20/09

Soluções para a vulnerabilidade XSS (Cross-Site Scripting) - repositório `vulcom-xss-2025-2`

Solução 1: DOMPurify

1. **`DOMPurify`** é uma biblioteca disponível no ecossistema do Node.js. Sua documentação e instruções de uso podem ser acessadas em https://www.npmjs.com/package/dompurify. Como vamos usá-lo no *back-end*, precisamos também instalar a biblioteca **`jsdom`** que provê um DOM virtual para que **`DOMPurify`** possa funcionar. Para instalá-los, abra o terminal no repositório e execute:
   `npm install dompurify jsdom`
2. Em seguida, é necessário importar, configurar e usar o DOMPurify no arquivo index.js, conforme demonstrado neste Gist: https://gist.github.com/faustocintra/28cdf66f947444c43d72fdfd4d5a4d86

Solução 2: *Content Security Policy* (CSP)

1. Antes de testar a solução CSP, vamos reverter o arquivo index.js ao seu conteúdo original, para desativar o DOMPurify.
   Gist: https://gist.github.com/faustocintra/da179a96064d0ac83c15c8b898306a64
2. A solução CSP é implementada por meio de uma *meta tag* no arquivo HTML principal da aplicação. No nosso caso, para implementá-la, precisaremos alterar o arquivo **`views/comment.ejs`**. A configuração feita na *tag* **`meta`** no topo do arquivo faz com que *scripts* ***importados*** localmente ***localmente*** (ou seja, *scripts* remotos ou *inline* não serão executados).
   Gist: https://gist.github.com/faustocintra/6035941e7fba5499033b580e9963f492

Solução 3: configuração de *cookie* como `HTTPonly`

1. Para testar essa configuração, vamos desabilitar a tag meta do arquivo views/content.ejs, adicionada na solução anterior, para que ela não interfira nesta solução. Vamos transformá-la em um comentário:
   ```
   <!-- meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self'" -->
   ```
2. Voltamos a estar vulneráveis a XSS. Teste o seguinte comentário:
   ```
   <script>document.write('<h1>' + document.cookie + '</h1>');</script>
   ```
3. Para evitar que o *cookie* acessado por JavaScript, ficando exposto, devemos configurar o *cookie* como **`HTTPOnly = true`**, conforme demonstrado no
   Gist: https://gist.github.com/faustocintra/a5dbf024840e0198287b072e81044e9a

Solução 4: usando a *tag* de escape correta no EJS

1. No *engine* de *templates* EJS, a tag **<%- %>** exibe o conteúdo da variável sem sanitização, ocasionando a vulnerabilidade XSS. Nos casos em qu usamos esse *engine*, é mais seguro usar **<%= %>** para exibir variáveis, pois essa *tag* sanitiza o conteúdo antes de exibi-lo. Portanto, basta editar o arquivo **`views/comment.ejs`** e usar a *tag* segura.
   Gist: https://gist.github.com/faustocintra/7ef688a1df0b58b51b479f5993952177

04/10

1. Caso ainda não tenha feito, acesse github.com/faustocintra/vulcom-main-2025-2 e siga as instruções do README.
2. No VS Code, abra um terminal e use o botão  para duplicar o terminal.
   - No terminal da esquerda:
     ```
     cd back-end
     npm run dev
     ```
   - No terminal da direita:
     ```
     cd front-end
     npm run dev
     ```
3. Acesse o *front-end* em http://localhost:5173
4. Abra um terceiro terminal clicando sobre o botão  e execute os comandos:
   ```
   cd back-end
   npx prisma studio
   ```
5. Isso irá abrir o Prisma Studio no endereço http://localhost:5555. Essa ferramenta permite visualizar o banco de dados acessado pela aplicação *back-end*. Acesse a tabela **`Users`**. Repare que as senhas foram gravadas na tabela em texto claro, o que constitui uma vulnerabilidade gravíssima, já que, caso esses dados venham a vazar de alguma forma, um atacante terá acesso imediato às senhas dos usuários.
6. Por esse motivo, não se deve jamais armazenar senhas em texto claro no banco de dados. **O procedimento correto é armazenar apenas o *hash* da senha**, gerado por meio de um algoritmo comprovadamente seguro. Uma das melhores opções nesse sentido, atualmente, é a biblioteca **Bcrypt**. Para utilizá-la, precisamos instalá-la no *back-end*.
   - Interrompa a execução do *back-end* no terminal correspondente teclando **`Ctrl+C`**.
   - Instale a biblioteca:
     ```
     npm install bcrypt
     ```
   - Retome a execução do *back-end*:
     ```
     npm run dev
     ```

