Aula 13 - REST

Definição

`REST` é um acrônimo para **Re**presentational **S**tate **T**ransfer. É uma arquitetura para sistemas de hipermidia distribuidas.

Em linhas gerais, a arquitetura REST permite definir um sistema de gerenciamento de dados usando como base o protocolo HTTP/HTTPS.

Princípios REST

A arquitetura REST segue 6 princípios:

1. **Cliente/Servidor** - separa o armazenamento de dados da interface de usuário
2. **Stateless** - toda requisição do cliente precisa conter toda informação necessária.
3. **Cacheable** - As respostas à uma requisição podem conter informações sobre a reutilização de uma resposta

4. **Interface Uniforme** - aplica o conceito de generalização de engenharia de software, padronizando o acesso aos dados da aplicação
5. **Sistema de Camadas** - permite a composição hierarquica de acesso aos dados
6. **Código sob-demanda** - permite a criação de clientes que podem requisitar informações e extende-las sem a necessidade de alterar a arquitetura do servidor.

URI - Unified Resource Identifier

A abstração chave na arquitetura REST é o `resource` (recurso). Toda informação que pode ser nomeada pode ser um resource na implementação REST. Por exemplo:

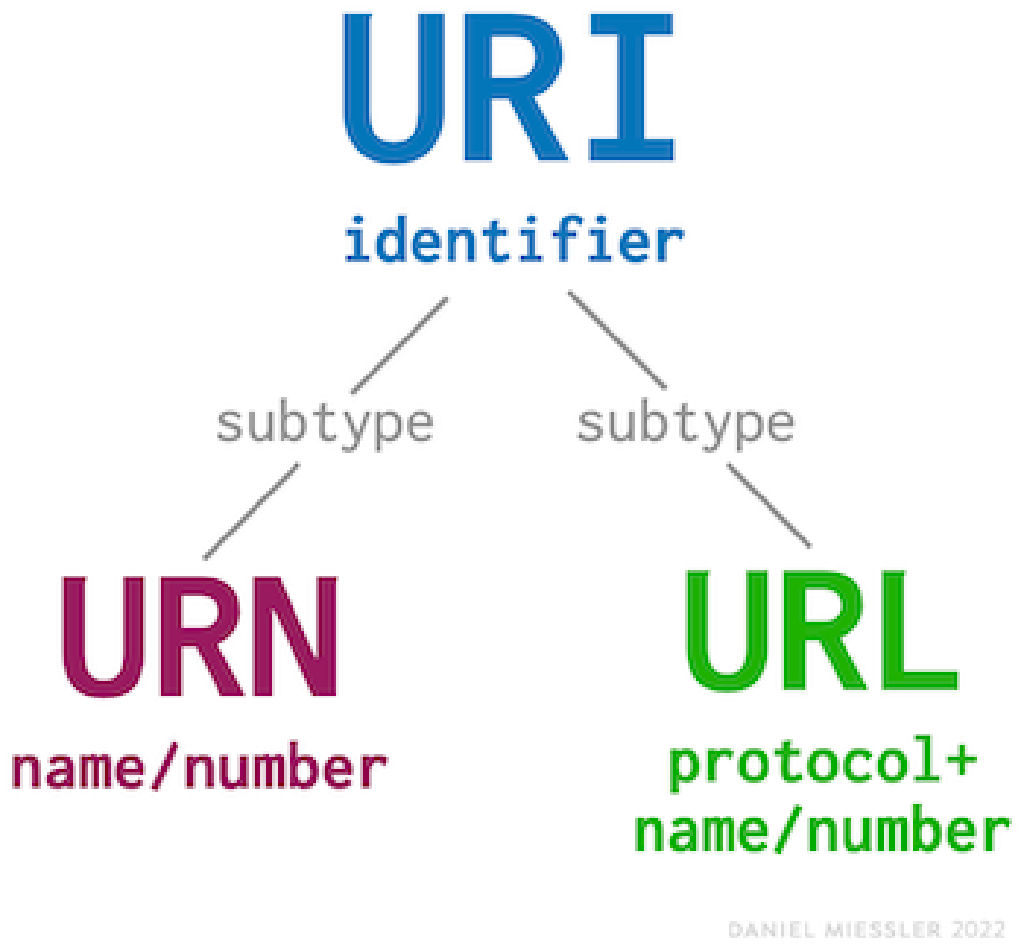
- Usuário
- Foto
- Jogador de Futebol
- Pokémon
- Calendário

Um recurso pode ser identificado de maneira hierarquica. Dessa forma, nascem os identificadores chamados de URI (Unified Resource Identifier)

```
/usuarios
/usuarios/fotos
/time/jogador
/treinador
/treinador/pokemon/25
/calendario/2024/11/25
```

