Osvaldo Santana Neto

A Web é uma API

Osvaldo Santana Neto

A Web é uma API

... REST

Eu vou reapresentar o básico porque no básico encontraremos os fundamentos.

Disclaimer para os mais experientes

A Web

No princípio era a Web

Berners-Lee* 1:1

* Inventor da Web

World Wide Web

O que é?

- ***Resource*** (Recursos/Conteúdo/Documento)
  - Representação (MIME Types: `text/html`, `application/json`)
- Endereçamento (Uniform ***Resource*** Locator)
- Protocolo (HTTP)
- Arquitetura Cliente/Servidores (*browser* & HTTP Server)
- Stateless

“A Web é um conjunto de recursos, que podem ser acessados por um cliente, localizados em algum local apontado por uma URL (Unified Resource Locator). Uma representação desses recursos serão servidos através do protocolo HTTP.”

World Wide Web

Resources, URLs e Representação

- Resource vs Representação: documentos, streams, imagens, áudio, vídeo, etc
  - Possuem um tipo associado (`Content-Type`) definido com um *MIME type*
  - **O mesmo *resource* pode ser representado com tipos diferentes**
- Apontados por um localizador (**URL**)
  - `http://example.com/index.html`
  - Uma **URL** sempre aponta para um **único *resource***. Uma **lista** de resources é **um resource** do tipo lista de *resources.* Múltiplas URLs podem apontar para o mesmo *resource* (uso de canonical URLs)

“Lembre-se: Resources e Models são conceitos diferentes”

HTTP, Clientes e Servidores

- Protocolo HTTP
  - Protocolo de comunicação Cliente/Servidor que opera na camada de Aplicação
  - A primeira versão do HTTP é baseada em texto. As versões seguintes podem funcionar com dados binários
  - Existe uma extensão que adiciona suporte a criptografia que é chamada HTTPS
- Clientes
  - Existem vários tipos de clientes (*User Agents*). O mais conhecido é o navegador (*browser*)
  - Os clientes usam HTTP para fazer requisições para um endereço onde tem um servidor Web
- Servidores
  - O servidor Web recebe requisições feitas por clientes e devolvem os recursos solicitados usando também o protocolo HTTP
  - Por padrão, em redes TCP/IP, o servidor recebe as requisições através das portas 80 (http) ou 442 (https)

Request

Um cliente faz uma requisição para o servidor...

- O request especifica uma ação com um verbo. Existe um conjunto padronizado de verbos tais como: `GET`, `POST`, `PUT`, `DELETE`, `PATCH`, etc
- O request especifica a URL do recurso desejado
- O request pode incluir outros detalhes sobre o tipo de resource ou como ele prefere receber esse resource
  - Usamos cabeçalhos do protocolo para esse propósito (ex. `Accept`)
  - Você **pode especificar** qual a **representação** (MIME type) você quer ter do recurso (Accept). **Exemplo: `text/html`, `application/json`**
- `Content-Type` especifica o formato da requisição. `Accept` especifica o formato que você prefere a resposta. Você pode informar uma lista qualificada de *MIME types*

Request

Exemplos

- Todos os exemplos ao lado solicitam **um único** resource com a lista de todas as tarefas (TODO list)
- O terceiro exemplo requisita preferencialmente o resource com representação `application/json` mas aceita `text/html` se for uma alternativa viável para o servidor
- Você tem que lidar com respostas `300 Multiple Choices` e `406 Not Acceptable`

```
> GET /todos HTTP/1.1
> Host: todos.com
> User-Agent: …
> Accept: text/html
```

```
> GET /todos HTTP/1.1
> Host: todos.com
> User-Agent: …
> Accept: application/json
```

```
> GET /todos HTTP/1.1
> Host: todos.com
> User-Agent: …
> Accept: text/html;
          q=0.2, application/json
```

Response

... que devolve uma resposta para o cliente

- A resposta inicia com um código de status
  - `https://github.com/osantana/staty`
- A resposta também inclui cabeçalhos HTTP com metadados da resposta (ex. tipo de representação do conteúdo)
- Na sequência, dependendo do status code, recebemos o conteúdo do resource representado com um dos formatos solicitados na requisição.

