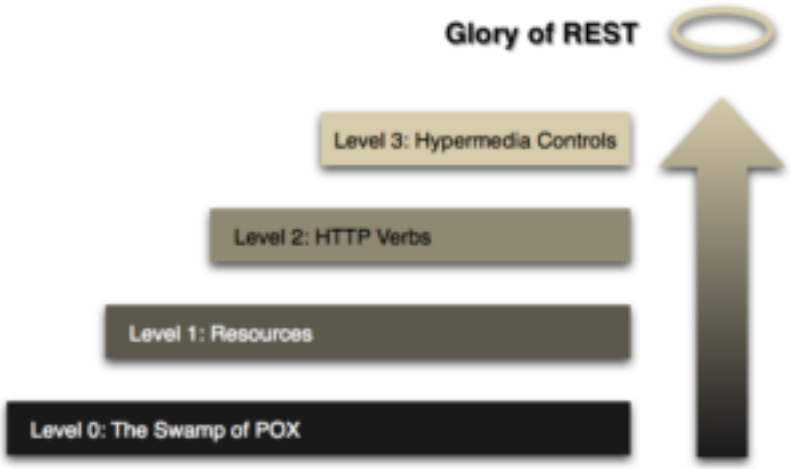
com a tese de doutorado de **Roy Fielding**, um dos principais autores das especificações HTTP, para que uma arquitetura seja considerada **RESTful**, um conjunto de regras e princípios devem ser seguidos:

- **Cliente-Servidor**: A separação das responsabilidades é o princípio por trás do cliente-servidor. Ao separar as preocupações de interface de usuário (UI) do armazenamento de dados, é possível melhorar a portabilidade através de múltiplas plataformas de UI, simplificar os componentes do servidor, mas principalmente, permitir a evolução de forma independente uma vez que não há dependência entre os lados cliente/servidor.
- **Interface Uniforme**: A característica principal que diferencia o estilo arquitetural REST dos demais é uma interface uniforme entre os componentes cliente e servidor. Como o cliente e servidor compartilham esta interface, deve-se ter um “contrato” bem definido para comunicação entre os lados. Há quatros princípios que devem ser seguidos para obter uma interface uniforme: Identificação dos Recursos, Representação dos recursos, Mensagens auto-descritivas e Hypermedia (HATEOAS).
- **Stateless**: A comunicação entre cliente-servidor deve ocorrer independente de estado, não cabendo ao servidor armazenar qualquer tipo de contexto, ou seja, cada requisição deve possuir toda informação necessária para que seja inteiramente compreensível. Este princípio acaba gerando um alto tráfego de dados e redução de performance, porém pode ser contrabalanceado utilizando adequadamente o recurso de cache.
- **Cache**: O cache ajuda a melhorar a performance, a escalabilidade e eficiência uma vez que reduz o tempo de resposta médio quando comparado entre uma série de interações cliente-servidor. As diretivas de cache são controladas pelo servidor através do cabeçalho HTTP (*HTTP Header*).
- **Camadas**: Arquitetura deve ser construída através de camadas gerenciadas de forma independente, onde cada layer não pode ver além do layer adjacente e mudanças de um layer não devem impactar nos demais. É recomendável que o cliente nunca conecte-se diretamente no servidor de aplicação e que uma camada de balanceamento de carga seja adicionada entre

cliente-servidor. A grande vantagem de trabalhar em camadas é que a arquitetura se torna menos complexa e fica mais propensa a mudanças.



Para saber se sua arquitetura é RESTful, ou seja, se ela realmente segue os princípios de design e as regras da arquitetura, é possível utilizar um modelo para medir em qual nível de maturidade a arquitetura se encontra. Este modelo é chamado de **Richardson Maturity Model** (RMM) e possui uma pontuação de 0 a 3. O último nível introduz o conceito de **HATEOAS** (*Hypertext As The Engine Of Application State*) que permite enviar nas respostas informações (links) para navegação dinâmica pela API REST. Por exemplo, ao consumir uma aplicação fazendo a inserção de um novo registro, a API pode lhe retornar o link para este novo registro no corpo da resposta HTTP. Poderá ver um exemplo interessante na sessão de paginação. Caso tenha interesse em conhecer este modelo e suas etapas mais de perto, veja este link.