7. Para ulilizar a biblioteca **bcrypt** no projeto, precisamos importá-la no arquivo **back-end/src/controllers/users.js**:
   import bcrypt from 'bcrypt'
8. Em seguida, modificamos o método **create()** desse *controller* para passar a usar o **bcrypt** ao criar novos usuários:

```
controller.create = async function(req, res) {
  try {

    // Verifica se existe o campo "password" em "req.body".
    // Caso positivo, geramos o hash da senha antes de enviá-la
    // ao BD
    // (12 na chamada a bcrypt.hash() corresponde ao número de
    // passos de criptografia utilizados no processo)
    if(req.body.password) {
      req.body.password = await bcrypt.hash(req.body.password, 12)
    }

    await prisma.user.create({ data: req.body })

    // HTTP 201: Created
    res.status(201).end()
  }
  catch(error) {
    console.error(error)

    // HTTP 500: Internal Server Error
    res.status(500).end()
  }
}
```

9. O mesmo procedimento deve ser feito no método update(), para que eventuais alterações de senhas em usuários já existentes também sejam processadas.

```
controller.update = async function(req, res) {
```

```
  try {

    // Verifica se existe o campo "password" em "req.body".
    // Caso positivo, geramos o hash da senha antes de enviá-la
    // ao BD
    // (12 na chamada a bcrypt.hash() corresponde ao número de
    // passos de criptografia utilizados no processo)
    if(req.body.password) {
      req.body.password = await bcrypt.hash(req.body.password, 12)
    }

    const result = await prisma.user.update({
      where: { id: Number(req.params.id) },
      data: req.body
    })

    // Encontrou e atualizou ~> HTTP 204: No Content
    if(result) res.status(204).end()
    // Não encontrou (e não atualizou) ~> HTTP 404: Not Found
    else res.status(404).end()
  }
  catch(error) {
    console.error(error)

    // HTTP 500: Internal Server Error
    res.status(500).end()
  }
}
```

10. Para testar, instale a extensão **RapidAPI Client** no VS Code. Trata-se de um cliente HTTP, com funcionamento semelhante a ferramentas como Postman ou Insomnia. Será acrescentado um ícone semelhante a uma letra **R** na barra vertical esquerda do VS Code.
11. Clique sobre o ícone do RapidAPI Client para ativá-lo e, em seguida, sobre o botão **+** para criar uma nova requisição. Use os seguintes parâmetros nessa requisição:
    - Método: **PUT**
    - URL: **`http://localhost:8080/users/1`**
    - Corpo da requisição, na aba "Body/JSON":
      ```
      {
        "password": "Vulcom@DSM"
      }
      ```
    - Clique sobre o botão "Send". Se tudo der certo, você receberá uma resposta **`204 No Content`**.
12. Para conferir se o *hash* da senha foi realmente gerado e gravado no banco de dados, troque o método da requisição de **PUT** para **GET** e clique novamente sobre o botão "Send".
13. Repita requisições **PUT** para os demais usuários do seu banco de dados, de modo a gerar o *hash* para a senha "senha123" criada inicialmente para eles.
    - Repare que, mesmo que a senha dos quatro usuários seja a mesma, o *hash* gerado para cada usuário é diferente. Isso se deve ao fato de que o **`bcrypt`** usa sais diferentes para cada geração de *hash*. Mais detalhes sobre o funcionamento do **`bcrypt`** podem ser encontrados aqui.
14. Vamos terminar as atividades do dia com um *commit* + *push* no repositório do GitHub.
    - Feche todos os terminais abertos usando o ícone da lixeira 🗑 e abra um terminal novo. Nele, execute:
      ```
      git add .
      git commit -m "(04/10) Armazenamento de senhas de modo seguro no banco de dados"
      git push
      ```
    - **IMPORTANTE: o Trabalho 2 da disciplina será constituído pelo conjunto de *commits* no repositório do aluno, que deve ser um *fork* do repositório do professor. Serão considerados, para fins avaliativos, apenas os *commits* feitos entre o dia da aula e o dia anterior à aula seguinte. Por exemplo, o *commit* de uma aula de sábado deve ser feito, no máximo, até a terça-feira seguinte.**