Response

Exemplos

- Todos os exemplos ao lado solicitam **um único** resource com a lista de todas as tarefas (TODO list)
- O terceiro exemplo requisita preferencialmente o resource com representação `application/json` mas aceita `text/html` se for uma alternativa viável para o servidor
- Você tem que lidar com respostas `300 Multiple Choices` e `406 Not Acceptable`

```
< HTTP/1.1 200 OK
< Cache-Control: [cache info]
< Content-Type: text/html
< Etag: [tag]
< Expires: [data]
< Last-Modified: [data]
<
< [HTML]
```

```
< HTTP/1.1 200 OK
< Cache-Control: [cache info]
< Content-Type: application/json
< Etag: [tag]
< Expires: [data]
< Last-Modified: [data]
<
< [JSON]
```

HTML

Um tipo de representação para um *resource*

- Documentos Hipertexto
  - Conceito já explorado na literatura desde a década de 40
  - Marcações (tags) especificam a semântica do conteúdo
- Content Type (*MIME type*: `text/html`)
  - Linguagem de marcação subset do SGML
  - Suporte a hiperlinks
- Clientes do tipo Web Browser sabem renderizar esses documentos
  - Usam CSS e JS para especificar visual e comportamento
- Pode ser gerado por um renderizados de linguagem de template.

As APIs

HTTP

HTTP como transporte

- RPC
  - XMLRPC
  - SOAP
  - gRPC
- CORBA
  - 🙄
- Vamos pular isso aqui...

REST e a Web: somos um

Roy Fielding* 10:30

*** Um dos criadores da especificação HTTP e co-fundador da Apache**

REST

Representational State Transfer

- Estilo de arquitetura
- Dissertação de doutorado de Roy Fielding (capítulo 5)
  - https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
    https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
    (ah a ironia! como usar a Web do modo errado —^)
- O estilo REST não se aplica apenas a aplicações Web. Mas tem um “fit” perfeito.

REST

Características

- Client / Server: força separação de responsabilidades
- Stateless: todo request é completo. Contém todas as informações necessárias para ter a resposta certa (Sua aplicação Web sobrevive aqui?)
- Cacheable: as respostas devem oferecer (Bye Bye GraphQL!)
- Uniform Interface: garantido pelo combo HTTP + convenções
- Layered System: similar ao conceito de polimorfismo em OO
- Code on demand (opcional): código sendo entregue pra execução no cliente (Ajax/JS? alguém?)