Desenvolvimento API REST

Recursos (*Resources*)

Uma abstração muito utilizada dentro deste universo é a de Recursos. Se você já teve algum contato com REST, com certeza já ouviu falar sobre. Recurso é algum tipo de informação que é gerenciado pela aplicação. São entidades da aplicação! Por exemplo, se sua aplicação realiza gestão de emissão de notas fiscais, alguns recursos seriam: contribuintes, notas fiscais, empresas, municípios, atividades, etc.

Os recursos devem possuir uma identificação única, para isso, utiliza-se o conceito de URI (*Uniform Resource Identifier)*. Por exemplo, para um serviço que consome uma API REST de gestão pública acessar os dados de uma determinada nota fiscal, poderia utilizar o seguinte URI

```
https://api.smartilab.com.br/v1/notasfiscais/{id}
```

Nem sempre é dado a devida importância aos nomes utilizados, porém quando os recursos são nomeados de forma adequada, a utilização da API se torna muito mais intuitiva e, inclusive, auxilia a reduzir a documentação.

Algumas boas práticas são:

- Utilizar **substantivos** para nomear os recursos e não verbos. Os verbos devem ser utilizados para ações (*Actions*);
- Recomenda-se também adotar os nomes dos recursos no plural (Apesar que boas APIs, como LinkedIn, utilizam no singular);
- Utilizar letras minúsculas;
- Se necessário utilizar separador, utilize hífen para melhor legibilidade.

https://marketplace.walmartapis.com/v3/items/{id} 

https://marketplace.walmartapis.com/v3/getItems/{id} 

Quando há algum tipo de **Associação**, uma boa prática é utilizar junto da URI da associação o próprio *Resource*. Ah… e evite utilizar mais que 3 níveis de associação. Um exemplo bem comum de associação são suas fotos do Facebook, ou seja, através de

níveis da URI é possível acessar um *Resource* diferente. veja o exemplo abaixo.

URI do Perfil do usuário:
`https://graph.facebook.com/v3.1/{profile-id}`

Agora, para acessar a imagem associada ao perfil, basta adicionar **/picture** ao *Resource* do perfil:
`https://graph.facebook.com/v3.1/{profile-id}/picture`

Ações (*Actions*)

Frequentemente é necessário realizar operações que diferem das operações CRUD. Estas operações são chamadas de *Actions*, ou ações. É bastante comum utilizar **verbos** para nomear operações de ação. Por exemplo, se você precisar criar um ação para calcular o imposto de uma nota fiscal, poderia utilizar: `http://baseurl/calculariss`

Outras abordagens podem ser utilizadas, como tentar tratar a ação como um *resource* ou *sub-resources* ou utilizar *querys*. Abaixo podemos ver alguns exemplo interessantes.

API	Descrição	Especificação
GitHub	Utiliza *actions* como subresources para "favoritar" um Gist. É utilizado o verbo PUT para adicionar e DELETE para remover dos favoritos.	`PUT /gists/{id}/star` `DELETE /gists/{id}/star`
Twitter	Utiliza sub-resources	`GET /friendships/lookup`
Uber	Utiliza ações através de *queries* com parâmetros.	`GET /estimates/price?latitude=37.7&longitude=-12.4`

As ações, no entanto, devem ser utilizadas como exceção. Há algumas discussões sobre cuidados a respeito das *Actions* para evitar ferir os princípios de um *Web Service* RESTful. Um exagero no número de ações, por exemplo, pode indicar que talvez outro estilo arquitetural seja mais adequado para sua aplicação, como o RPC.