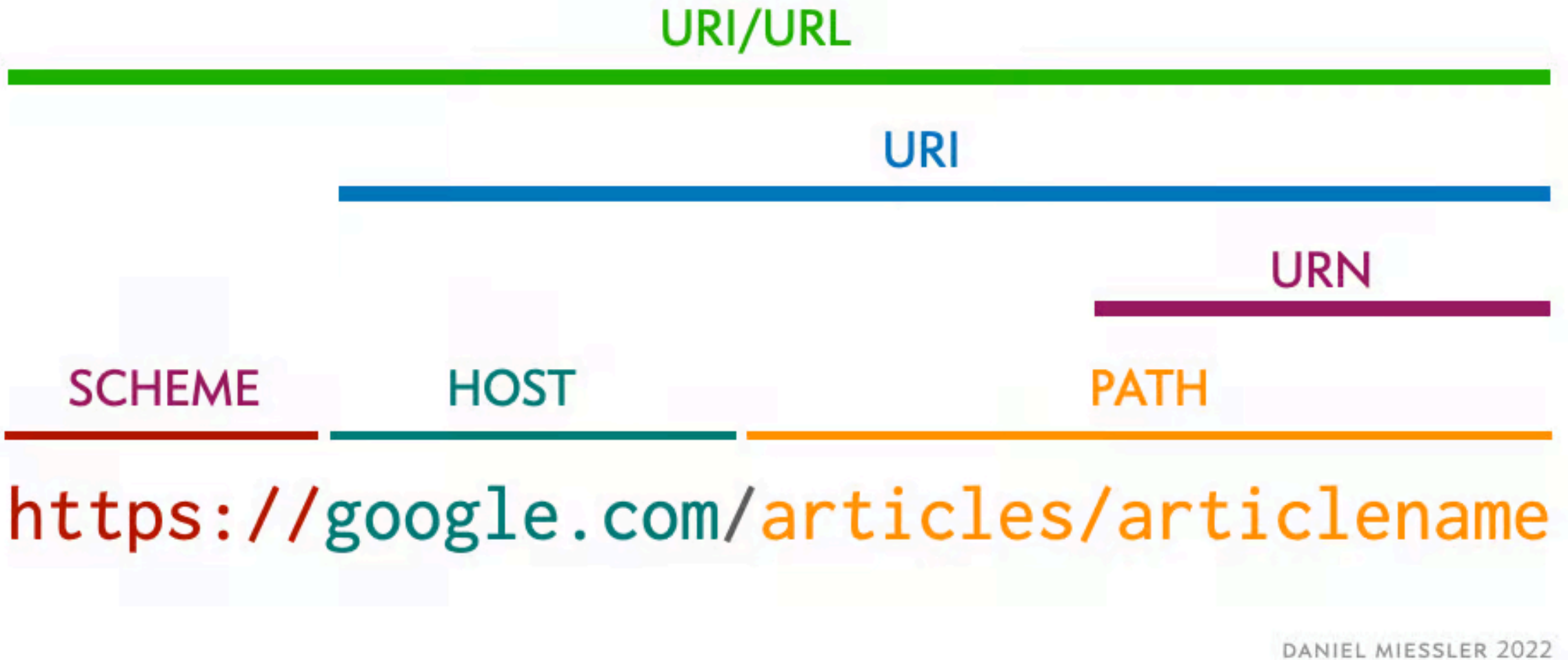
É importante destacar que o recurso representado não precisa ser necessariamente um mapeamento direto para uma tabela ou entidade da aplicação.

Por exemplo, um recurso **Agenda** pode ser mapeado para retornar os próximos compromissos de um usuário, porém os compromissos podem ser uma abstração de uma entidade **Aulas**, que representa as aulas de um semestre.



URL vs URI

- URI: Identifica um recurso de maneira única (ex.: /jogador)
- URL: Descreve como um recurso pode ser localizado (ex.: https://venson.net.br/jogador)


URI/URL
URI
URN
SCHEME
HOST
PATH
https://google.com/articles/articlename
DANIEL MIESSLER 2022

Verbos HTTP

Verbos HTTP são utilizados para adicionar semântica à uma requisição HTTP. Esses verbos auxiliam o servidor na identificação da natureza de uma requisição.

Verbo	Request Body	Response Body	Safe	Descrição
GET	Opcional	Sim	Sim	Recupera um recurso
POST	Sim	Sim	Não	Processa recurso
PUT	Sim	Sim	Não	Substitui recurso
PATCH	Sim	Sim	Não	Altera recurso
DELETE	Não	Sim	Não	Remove recurso

Verbo	Request Body	Response Body	Safe	Descrição
HEAD	Não	Não	Sim	Apenas cabeçalhos
CONNECT	Opcional	Sim	Não	Estabelece conexão
OPTIONS	Opcional	Sim	Sim	Opções de comunicacão
TRACE	Opcional	Sim	Não	Ping

> Mais informações em RFC 9110

Códigos de Estado HTTP

Códigos de Estado HTTP representam uma abstração do resultado de uma resposta. Os códigos são divididos em números de 100 a 599:

- Códigos informativos (100-199);
- Códigos sucesso (200-299);
- Códigos redirecionamento (300-399);
- Códigos erros de cliente (400-499);
- Códigos erros de servidor (500-599);

> Mais informações em Mozilla ou HTTP Cat

Padronização de Mensagens

RESTful APIs são construídas para fornecerem serviços à outras aplicações. Por isso é essencial que mensagens de sucesso e de erro sejam padronizadas. A estrutura do formato das mensagens deve ser sempre o mesmo, ainda que uma API possa responder com diferentes formatos. Os mais populares são:

- JSON
- XML
- YAML

HATEOS

- Links de hipermídia para navegação
- Interação dinâmica permite que clientes descubram os caminhos da aplicação
- Acoplamento reduzido, permitindo modificações sem quebrar a navegação

Mapeamento REST/HTTP

Para mapearmos a implementação de um CRUD (**C**reate, **R**ead, **U**pdate, **D**elete) usando a arquitetura REST, iremos utilizar os métodos descritos pelo protocolo HTTP como referência.