JSON

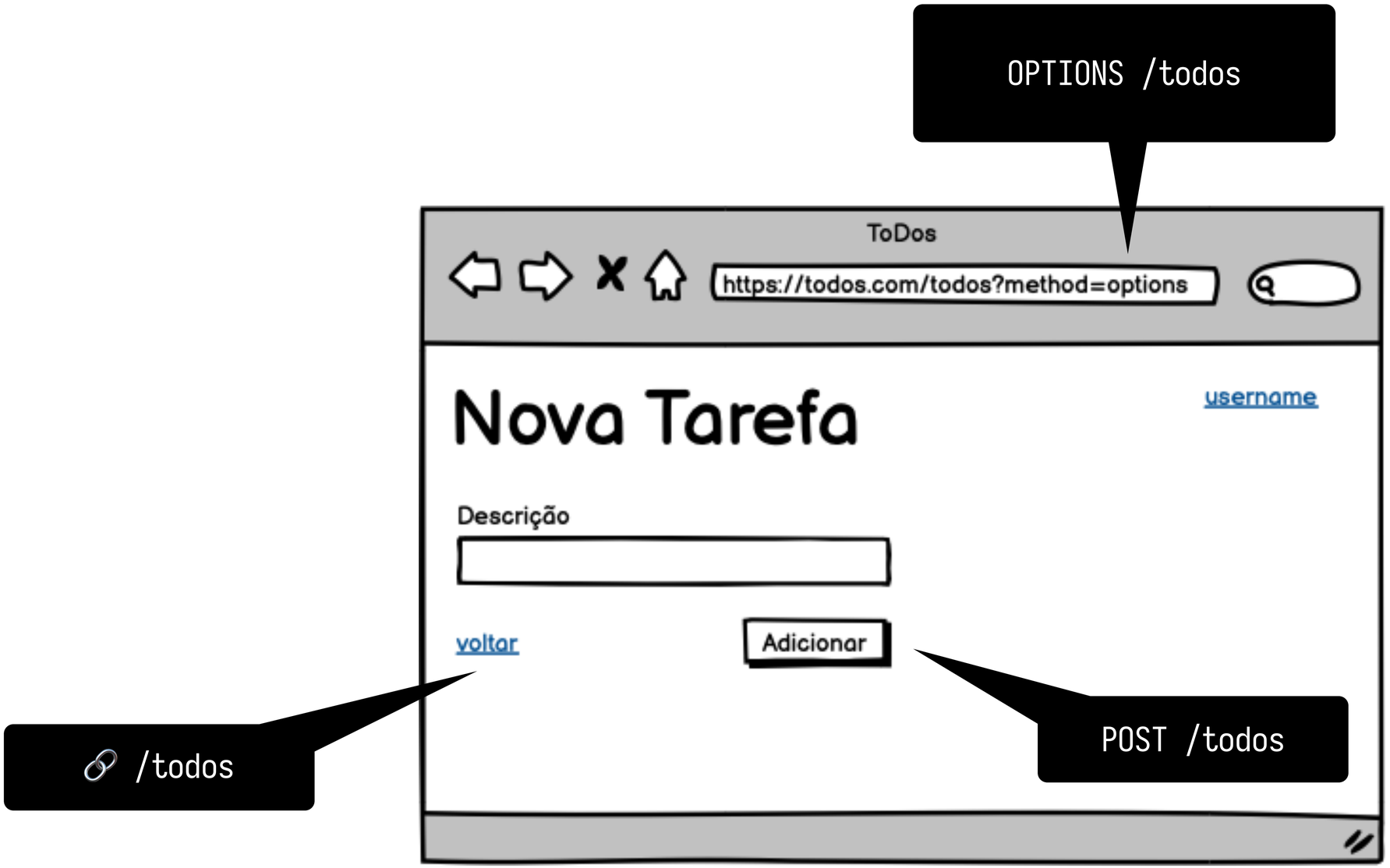
Um tipo de representação para um *resource*