URI

Com certeza você sabe o que é URL, mas e URI? URI, ou Identificador de Recursos Universal, é o elemento mais simples e importante da arquitetura web. Neste contexto, URI é a interface de comunicação entre cliente-servidor de sua API REST. É através dele que é possível acessar os recursos da aplicação. Na grande maioria dos casos, a URI de acesso é o que chamamos de URL, pois uma URI pode ser uma URL. Já o contrário nem sempre é verdade! Mas este não é nosso foco, caso queira entender melhor sobre URI e sua relação com URL/URN, veja este link.

Uma vez que a URI é a porta de entrada para sua aplicação, é muito importante que seja evitado alterações, pois pode haver impacto nos serviços que já estão consumindo sua API. Caso seja inevitável, procure manter o recurso antigo ou faça um redirecionamento (falaremos mais na sessão de versionamento). Também busque utilizar nomes simples que estejam dentro do contexto da aplicação e adote um padrão que seja reutilizado para toda aplicação.

Algumas boas práticas são:

- Mantenha o URL base simples;
- Não utilize o mesmo domínio de sua página para a aplicação;
  `http://marketplace.walmartapis.com`
- Se possível, utilizar um subdomínio distinto;
  `http://api.twitter.com`
- Adote um padrão de nomenclatura a ser utilizado
- Indique a versão na URI
  `http://api.uber.com/v1.2/`

A regra de Interface Uniforme para a arquitetura REST, como vimos acima, sugere um contrato bem definido entre cliente-servidor. Este contrato é composto por 3 partes: (i) métodos HTTP, (ii) Códigos de Status HTTP e (iii) Representação de formatos.

Métodos HTTP

Um dos princípios de uma arquitetura RESTful é a manipulação dos recursos através de métodos do protocolo HTTP. Ou seja, os recursos de uma aplicação podem ser manipulados de diversas formas e são os métodos HTTP que indicam os diferentes tipos de

operações que poderão ser executadas ao realizar a requisição de um serviço. Os principais métodos HTTP utilizados são os método que representam as operações **CRUD**, porém há outros métodos HTTP disponíveis se necessários. Já que os próprios métodos deixam subentendido o tipo de ação que será executado em um determinado recurso, evite utilizar na URI nomes de operações do tipo: `/cadastrar` ou `/atualizar`. Após realizar uma requisição com os métodos HTTP, é possível retornar códigos de status que indicam a situação da requisição, conforme sessão a seguir.

[...] https://marketplace.walmartapis.com/items/cadastrar

[...] https://marketplace.walmartapis.com/items/atualizar

POST https://marketplace.walmartapis.com/items

PUT https://marketplace.walmartapis.com/items

Verbo HTTP	Descrição	Resposta
POST	Criar um Recurso	Pode retornar o link(id) para o novo recurso ou o objeto.
GET	Ler um Recurso	Retorna a resposta no formato solicitado.
PUT	Atualizar um Recurso por completo. Todos atributos.	Pode retornar ou não o recurso.
PATCH	Atualizar parte de um Recurso existente.	Pode retornar ou não o recurso.
DELETE	Deletar um Recurso	Pode retornar o objeto deletado ou não (204)
Outros	Connect, Head, options, trace (+ info)	

Códigos de Status HTTP

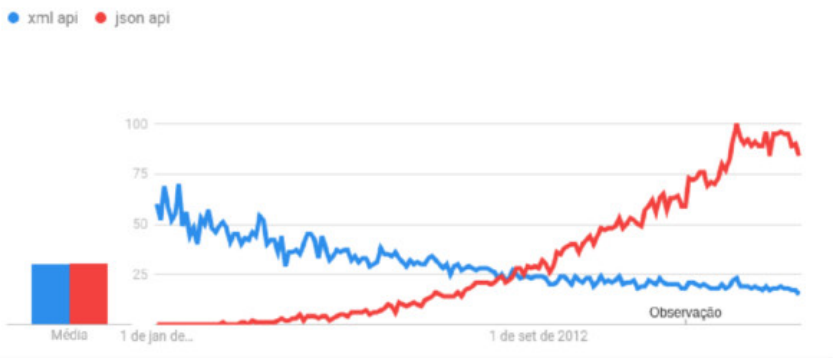
Os códigos de status indicam a situação de uma operação requisitada à API REST e é exibida na mensagem de retorno (*Response*) do servidor. O uso correto destes códigos auxiliam muito o cliente que está consumindo a aplicação. Há diversos códigos (todos de 3 dígitos) que se enquadram em 5 categorias. Quando o projeto é desenvolvido sem o planejamento adequado, é comum ver mais de um código de status ser utilizado para situações similares, principalmente quando vários programadores trabalham numa mesma API. Logo, uma recomendação é que como uma decisão de projeto, seja restringido o número de código de erro utilizados na aplicação a no máximo 8. Os códigos de uso