Para isso, usaremos:

- `GET`, para consultar resources
- `POST`, para adicionar resources
- `PUT`, para atualizar resources
- `DELETE`, para deletar resources

Adicionalmente, podemos utilizar:

- `PATCH`, para atualizar partes de um resource
- `PUT`, para adicionar resources

GET

O método `GET` tem a finalidade de retornar dados em nossa API. Podemos consultar um conjunto de dados (por exemplo, uma collection). Não é necessário passar nada no corpo da mensagem.

```
GET /usuarios

//RETORNO
[
    { "_id": 1, "login": "prezi", "senha": "abc123"},
    { "_id": 2, "login": "delta", "senha": "!@#$DFSER"},
    { "_id": 3, "login": "alfa", "senha": "USER123$"},
]
```

Ou um único recurso:

```
//REQUISIÇÃO
GET /usuarios/3 // usuarios/:id

//RETORNO
{ "_id": 3, "login": "alfa", "senha": "USER123$"}
```

POST

O método `POST` pode ser utilizado para inserir novos objetos em uma coleção. É necessário passar no corpo da mensagem o objeto completo a ser inserido. O retorno pode ser o próprio objeto enviado.

```
//REQUISIÇÃO
POST /usuarios

//BODY
````json
{ "login": "beta", "senha": "betinho123"}
```
```

PUT

O método `PUT` será utilizado para atualizar um recurso ou coleção. É necessário passar no corpo da mensagem um objeto completo a ser atualizado. O retorno, em caso de sucesso, pode ser o próprio objeto enviado.

```
//REQUISIÇÃO
PUT /usuarios/3

//BODY
{ "login": "diddy", "senha": "QWERTY"}
```

Adicionalmente, o método PUT também pode ser utilizado para inserir novos objetos, dado que caso o usuário de id `3` não exista, ele será criado.

DELETE

O método `DELETE` é utilizado para deletar recursos. Não é necessário passar nada no corpo. O retorno pode ser, em caso de sucesso, o próprio objeto deletado.

```
//REQUSIÇÃO
DELETE /usuarios/5000
```

PATCH

O método PATCH é utilizado para realizar alterações em um recurso, geralmente em parte dele. O retorno geralmente é o recurso atualizado.

```
//REQUISIÇÃO
PATCH /usuarios/5000

//BODY
{ "senha": "CASACO1992" }
```

Boas Práticas

Por boa prática, utilizamos algumas regras para declarar resources:

- Utilize substantivos para representar recursos e verbos para representar ações:
  - *Document*: representa um objeto singular

```
http://localhost/gerenciamento
http://localhost/admin
```

  - *Collection*: representa uma coleção de objetos gerenciada pelo servidor

```
http://localhost/usuarios
http://localhost/labs/07/computadores
```

- ○ *Store*: representa uma coleção de objetos gerenciada pelo cliente

```
http://localhost/usuario/200/carrinho
http://localhost/usuario/200/playlist
```

- ○ *Controller*: representa uma função adicional aplicada aos dados

```
http://localhost/hotel/200/check-in
http://localhost/temporada/01/start
```

- Use a barra `/` para indicar relações de hierarquia

```
http://localhost/bancos/contas/400A
```

- Não use barras `/` ao final de um URI

```
http://localhost/usuarios/
http://localhost/usuarios # Melhor
```

- Use hífens `-` para melhorar a leitura de um URI

```
http://localhost/carrosDeAluguel
http://localhost/carros-de-aluguel # Melhor
```

- Evite o uso de traço baixo `_`

```
http://localhost/carros_de_aluguel
http://localhost/carros-de-aluguel # Melhor
```

- Use letras minúsculas

```
http://localhost/Usuarios
http://localhost/usuarios # Melhor
```

- Não adicione extensão de arquivo

```
http://localhost/usuarios.json
http://localhost/usuarios # Melhor
```

- Use Query String para filtrar coleções

```
http://localhost/paises?continente=America
http://localhost/paises?continente=Africa&limite=5
```

- Nunca use o nome das funções CRUD na URI

```
http://localhost/usuarios/GET
http://localhost/usuarios/Adicionar
```

Estes URI podem ser utilizados para manipular os resources. Para isso, utilizamos os `Resource Methods`, métodos associados ao protocolo HTTP que realizam operações sobre os dados da aplicação.

O que aprendemos hoje

- O que é o modelo REST;
- Conceitos básicos do modelo
  - Recursos
  - URIs
  - Códigos HTTP
  - Verbos HTTP
- Boas práticas para implementar o modelo REST.