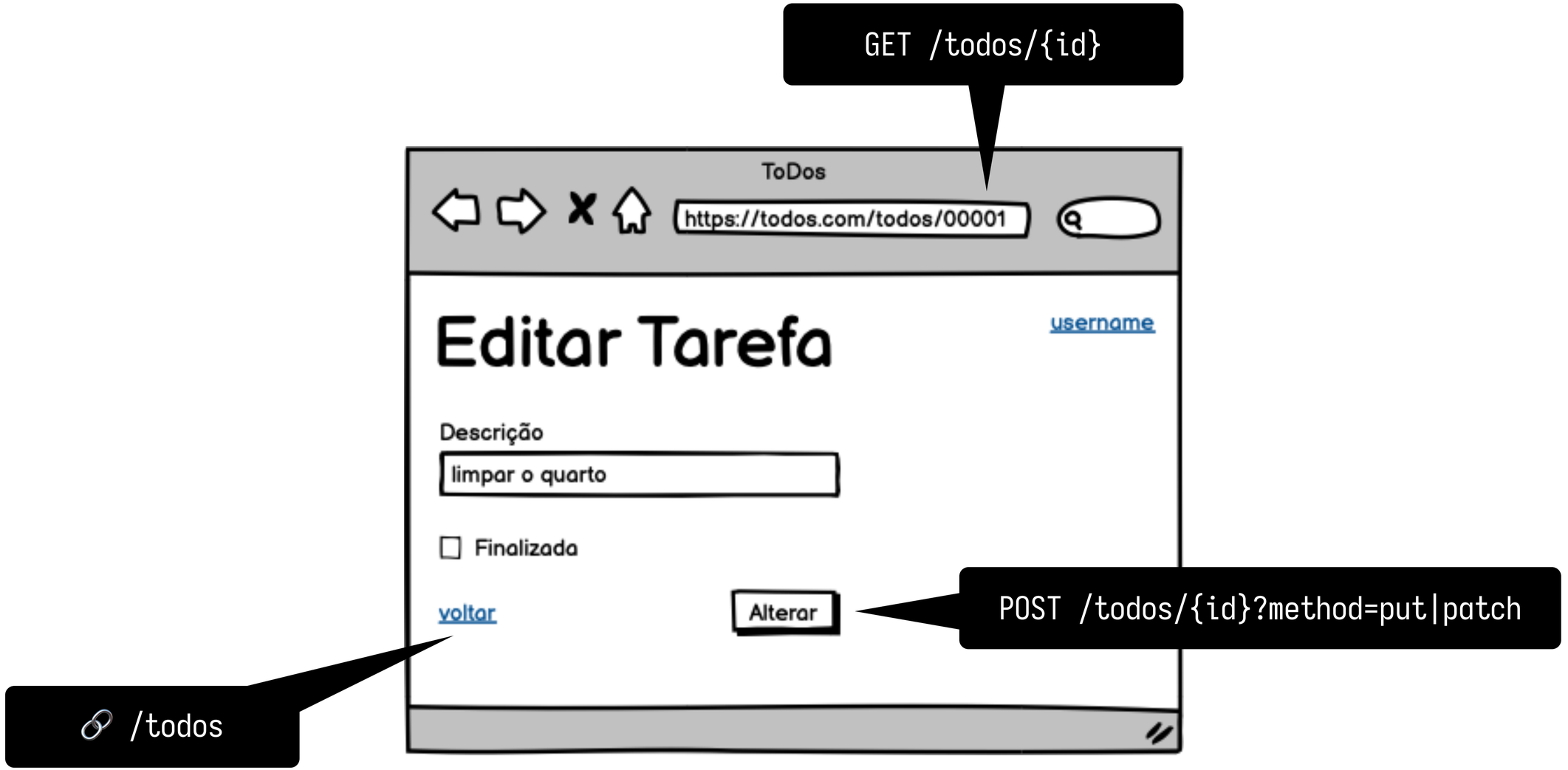
- Serialização de objetos Javascript
- Content Type (*MIME type*: `application/json`)
- Pode ser gerada por um serializado de resources

Uma App Web “Raiz”

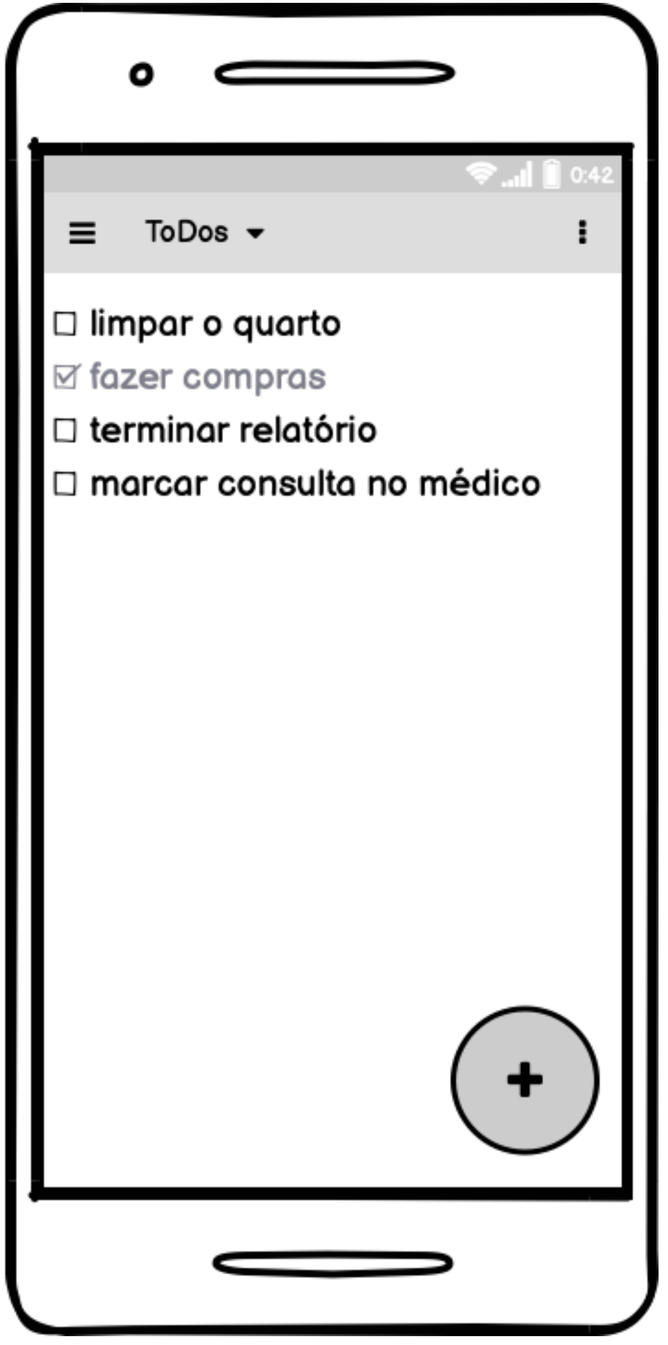
Porque SPAs são uma solução em busca de um problema