08/10

1. No VS Code, abra um terminal e use o botão  para duplicar o terminal.
   a. No terminal da esquerda:
   `cd back-end`
   `npm run dev`
   b. No terminal da direita:
   `cd front-end`
   `npm run dev`
2. Acesse o *front-end* em http://localhost:5173 e tente fazer *login* com o usuário **`admin`** e a senha **`Vulcom@DSM`**. Não funcionará, pois encriptamos a senha armazenada no banco de dados, e agora precisamos descriptografá-la ao fazer a conferência da senha no processo de *login*. No entanto, se você tentar com **`admin`** e **`admin123`**, conseguirá entrar, uma vez que essa autenticação (com a senha **`admin123`**) está no próprio código, gerando uma vulnerabilidade de autenticação fixa.
3. Vamos alterar o código do método **`login()`** em **`back-end/src/controllers/users.js`** para remover as credenciais fixas e processar a descriptografia da senha. Com as alterações abaixo, a autenticação com **`admin`**/**`Vulcom@DSM`** deve voltar a funcionar (note a linha com **`bcrypt.compare()`**).

```
controller.login = async function(req, res) {
  try {

      // Busca o usuário no BD usando o valor dos campos
      // "username" OU "email"
      const user = await prisma.user.findFirst({
        where: {
          OR: [
            { username: req.body?.username },
            { email: req.body?.email }
          ]
        }
      })

      // Se o usuário não for encontrado, retorna
      // HTTP 401: Unauthorized
```

```
      if(! user) return res.status(401).end()

      // REMOVENDO VULNERABILIDADE DE AUTENTICAÇÃO FIXA
      // if(req.body?.username === 'admin' && req.body?.password === 'admin123')
passwordIsValid = true
      // else passwordIsValid = user.password === req.body?.password
      // passwordIsValid = user.password === req.body?.password

      // Chamando bcrypt.compare() para verificar se o hash da senha
      // enviada coincide com o hash da senha armazenada no BD
      const passwordIsValid = await bcrypt.compare(req.body?.password, user.password)

      // Se a senha estiver errada, retorna
      // HTTP 401: Unauthorized
      if(! passwordIsValid) return res.status(401).end()

      // Usuário e senha OK, passamos ao procedimento de gerar o token
      const token = jwt.sign(
        user,                          // Dados do usuário
        process.env.TOKEN_SECRET,      // Senha para criptografar o token
        { expiresIn: '24h' }           // Prazo de validade do token
      )

      // Formamos o cookie para enviar ao front-end
      res.cookie(process.env.AUTH_COOKIE_NAME, token, {
        httpOnly: true, // O cookie ficará inacessível para o JS no front-end
        secure: true,   // O cookie será criptografado em conexões https
        sameSite: 'None',
        path: '/',
        maxAge: 24 * 60 * 60 * 100  // 24h
      })

      // Retorna o token e o usuário autenticado com
```

```
      // HTTP 200: OK (implícito)
      res.send({token, user})

  }
  catch(error) {
    console.error(error)

    // HTTP 500: Internal Server Error
    res.status(500).end()
  }
}
```

4. Temos ainda uma última questão relacionada à segurança de senhas para resolver. Crie uma nova requisição no RapidAPI Client com as seguintes características:
    a. Método: **`GET`**
    b. URL: **`http://localhost:8080/users`**
    c. Envie clicando sobre o botão "Send".
5. No resultado, você deverá observar que o campo **`password`** está sendo retornado. Embora esteja criptografado, ele não deveria ficar disponível para que um eventual atacante pudesse tentar descobrir senhas usando o método da força bruta. Por isso, o melhor que fazemos é ***OMITIR*** esse campo no retorno dos métodos **`retrieveAll()`** e **`retrieveOne()`** do *controller* de usuários.
6. No arquivo **`back-end/src/controllers/users.js`**:
    a. No método **`retrieveAll()`**:

```
controller.retrieveAll = async function(req, res) {
  try {
    const result = await prisma.user.findMany(
      // Omite o campo "password" do resultado
      // por questão de segurança
      { omit: { password: true } }
    )
// (...código existente...)
```

    b. No método **`retrieveOne()`**:

```
controller.retrieveOne = async function(req, res) {
```

```
  try {
    const result = await prisma.user.findUnique({
      // Omite o campo "password" do resultado
      // por questão de segurança
      omit: { password: true },
      where: { id: Number(req.params.id) }
    })
// (...código existente...)
```

c. Repita a requisição **GET** nas URLs http://localhost:8080/users e http://localhost:8080/users/1. Em ambos os casos, o campo **password** não deve mais aparecer.

