

Teste Quero Passagem

Especificar o Uso de scrapy

Especificar o Uso de splash

Especificar que haviam formas mais eficientes de encontrar os ids das cidades doq a forma feita por nosotros :) (utilizando o span que é preenchido com o id)

Processo mental 1 - Exploração:

- Primeiramente, após uma análise cuidadosa dos requests feitos pelo <https://www.viacaocometa.com.br> é fácil notar que:
 - O primeiro loading toma muito tempo e muitos recursos computacionais
 - Não há AJAX sendo executados na barra de pesquisa

com isso em mãos, vale a pergunta: e se nós simularmos uma primeira requisição?

Abaixo, a imagem mostrando todas as requisições feitas pelo cometa (vamos chamar assim daqui em diante) e o tempo >1000000 ms

Blocked response cookies Blocked requests 3rd-party requests						50000 ms	100000 ms	150000 ms	200000 ms	250000 ms	300000 ms	350000 ms	400000 ms	450000 ms	500000 ms	550000 ms	600000 ms	650000 ms
Name	Status	Type	Initiator	Size	Time													
icone_whats_min.png	200	png	gtm.js?id=GTM-WN6SS23:169	5.6 kB	48 ms													
styles_ltr.css	200	stylesheet	anchor?ar=1&k=m&co=aHR0cHM	42.7 kB	280 ms													
recaptcha_pt_br.js	200	script	anchor?ar=1&k=m&co=aHR0cHM	222 kB	470 ms													
26032402.js	200	script	granite.min.js:8	417 B	160 ms													
07ti=26032402&tm=gtm002&Ver=2&mid=e542280a-2579-40...	204	text/plain	bat.js:1	121 B	173 ms													
s032967411821247AQB=1&ndh=1&pf=1&t=22%2F9%2F2024...	200	gif	AppMeasurement.min.js:2	344 B	902 ms													
jcr:content.authorization.json?clear=1	200	xhr	granite.min.js:7	854 B	46 ms													
login	200	preflight	Preflight	0 B	127 ms													
login	201	xhr	granite.min.js:7	822 B	492 ms													
searchOrigin	200	preflight	Preflight	0 B	46 ms													
Loading.gif	200	gif	vendor.min.js:2	109 kB	67 ms													
searchOrigin	200	xhr	granite.min.js:7	70.3 kB	901 ms													
ibs:dpid=411&dpuuid=Zxgl-wAAKYqjgNz	200	gif	dd	713 B	31 ms													
icon.png	200	png	Other	983 B	24 ms													
data:image/gif;base...	200	gif	clientlib-site.min.css	(memory cache)	0 ms													
collectorPost	200	preflight	Preflight	0 B	565 ms													
collectorPost	200	preflight	Preflight	0 B	159 ms													
collectorPost	200	fetch	tealeaf.min.js:354	364 B	405 ms													
collectorPost	200	fetch	tealeaf.min.js:354	364 B	389 ms													
collectorPost	200	preflight	Preflight	0 B	164 ms													
collectorPost	200	fetch	tealeaf.min.js:354	364 B	158 ms													
login	200	preflight	Preflight	0 B	46 ms													
login	201	xhr	Other	822 B	646 ms													
collectorPost	200	preflight	Preflight	0 B	162 ms													
collectorPost	200	fetch	tealeaf.min.js:354	364 B	171 ms													
token.json	200	xhr	granite.min.js:7	613 B	115 ms													
collectorPost	200	preflight	Preflight	0 B	173 ms													
collectorPost	200	fetch	tealeaf.min.js:354	364 B	176 ms													
token.json	200	xhr	granite.min.js:7	617 B	94 ms													
collectorPost	200	preflight	Preflight	0 B	167 ms													
collectorPost	200	fetch	tealeaf.min.js:354	364 B	191 ms													

De primeira já podemos notar que houveram chamadas para uma api (<https://api.jcatlm.com.br/oauth/v3/login>) que acabou (após um google rapido) mostrando que são um grupo no ramo de logística. Legal!