GET /todos
ToDos
https://todos.com/todos
username
Lista de Tarefas
edit remove limpar o quarto
edit remove fazer compras
edit remove terminar relatório
edit remove marcar consulta no médico
JS/PATCH /todos/{id}
dica: JSON/PATCH
/profile
/todos/{id}
DELETE /todos
/todos?method=options


OPTIONS /todos
ToDos
https://todos.com/todos?method=options
Nova Tarefa
username
Descrição
voltar
Adicionar
🔗 /todos
POST /todos


GET /todos/{id}
ToDos
https://todos.com/todos/00001
Editar Tarefa
username
Descrição
limpar o quarto
Finalizada
voltar
Alterar
POST /todos/{id}?method=put|patch
🔗 /todos

e por último…


0:42
ToDos
limpar o quarto
fazer compras
terminar relatório
marcar consulta no médico

Precisamos de uma API

https://api.todos.com/ (?)

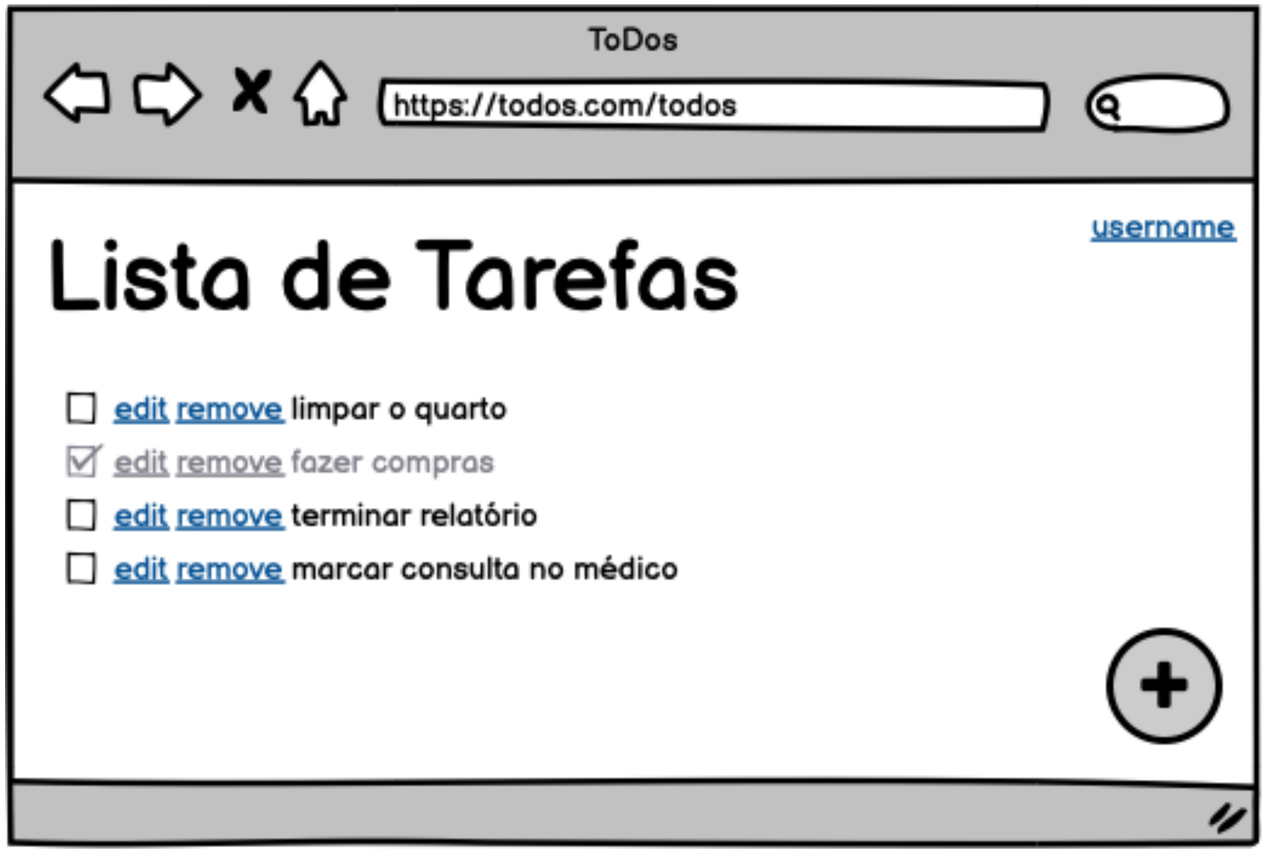
Precisamos de uma API

https://[illegible].com/ (?)

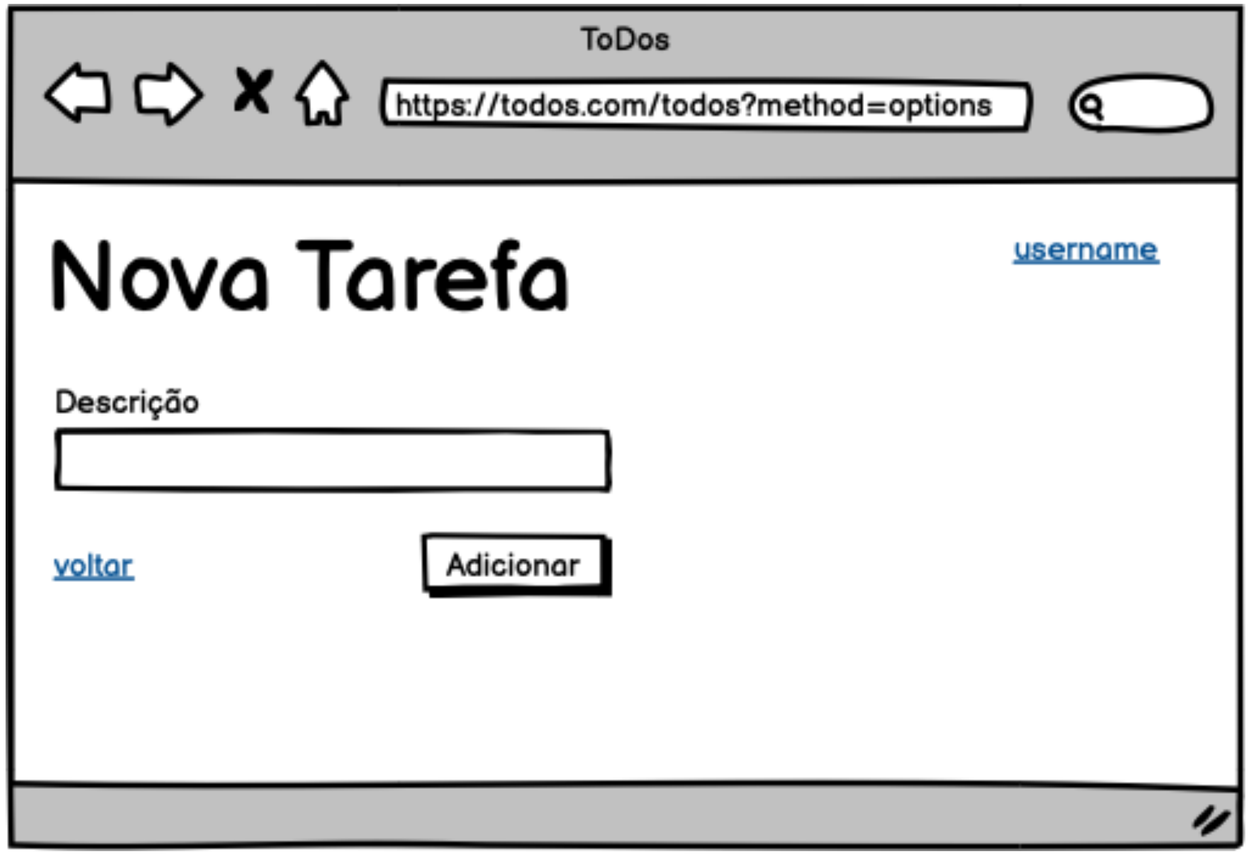
não!