7. No método **login()**, não podemos omitir o campo password da consulta, pois precisamos dele para fazer a conferência da senha. No entanto, não devemos incluí-lo entre as informações do usuários que formam o *token*, por razões de segurança. Por isso, devemos excluir o campo **password** do JSON antes de formar o *token*.

   a. No arquivo **back-end/src/controllers/users.js**, método **login()**:

```
// (...código existente...)
      // Chamando bcrypt.compare() para verificar se o hash da senha
      // enviada coincide com o hash da senha armazenada no BD
      const passwordIsValid = await bcrypt.compare(req.body?.password, user.password)

      // Se a senha estiver errada, retorna
      // HTTP 401: Unauthorized
      if(! passwordIsValid) return res.status(401).end()

      // Por motivos de segurança, exclui o campo "password" dos dados do usuário
      // para que ele não seja incluído no token
      if(user.password) delete user.password

      // Usuário e senha OK, passamos ao procedimento de gerar o token
      const token = jwt.sign(
        user,                       // Dados do usuário
        process.env.TOKEN_SECRET,   // Senha para criptografar o token
        { expiresIn: '24h' }        // Prazo de validade do token
```

```
    )
// (...código existente...)
```

8. Terminada a segurança de senhas, vamos nos concentrar na segurança da API do back-end, que atualmente encontra-se totalmente aberta. É possível fazer qualquer requisição ao *back-end* sem apresentar o *token* de autenticação. Experimente, por exemplo, fazer uma requisição **GET** em http://localhost:8080/cars ou http://localhost:8080/customers.
9. Dentro do nosso *back-end*, a sequência desde a recepção da requisição até o envio da resposta passa pelos seguintes arquivos, em ordem:
    a. **src/bin/server.js**
    b. **src/app.js**
    c. um arquivo de rota da pasta **src/routes**, de acordo com a rota da requisição detectada em **src/app.js**
    d. um arquivo de *controller* da pasta **src/controllers**, de acordo com o arquivo de *routes*
10. A segurança da nossa API será implementada por meio do *token* JWT, que já está sendo gerada pelo processo de *login* e enviado ao *front-end*. Agora, o *front-end* deverá apresentar o *token* a cada nova requisição, sendo que o *back-end* somente deverá permitir a conclusão bem-sucedida da requisição se o *token* estiver presente e válido.
11. A ideia é ***interceptar*** o fluxo da requisição, conforme especificado no item 9, e fazer uma verificação entre as etapas b) e c). Para isso, vamos encaixar uma nova etapa no fluxo, um *middleware*, cuja função será exatamente fazer a conferência do *token*.
12. Crie uma nova pasta chamada **middleware** dentro de **back-end/src**. Dentro da nova pasta, crie um arquivo chamado **auth.js** com o conteúdo de https://gist.github.com/faustocintra/d4abeac71382ad0615866ccfc7110b7f
13. No arquivo **back-end/src/app.js**, vamos inserir uma chamada ao *middleware* de autenticação ***ANTES*** de qualquer outra rota ser processada:

```
// (...código existente...)
/*********** ROTAS DA API **************/

// Middleware de verificação do token de autorização
import auth from './middleware/auth.js'
app.use(auth)
// (...código existente...)
```

14. Tente agora repetir as requisições sugeridas no item 8. Você deve receber uma resposta **403 Forbidden**. No entanto, pela interface *front-end* do Projeto Vulcom, é possível acessar normalmente as listagens de clientes e veículos a partir do menu da aplicação, uma vez que a lógica de envio do *token* de volta ao *back-end* já estava implementada.