mais comum são: **200, 201, 400, 401, 403, 415** e **500**. Veja neste link todos códigos e descrições.

Categoria	Descrição	Exemplos
1xx	Mensagens Informativas	100, 102
2xx	Indicação de que a Requisição foi recebida com Sucesso	**200, 201**, 204
3xx	Indicação de Redirecionamento	307
4xx	Indicação de Erro por parte do *Client*	**400, 401, 403,415**
5xx	Indicação de Erro por parte do *Server*	**500**

Representação de Formatos

Ao fazer uma requisição para um *Web Service,* como sabemos se os dados retornados serão no formato JSON, XML, CSV, ou qualquer outro? Se não sabemos, como nosso sistema irá tratar estes dados? Geralmente, a maior parte das APIs utilizam o JSON como formato padrão para representação de seus dados. Fica claro através da figura abaixo que nos últimos 10 anos o uso do XML tem caído consideravelmente mundo a fora.



Todo o mundo. 01/01/2004 – 02/08/2018. Pesquisa Google na Web.

Para agregar mais valor à sua API REST e garantir um suporte mais amplo para os clientes que irão consumir o seu serviço, recomenda-se como uma decisão de projeto, que sua arquitetura REST seja escalável no sentido de suportar diversos tipos de formatos de dados. Isso não significa que deverá desenvolver a API com suporte a JSON, XML, etc. Afinal, este pode não ser um fator de alta relevância no início do projeto, gerando mais custo e tempo de desenvolvimento. No entanto, se a arquitetura foi projetada para ser escalável e, futuramente, for necessário fazer a

evolução da API para suportar outros tipos de formato de dados, então isso não será um problema.

A informações sobre os tipos de dados suportados e a forma de receber a resposta no formato desejado devem ser documentadas e informadas no portal do desenvolvedor. Há 3 formas do cliente especificar o formato durante a requisição ao fazer o consumo de sua API REST. O método *Accepts* do HTTP Header é o menos comum.

	Descrição	API	Especificação
1	Parâmetros de Query	**Walmart**	`/contribuintes/{id}?format=xml` `/contribuintes/{id}?format=json`
2	Sufixo	**BBC**	`/programmes/schedules/fm/today.xml` `/programmes/schedules/fm/today.json`
3	HTTP Header Accepts	**Digg**	`Accepts: application/xml` `Accepts: application/json`

Mudanças e Versionamento

Realizar mudanças na API é algo que deve ser feito com muita cautela, pois estas alterações além de ocasionar mudanças internas, podem impactar nas aplicações externas que estão consumindo o seu serviço. Imagine que um *Resource* da sua aplicação foi desenvolvido para retornar uma String, porém houve uma evolução do sistema e este Recurso passou a retornar um Array. Este tipo de mudança quebraria os sistemas que já estão consumindo o seu serviço. Portanto, o melhor é avaliar se a mudança em questão é realmente importante e se irá, de fato, agregar valor à API REST. Se possível, evite estes tipos de mudanças.