Porque separamos App e API?

E se?



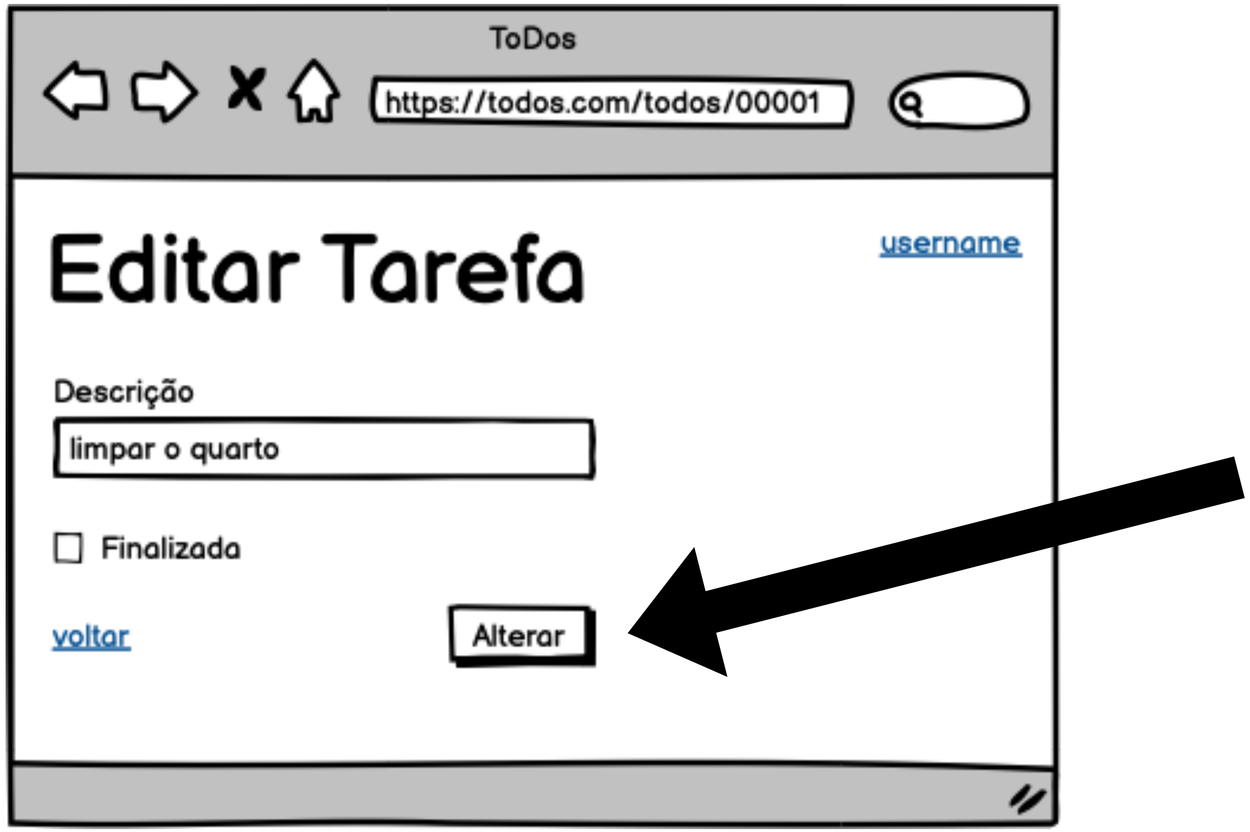

```
> GET /todos
> Accept: application/json

<<<
[HEADERS]

{
  "_context": {
    "username": "username"
  },
  "offset": 0, "limit": 10,
  "objects": [
    {
      "_href": "/todos/00001",
      "id": 1,
      "description": "limpar o quarto",
      "checked": false
    },
    ...
  ]
}
```




```
> OPTIONS /todos
> Accept: application/json

<<<
Allow: OPTIONS, GET, POST, PUT, PATCH
[MORE HEADERS]

{
  "_context": {
      "username": "text"
  },
  "_meta": {
    "id": "number",
    "description": "text",
    "checked": "boolean"
  }
}
```