15. No entanto, para fazer o mesmo por meio de uma requisição no RapidAPI Client, o processo é mais longo.
    a. Primeiramente, é necessário obter um *token* de autenticação. Para isso, crie uma nova requisição com as seguintes características:
        i. Método: **`POST`**
        ii. URL: http://localhost:8080/users/login
        iii. Conteúdo da aba "Body/JSON":
        ```
        { "username": "admin", "password": "Vulcom@DSM" }
        ```
        iv. Clique sobre o botão "Send"
        v. Copie o *token* da resposta. Tenha cuidado para copiá-lo ***INTEIRO*** e ***SEM AS ASPAS***.
    b. Crie uma segunda requisição no RapidAPI Client, com as seguintes características:
        i. Método: **`GET`**
        ii. URL: http://localhost:8080/customers
        iii. Conteúdo da aba "Auth/Bearer": **o *token* copiado anteriormente**
        iv. Clique sobre o botão "Send"
        v. Agora, os dados dos clientes deverão ser listados normalmente, pois estamos enviando um *token* válido no cabeçalho da requisição.
16. Vamos terminar as atividades do dia com um *commit* + *push* no repositório do GitHub.
    a. Feche todos os terminais abertos usando o ícone da lixeira e abra um terminal novo. Nele, execute:
    ```
    git add .
    git commit -m "(08/10) Término da segurança de senhas + segurança de API com token JWT"
    git push
    ```
    b. IMPORTANTE: o Trabalho 2 da disciplina será constituído pelo conjunto de *commits* no repositório do aluno, que deve ser um *fork* do repositório do professor. Serão considerados, para fins avaliativos, apenas os *commits* feitos entre o dia da aula e o dia anterior à aula seguinte. Por exemplo, o *commit* de uma aula de sábado deve ser feito, no máximo, até a terça-feira seguinte.

11/10

1. No VS Code, abra um terminal e use o botão para duplicar o terminal.
    a. No terminal da esquerda:
    ```
    cd back-end
    ```

```
npm run dev
```

b. No terminal da direita:

```
cd front-end
npm run dev
```

2. Acesse o *front-end* em http://localhost:5173.
3. Com a configuração do *middleware* de autenticação no *back-end*, feita na aula anterior, todas as nossas rotas (com exceção de **`/users/login`**, que é utilizada para a própria autenticação) só podem ser acessadas com a apresentação de um *token* JWT válido. Agora, precisamos voltar nossa atenção para o *front-end* e verificar como esse *token* é recebido ao final do processo de autenticação e como o *front-end* o armazena.
4. Abra o arquivo **`front-end/src/pages/Login.jsx`** e observe o método **`handleSubmit()`**. Podemos observar que, nas linhas 69 a 72, o *token* JWT, enviado como resposta da requisição, é armazenado no *local storage*.
    a. Faça *login* na aplicação Vulcom. Em seguida, abra as Ferramentas de Desenvolvedor (botão direito do *mouse* > Inspecionar... > aba "Application" > "Local Storage" no painel esquerdo > http://localhost:5173 > chave **`_auth`**) e você encontrará o *token* de autenticação.
5. Embora prático e fácil de usar, o *local storage* não é o local mais seguro para armazenar o *token* de autenticação. Isso porque ele é totalmente acessível por qualquer código JavaScript que execute no *front-end*, como vamos demonstrar.
6. Substitua o código do arquivo **`front-end/src/pages/Homepage.jsx`** pelo código abaixo:

```
import React from 'react'
import Typography from '@mui/material/Typography'

export default function Homepage() {
  const [ls, setLs] = React.useState([])

  React.useEffect(() => {
    const _ls = []
    for(let i = 0; i < window.localStorage.length; i++) {
      const key = window.localStorage.key(i)
      const value = window.localStorage.getItem(key)
      _ls.push({ [key]: value })
    }
    console.log(_ls)
    setLs(_ls)
```

```
  }, [])

  return(
    <>
      <Typography variant="h1" gutterBottom>
        Projeto VULCOM
      </Typography>

      <Typography>
        Sistema para análise e estudo de vulnerabilidades comuns
      </Typography>

      <Typography variant="h6">
        Exposição de valores do <em>local storage</em>
        <Typography variant="caption" style={{ fontFamily: 'monospace '}}>
          {
            ls.map(kv => (
              <p>{Object.keys(kv)[0]} =&gt; {kv[Object.keys(kv)[0]]}</p>
            ))
          }
        </Typography>
      </Typography>

    </>
  )
}
```

7. Ao acessar a *home page* do projeto, todo o conteúdo armazenado no *local storage* será listado. Isso é um problema, pois, no caso de a aplicação sofrer um ataque de XSS (*cross-site scripting*), o *script* externo terá o mesmo acesso amplo ao *local storage*.
8. Por isso, precisamos encontrar uma forma mais segura de armazenar o *token* no *front-end*. E o método recomendável é o uso de um *cookie* seguro e *HTTP-only*.
    a. *cookie* seguro: é aquele que só é transmitido por meio de conexões cripografadas (protocolo **`https://`**). A exceção, para efeitos de teste, é se o servidor for **`localhost`** ou **`127.0.0.1`**, quando se permite que o *cookie* seguro circule em conexões **`http://`** simples.

b. *cookie HTTP-only*: é aquele acessível pelo protocolo HTTP, mas não pelo JavaScript. Com isso, *scripts* maliciosos não conseguem capturá-lo.