A princípio, não iríamos ir muito mais a fundo, mas, quando clicamos sobre o input de pesquisa de passagens (partindo de) outra chamada é feita para essa mesma api, agora em outra rota (<https://api.jcatlm.com.br/place/v1/searchOrigin>) e, interessantemente, essa api nos retorna uma serie de informações da forma:

```
{
  "success": true,
  "result": [
    {
      "id": 22131,
      "city": "Aeroporto Galeão",
      "state": "RJ",
      "address": "",
      "latitude": null,
      "longitude": null,
      "operationsDay": null,
      "providerId": 1,
      "destinationProviderId": 0,
      "startTravelDate": null,
      "placeOrigin": null
    },
    {
      "id": 14234,
      "city": "Búzios",
      "state": "RJ",
      "address": "",
      "latitude": null,
      "longitude": null,
      "operationsDay": null,
      "providerId": 1,
      "destinationProviderId": 0,
      "startTravelDate": null,
      "placeOrigin": null
    },
    .
    .
    .
  ]
}
```

Interessante...

Agora, ainda acompanhando as requisições, fazemos a primeira pesquisa do desafio (passagens de SP - Tiete para Belo Horizonte);

Surpresa! Nosso url pesquisado é da forma:

https://www.viacaocometa.com.br/disponibilidade?data_ida=23102024&origem_id=18697&destino_id=5410&num_psgr=1&num_chda=0&num_chds=0&

[deep=false](#)

Ou seja, nos passamos origem_id/destino_id (exatamente iguais aqueles fornecidos pela rota que descobrimos antes), uma data no formato DDMMAAAA e outras informações (como o número de passageiros [num_psg], etc)

bacana demais, então, se nós buscamos agilizar nosso scraping, seria bom se conseguíssemos gerar esses url's diretamente pelo nosso programa (efetuando o login na api e buscando os id's diretamente);

Mas calma! Tem mais:

Quando olhamos as requisições feitas pela página de passagens nós vemos a mesma api na rota getRoutes! Isso chama atenção!

E, de fato, deveria chamar mesmo, por que nela nós encontramos todas as informações que estamos buscando:

```
{
  "success": true,
  "result": {
    "origin": {
      "id": 18697,
      "city": "SAO PAULO (TIETE) - SP",
      "state": "SP"
    },
    "destination": {
      "id": 5410,
      "city": "BELO HORIZONTE (RODOVIARIA) - MG",
      "state": "MG"
    },
    "date": "2024-10-22T00:00:00",
    "servicesList": [
      {
        "serviceId": "74985",
        "routeId": 1201,
        "brandId": 7,
        "group": "VIACA",
        "originId": 18697,
        "originDesc": "S\u00E3o Paulo (Rod. Tiet\u00EA)",
        "destinationId": 5410,
        "destinationDesc": "Belo Horizonte",
        "lineDate": "2024-10-22T00:00:00",
        "departureDate": "2024-10-22T20:45:00",
        "arrivalDate": "2024-10-23T06:15:00",
        "freeSeats": 14,
        "totalSeats": 46,
        "price": 179.16,
```

```
"priceWithDiscount": null,  
"discountValue": null,  
"class": "SEMILEITO PREMIUM",  
"classMatch": null,  
"company": "VIACAO COMETA S A",  
"companyId": 7,  
"connection": null,  
.  
.  
.
```

Lindo e sensacional! Mas, calma, claro que não vai ser tão simples assim, né?

Primeiro precisamos descobrir se alguma dessas rotas é publica! Se for, trabalho acabado, é buscar, filtrar o que precisamos, guardar em um banco de dados (pasta json) e enviar o desafio, se não, tem mais pela frente...

Bom... após testarmos a mesma requisição com o cURL nós deparamos com um 401 unauthorized;

Mas, não vamos nos abater por isso! Ainda tem muito html pela frente para ser analisado!

Processo mental 2 - Atenção aos pequenos detalhes

Já com poucas esperanças de me autenticar na API fora de um navegador padrão, estava buscando formas de extrair esses id's através de um *hover* sobre as cidades que caíam (dropdown) abaixo do input field utilizando selenium, já que, quando o *hover* era *triggered*, um `` era preenchido com, justamente, esse id! Valia então a tentativa!