Uma boa prática é manter um **versionamento** adequado da aplicação, através do gerenciamento de múltiplas versões. O versionamento em questão não é referente ao controle de versão

de código fonte, mas sim dos lançamentos (*Releases*) da API. O versionamento ajuda numa evolução mais rápida, a prevenir requisições inválidas por alterações de URI, permite uma transição mais suave entre versões uma vez que é possível manter versões depreciadas, etc.

Há duas decisões de projeto que o arquiteto/engenheiro de software deve tomar em relação ao versionamento. (i) Como o cliente irá especificar a versão; e (ii) Qual será o formato da versão. O cliente pode especificar a versão através do *HTTP Header*, parâmetros de *Query* ou então pelo URL. Segue abaixo exemplo de algumas APIs.

API	Descrição	Especificação
Uber	Opção *major.minor* no parâmetro	`/payment-methods?version=3.1`
LinkedIn	Apenas a opção *major*na URL	`/v1/companies` (mais comum)
GitHub	Apenar a opção Major através do cabeçalho HTTP	`Accept: application/vnd.github.v3`

Cache

O cache é a ação de armazenamento de uma resposta da aplicação pelo cliente ou por pontos intermediários entre o cliente e a API. Desta forma, o cliente evita ter que refazer uma requisição já feita anteriormente. Inicialmente pode parecer estranho o termo "pontos intermediários entre o cliente e a API", mas lembre-se que até sua requisição chegar ao destino, os dados passam por diversos outros servidores nesta linda e emaranhada rede, a qual chamamos de internet 😛 . O principal motivo que leva o Cache a ser utilizado é a performance, porém outros dois fatores bastante relevantes para uma API REST são a escalabilidade e o *Throughput (*taxa de transferência).

O Controle do comportamento do Cache é feito através de diretivas que nos ajudam a responder a perguntas, como: Quem pode fazer cache, por quanto tempo e em quais condições os dados podem ser cacheados? Para isso, deve-se levar em consideração a velocidade das mudanças, a sensibilidade ao tempo e a segurança necessária. Tudo isso, para que seus dados sejam disponibilizados de forma adequada. Você não gostaria que seus dados de cartão

de crédito fiquem armazenados por ai, certo? Neste caso, a diretiva de permissão para fazer cache não deve ser pública.

Algumas boas práticas são:

- Utilize o cache principalmente quando há alto volume de dados;
- Utilize a diretiva ***no-store*** ou ***private*** para dados sensíveis;
- Utilize o eTag do *HTTP Header* quando há respostas grandes;
- Decida cuidadosamente o tempo de cache (***max-age***).

Resposta Parcial e Paginação

Dois tópicos bastante interessante que não poderia deixar de falar são: *Partial Respose* e *Pagination*. Muitas pessoas acreditam que são nomes diferentes para a mesma coisa, mas não são! E entender como e quando usá-los pode fazer bastante diferença em sua API.

Resposta parcial, ou *Partial Response,* é a possibilidade o retornar diferentes respostas de um recurso de acordo com a necessidade. Por exemplo, um determinado cliente pode optar por exibir apenas um resumo de informação para uma aplicação mobile, mas para um browser padrão, permitir a exibição completa dos dados. Esse recurso é benéfico por possibilitar a otimização do sistema, melhorar a performance e deixar esse controle de granularidade da informação como responsabilidade do cliente.

Pode-se representar a reposta parcial de duas formas, sendo a segunda opção mais comum:

API	Especificação
LinkedIn	`/people:(id,first-name,last-name)`
Facebook	`/{id}/photos?fields=height,width,link` (mais comum)

Assim como o *Partial Response*, a **Paginação** também permite o controle de resposta pelo lado do consumidor da API, mas por sua vez, através da informação do número de objetos que deseja retornar e o descocamento desejado. Por exemplo, para um mesmo *resource,* o cliente pode retornar 50 objetos por página para versão Web e apenas 10 para a versão mobile. Para

satisfazer a regra de **HATEOAS**de arquiteturas RESTful, ainda poderá retornar os links (URI) da página anterior, próxima página, primeira e última página. Este recurso também oferece como benefício a melhora de performance e uso otimizado de recursos, como CPU, memória, etc.