9. Para implementar *cookies* seguros e *HTTP-only*, precisamos modificar ambos os projetos.
   a. No projeto *front-end*: abra o arquivo **`front-end/src/lib/myfetch.js`**. Esse é um arquivo que configura todas as requisições enviadas ao *back-end* por meio da API fetch() do JavaScript realizadas pela aplicação. **Descomente a linha 27**. Com isso, o *front-end* estará solicitando ao *back-end* para que envie *cookies* como resposta à requisição.

```
// (...código existente...)
function getOptions(body = null, method = 'GET') {
  const options = {
    method,
    headers: {"Content-type": "application/json; charset=UTF-8"},
    credentials: 'include'  // Instrui o back-end a gravar cookies no front
  }
// (...código existente...)
```

   b. No projeto *back-end*: abra o arquivo **`back-end/src/controllers/users.js`** e vá até o método

```
controller.login = async function (req, res) {
  try {
    // Busca o usuário no BD usando o valor dos campos
    // "username" OU "email"
    const user = await prisma.user.findFirst({
      where: {
        OR: [
          { username: req.body?.username },
          { email: req.body?.email }
        ]
      }
    })

    // Se o usuário não for encontrado, retorna
    // HTTP 401: Unauthorized
    if(! user) {
```

```
  console.error('ERRO DE LOGIN: usuário não encontrado')
  return res.status(401).end()
}

// Usuário encontrado, vamos conferir a senha
const passwordIsValid = await bcrypt.compare(req.body?.password, user.password)

// Se a senha estiver errada, retorna
// HTTP 401: Unauthorized
if(! passwordIsValid) {
  console.error('ERRO DE LOGIN: senha inválida')
  return res.status(401).end()
}

// Deleta o campo "password" do objeto "user" antes de usá-lo
// no token e no valor de retorno
if(user.password) delete user.password

// Usuário/email e senha OK, passamos ao procedimento de gerar o token
const token = jwt.sign(
  user,                         // Dados do usuário
  process.env.TOKEN_SECRET,     // Senha para criptografar o token
  { expiresIn: '24h' }          // Prazo de validade do token
)

// Formamos o cookie para enviar ao front-end
res.cookie(process.env.AUTH_COOKIE_NAME, token, {
  httpOnly: true,     // Torna o cookie inacessível para JavaScript
  secure: true,       // O cookie só trafegará em HTTPS ou localhost
  sameSite: 'None',
```

```
        path: '/',
        maxAge: 24 * 60 * 60 * 1000   // 24h
      })

      // Retorna o token e o usuário autenticado, com o status
      // HTTP 200: OK (implícito)
      res.send({ user, token })
    }
    catch(error) {
      // Se algo de errado acontecer, cairemos aqui
      // Nesse caso, vamos exibir o erro no console e enviar
      // o código HTTP correspondente a erro do servidor
      // HTTP 500: Internal Server Error
      console.error(error)
      res.status(500).end()
    }
  }
```

c. Observe que, entre as linhas 170 e 176, o *cookie* já está sendo formado, com as características desejadas (**`httpOnly: true`** e **`secure: true`**). Esse *cookie* será enviado automaticamente ao *front-end* juntamente com a resposta à requisição. Aqu tudo já está OK, então nenhuma modificação será necessária.

d. Ainda no projeto *back-end* abra o arquivo **`back-end/src/app.js`** e descomente a linha 14, para permitir a circulação do *cookie* entre *back-end* e *front-end*:

```
// (...código existente...)
app.use(cors({
  origin: process.env.ALLOWED_ORIGINS.split(','),
  credentials: true
}))
// (...código existente...)
```

10. Faça *logoff* e *login* novamente na aplicação Vulcom. Em seguida, acesse as Ferramentas de Desenvolvedor, aba "Application", "Cookies" e "http://localhost:5173". Você deverá ver o *cookie* com o *token*, com um tique marcando as colunas "HTTPOnly" e "Secure'.