➡Ou ainda um trecho da especificação OpenAPI que descreve o resource.


```
> GET /todos/00001
> Accept: application/json

<<<
[HEADERS]

{
  "_context": {
      "username": "text"
  },
  "object": {
    "_href": "/todos/00001",
    "id": 1,
    "description": "limpar o quarto",
    "checked": false
  }
}
```




```
> PUT /todos/00001
> Content-Type: application/json
> Accept: application/json
>
> {
>   "description": "limpar o quarto",
>   "checked": false
> }

<<<
200 OK
```

Problemas & “Soluções”

Dados de sessão

Como fazer isso de modo “stateless”?

- Criar o conceito de “Embedded Resource” com dados globais (ex. usuário logado, timezone configurado, idioma selecionado, etc).
- Disponibilizar isso para o renderizados do resource embutir no resource

```
{
  "_context": {
    "username": "foo",
    ...
  },
  "object": {
    "task": "limpar o quarto",
    ...
  }
}
```

Suporte a verbos

Navegadores suportam GET/POST

- A tag <form> suporta apenas GET e POST como métodos HTTP.
- Utilizar JS para extender o funcionamento do navegador.
- Usar o header HTTP não-padrão (`X-HTTP-Method-Override`) ou uma *query string* que o servidor entenda como

```
> GET /todos?method=head HTTP/1.1
> Host: todos.com
> User-Agent: …
> Accept: text/html
```

Queries e Embedded Resources

Resources muito leves ou muito pesados

- Uso de query string
- Tipo de problema resolvido pelo GraphQL caso não ele não violasse o estilo REST
- restQL é uma alternativa padrão viável (desenvolvida no Brasil pela B2W):
  - http://restql.b2w.io

Formulários Interativos / UX

- Use componentes JS para isso. A abordagem Web/API não invalida ou impede o uso de Web Components frontend
- Web Components nativos (suportados, pelo Polymer, Vue e React) são complementos perfeitos pra essa abordagem
- Abordagem híbrida com técnicas de Hotwire (https://hotwired.dev)

Isso já existe?

- Não existe em nenhum framework muito famoso mas não deve ser complicado implementar os conceitos em seu framework favorito
    - Necessário chamar o “renderizador” adequado de acordo com o Accept do request, lidar com as sobrecargas de métodos HTTP para os navegadores e implementar um componente “Resource" para representar os seus resources
- Existe um esboço no meu framework Toy (um framework de brinquedo)
    - Aviso: use o Toy para estudar mas nunca em produção
    - O seletor de render baseado em MIME type ainda não funciona 100%
    - https://github.com/osantana/toy
- O projeto do Pactum arranhava essas ideias:
    - https://github.com/pactum-org/pactum

Toy

Um framework de brinquedo