Foi então, que eu vi uma *script tag* ao fim do html que constava:

```
<input type="hidden" name="clientId" id="clientId" value="52a68031-74aa-4635-9eb0-448f699d5be5">
```

Sensacional! Então temos o `clientId` disponível sem muito segredo no html da página!

Podemos então, seguir o caminho mais simples: Autenticação na API!

Processo mental 3 - Tentativa e erro

Após muito analisar e fazer diversas chamadas na api utilizando o cURL e o Postman, os detalhes começam a ficar mais claros:

- na requisição de login, dentre os *headers* enviados, consta "Client_id" e "Authorization"

o `client_id` já obtivemos! O site nós deu, mas o `authorization` não...

Muitos e muitos *refreshs* depois, encontramos a rota

<https://www.viacaocometa.com.br/content/jca/cometa/pt-br/jcr:content.authorization.json?clear=1>
nossa salvadora!

Nela, temos o retorno de um campo "authorizationId", exatamente o que precisavamos para fazermos o login dentro da api!

O retorno do login? um "access_token" utilizado para buscarmos coisas, como as rotas no getRoutes!

Perfeito!

Já sabemos (baseado na forma que a requisição é feita no navegador) que ela é da forma:

```
{"origin":18697,"destination":5410,"departureDate":"2024-10-24","availability":true}
```

Então é só logar, encontrar os id's solicitados e buscar fazer as buscas na API!

Podemos deixar de nos preocupar com o captcha (que o site principal indicou ter) e quaisquer outros freios no processo :)

Processo mental 4 - Codar

Documentando tudo que é plausível e buscando utilizar as técnicas de clean code, obtive os resultados [que estão no meu repositório](#)

Procurei uma arquitetura um pouco mais modularizada, então introduzi um ApiController que autentica e, efetivamente, maneja todas as operações com a API, enquanto que outras interfaces, como RequestGenerator, utilizam essa interface para gerar as requisições e, por fim, extrair os dados!

Por conta do tempo, não houve uma filtragem dos campos que serão utilizados.

Toda via, garanto que todos os campos pedidos estão lá!

Processo mental 5 - Esse é um teste de crawl né?

Após outra análise na forma com que as requisições são carregadas, percebi que não haveria muitas chances (em tempo hábil) de implementar uma solução em scrapy (apesar de conhecer as vantagens principalmente de desempenho).

Sendo assim, utilizei as ferramentas colocadas como sugestão no desafio, afinal por que não!

Então, utilizando o selenium implementei um crawler simples, porém versátil e simples para abstrair em uma classe mais geral (para outros tipos de sites similares).

Para a obtenção das poltronas ainda disponível, a API ainda é o melhor caminho, visto que para obtê-las através do selenium teríamos que fazer login (o que não seria um problema, mas o tempo está acabando).

Então, deixo aqui como implementaria esse restante do crawl-selenium:

primeiro, guardamos o index do serviço/passagem que estamos analisando no site; geramos um cpf fictício e uma conta fictícia (como a que deixei dentro do módulo crawl_from_website.py);

prosseguimos para contar os elementos class='seats' em que o texto interno é diferente de X (assentos livres para serem escolhidos). Com isso, já temos o número de assentos livres/ocupados!

Então, basta-nos voltar e continuar a partir do index que guardamos no início.

Poderia ter implementado esse final de forma "Porca", mas não valeria a pena já que aqui, meu código utilizando boas práticas que está sendo avaliado!

Conclusão

O teste cumpre os requisitos estabelecidos e, sem dúvida, o desafio da autenticação na api foi bastante interessante e fruto de muito aprendizado!

Como havia comentado nas reuniões anteriores, a análise cuidadosa do que está sendo extraído é tão importante quanto o processo do crawling em si! Principalmente quando analisamos que o tempo de crawl utilizando selenium é muito superior, utilizamos mais recursos computacionais e precisamos executar em uma máquina com capacidades gráficas.

Espero que seja o suficiente! Aguardo o retorno!