11. Vamos retornar ao arquivo **`front-end/src/pages/Homepage.jsx`** e tentar mostrar os *cookies* via JavaScript:

```
// (...código existente...)
      <Typography variant="h6">
        Exposição de valores do <em>local storage</em>
        <Typography variant="caption" style={{ fontFamily: 'monospace '}}>
          {
            ls.map(kv => (
              <p>{Object.keys(kv)[0]} =&gt; {kv[Object.keys(kv)[0]]}</p>
            ))
          }
        </Typography>
      </Typography>

      <Typography variant="h6">
        Exposição de <em>cookies</em>
        <Typography variant="caption" style={{ fontFamily: 'monospace '}}>
          <p>{ document.cookie }</p>
        </Typography>
      </Typography>
// (...código existente...)
```

12. O código em azul será capaz de mostrar apenas os *cookies* ***não marcados*** como HTTP-only. Ou seja, nosso *token* ***NÃO*** deverá ser listado.
13. Para ilustar melhor a diferença, vamos retornar ao arquivo **`back-end/src/controllers/users.js`** e enviar também um *cookie* não *HTTP-only* junto com a requisição de *login*.

```
// (...código existente...)
      // Formamos o cookie para enviar ao front-end
      res.cookie(process.env.AUTH_COOKIE_NAME, token, {
        httpOnly: true, // O cookie ficará inacessível para o JS no front-end
        secure: true,   // O cookie será criptografado em conexões https
        sameSite: 'None',
        path: '/',
        maxAge: 24 * 60 * 60 * 100  // 24h
```

```
      })

      // Cookie não HTTP-only, acessível via JS no front-end
      res.cookie('not-http-only', 'Este-cookie-NAO-eh-HTTP-Only', {
        httpOnly: false,
        secure: true,   // O cookie será criptografado em conexões https
        sameSite: 'None',
        path: '/',
        maxAge: 24 * 60 * 60 * 100  // 24h
      })
// (...código existente...)
```

14. Faça *logoff* e *logon* novamente. Ao acessar as Ferramentas de Desenvolvedor, você vai notar agora a presença do *cookie* **`not-http-only`**, que também será exposto na *home page*.
15. Com isso, terminamos a primeira parte da troca do armazenamento do *token* do *local storage* para *cookie HTTP-only*. O que ainda falta fazer:
    a. Alterar o *middleware* de autenticação no *back-end* para procurar o *token* de autenticação primeiramente em *cookies*, e só depois no cabeçalho "Authentication".
    b. Alterar o resposta do método de *login*, para deixar de enviar o *token* ali (o *token* será enviado exclusivamente via *cookie*).
    c. Alterar alguns processos no *front-end* que supõem que o *token* está armazenado no *local storage*.
16. Vamos terminar as atividades do dia com um *commit* + *push* no repositório do GitHub.
    a. Feche todos os terminais abertos usando o ícone da lixeira  e abra um terminal novo. Nele, execute:

```
git add .
git commit -m "(11/10) Início da transferência do armazenamento do token JWT para cookie"
git push
```

    b. **IMPORTANTE: o Trabalho 2 da disciplina será constituído pelo conjunto de *commits* no repositório do aluno, que deve ser um *fork* do repositório do professor. Serão considerados, para fins avaliativos, apenas os *commits* feitos entre o dia da aula e o dia anterior à aula seguinte. Por exemplo, o *commit* de uma aula de sábado deve ser feito, no máximo, até a terça-feira seguinte.**