Uma decisão de projeto, é definir como será feito a paginação. É possível fazê-la de três formas: Cursor, *OffSet* e através do uso de *HTTP Header*. Atualmente a abordagem mais popular é baseada em *offset*. Porém a que obtêm melhor performance é através de cursor.

API	Especificação
LinkedIn	`/people?start=5&limit=5`
Facebook	`/friends?offset=5&limit=5` (mais comum)
Twitter	`/search?page=3&rpp=5`

Por exemplo, ao consumir o recurso `/contribuintes?offset=5&limit=3` com o deslocamento (*offset*) de 5 objetos e limitando o retorno para 3 objetos, temos a seguinte resposta:

```
{
   "_links":{
"base":"http://api.smartilab.blog.br/contribuintes",
      "prev":"/page?offset=0&limit=5",
      "next":"/page?offset=10&limit=5",
"self":"http://api.smartilab.blog.br/contribuintes/page"
   },
   "limit":3,
   "results":[
      {
         "id":"0006",
         "name":"Paulo",
      },
      {
         "id":"0007",
         "name":"Roberto",
      },
      {
         "id":"0008",
         "name":"Cezar",
      },
   ],
   "size":3,
   "offset":5
}
```

Gerenciamento da API REST

O gerenciamento da API REST é o processo de publicação, documentação e supervisão da aplicação num ambiente seguro e escalável. Faz parte de seu escopo uma série de atividades, como: ciclo de vida, produtividade, segurança, tráfego e análises de dados, monetização e a transformação do serviço em produto. Plataformas como Apigee, Akana, WSO2, IBM API Connect, entre outras, são amplamente utilizadas na industria para realizar o gerenciamento da API. Hoje, a plataforma **Apigee** é a mais utilizada.

Há diversas plataformas de padronização/especificação de uma API REST, como o WADL, Apiary, Swagger, e outros. Dentre estas plataformas, o **Swagger** é o mais utilizada na industria e é suportado por praticamente todas grandes aplicações de gerenciamento de API do mercado. O Swagger pode ser utilizado em todo ciclo de vida da API e pode auxiliar desde a modelagem e documentação, até teste e implantação. Deixo como dica que leia mais a respeito das plataformas citadas, pois farão uma grande diferença para seu projeto. Seja pela adotação de uma padrão bem definido para interface de usuário, uma modelagem em linguagem mais natural, auxílio para documentação e implementação de códigos, a criação de políticas, a análise de dados para tomadas de decisão, dentre muitas outras vantagens.

Já, do lado do cliente, os desenvolvedores que irão fazer os sistemas para consumo da aplicação, são a parte mais importante da **cadeia de valores de uma API REST**. Logo, é importante que ele tenha em mãos todas as informações necessárias. Para isso, deve-se criar um **Portal do Desenvolvedor**, como um local único onde tenha acesso a toda documentação, possa solicitar requisição de acesso a API e tenha acesso a qualquer outro tipo de suporte. Uma boa prática é ter recursos do tipo *Try It (tester)* para que o desenvolvedor compreenda o funcionamento da API REST sem a necessidade de escrever qualquer código. Talvez não tenha associado o nome ao recurso, mas um exemplo de try it, seria os formulários que permitem a conexão com a API indicando os argumentos necessários e exibindo a resposta do *resource*. Também é recomendável que tenha códigos de exemplos,

diagramas, SDKs, etc. Segue abaixo alguns portais de API REST como exemplo:

API	Portal do Desenvolvedor
Github	https://developer.github.com/
Facebook	https://developers.facebook.com/
Walmart	https://api-mp.walmart.com.br/
LinkedIn	https://developer.linkedin.com

Referências:

Architectural Styles and the Design of Network-based Software Architectures. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Why Some Web APIs Are Not RESTful and What Can Be Done About It. https://www.infoq.com/articles/web-api-rest

Best Practices for Designing a Pragmatic RESTful API. https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api