18/10

1. No VS Code, abra um terminal e use o botão  para duplicar o terminal.
    a. No terminal da esquerda:
    `cd back-end`
    `npm run dev`
    b. No terminal da direita:
    `cd front-end`
    `npm run dev`
2. Acesse o *front-end* em http://localhost:5173.
3. Como dito ao final da última aula, o *middleware* de autenticação do *back-end* ainda procura o *token* de autorização no cabeçalho "Authentication" da requisição (e não nos *cookies*), de modo que a página de *login* do *front-end* parou de funcionar. Vamos modificar o *middleware*, fazendo-o procurar pelo *token* de autorização primeiramente nos *cookies* da requisição, e só então no cabeçalho "Authentication". Dessa forma, o *login* do *front-end* volta a funcionar (enviando o *token* via *cookie*) e não perderemos a capacidade de fazer requisições por meio de clientes HTTP como o RapidAPI, passando o *token* pela aba "Auth/Bearer".
    a. No arquivo **`back-end/src/middleware/auth.js`** altere o trecho destacado em azul:

```
/*
  Este middleware intercepta todas as rotas e verifica
  se um token de autorização foi enviado junto com a
  requisição
*/
import jwt from 'jsonwebtoken'

/*
  Algumas rotas, como POST /users/login, poderão ser
  acessadas sem a necessidade de apresentação do token.
  Cadastramos essas rotas no vetor bypassRoutes.
*/
const bypassRoutes = [
  { url: '/users/login', method: 'POST' },
  // Caso o cadastro de novos usuários seja público
  // { url: '/users', method: 'POST' }
```

```
]

// Função do middleware
export default function(req, res, next) {
  /*
    Verificamos se a rota interceptada corresponde a
    alguma daquelas cadastradas em bypassRoutes. Sendo
    o caso, permite continuar para o próximo middleware
    sem a verificação do token de autorização
  */
  for(let route of bypassRoutes) {
    if(route.url === req.url && route.method == req.method) {
      next()    // Continua para o próximo middleware
      return    // Encerra este middleware
    }
  }

  /* PROCESSO DE VERIFICAÇÃO DO TOKEN DE AUTORIZAÇÃO */
  let token

  // Primeiramente, procura pelo token de autorização em um cookie
  token = req.cookies[process.env.AUTH_COOKIE_NAME]

  if(! token) {
    // Se não tiver sido encontrado o token no cookie,
    // procura pelo token no cabeçalho de autorização
    const authHeader = req.headers['authorization']

    console.log({authHeader})

    // Se o cabeçalho 'authorization' não existir, retorna
    // HTTP 403: Forbidden
    if(! authHeader) {
```

```
    console.error('ERRO DE AUTORIZAÇÃO: falta de cabeçalho')
    return res.status(403).end()
  }

  /*
    O cabeçalho 'autorization' tem o formato "Bearer XXXXXXXXXXXXXXX",
    onde "XXXXXXXXXXXXXXX" é o token. Portanto, precisamos dividir esse
    cabeçalho (string) em duas partes, cortando onde está o caractere de
    espaço e aproveitando apenas a segunda parte (índice 1)
  */
  token = authHeader.split(' ')[1]
}

// Verificação de integridade e validade do token
jwt.verify(token, process.env.TOKEN_SECRET, (error, user) => {

  // Token inválido ou expirado, retorna
  // HTTP 403: Forbidden
  if(error) {
    console.error('ERRO DE AUTORIZAÇÃO: token inválido ou expirado')
    return res.status(403).end()
  }

  /*
    Se chegamos até aqui, o token está OK e temos as informações do
    usuário autenticado no parâmetro "user". Vamos guardá-lo dentro
    do objeto "req" para respoder ao front-end sempre que ele perguntar
    qual usuário está atualmente autenticado
  */
  req.authUser = user

  // Token verificado e validado, passamos ao próximo middleware
  next()
```

```
  })
}
```

b. Acesse o *front-end* e faça *login*. Deve ter voltado a funcionar.

4. Ao final do método **`login()`** do *controller* de usuários do *back-end*, retornamos como resposta o *token* e o usuário autenticado. Não é mais necessário retornar o *token* na resposta da requisição, já que isso está sendo feito via *cookie* HTTP seguro, e ainda aumenta a segurança por não expor o *token* ao acesso direito do *front-end*. Portanto, vamos modificar o final do método **`login()`** para remover o retorno do *token* na resposta da requisição.
   a. No arquivo **`back-end/src/controllers/users.js`**, ao final do método **`login()`**, altere as linhas destacadas em azul:

```
// (...código existente...)
      // Retorna APENAS o usuário autenticado com
      // HTTP 200: OK (implícito)
      res.send({user})
// (...código existente...)
```

5. A mudança do armazenamento do *token* de autorização no *front-end*, do *local storage* para um *cookie* HTTP seguro, afeta também a forma como o *front-end* faz *logout*. Vejamos como é feito, ainda supondo que o *token* esteja no *local storage*:
   a. Arquivo **`front-end/src/ui/AuthControl.jsx`**:

```
// (...código existente...)
  async function handleLogoutButtonClick() {
    if(await askForConfirmation('Deseja realmente sair?')) {
      // Apaga o token do localStorage
      window.localStorage.removeItem(import.meta.env.VITE_AUTH_TOKEN_NAME)

      // Remove as informações do usuário autenticado
      setAuthUser(null)

      // Redireciona para a página de login
      navigate('/login')
    }
  }
// (...código existente...)
```

   b. Ou seja, o procedimento de *logout* é extremamente simples: apagar o *token* do *local storage*, limpar o usuário autenticado da memória e redirecionar para a página de *login*. Um processo feito 100% pelo *front-end*.
6. Agora que o *token* de autorização está armazenado em um *cookie* HTTP seguro, o *front-end* não tem mais acesso a ele. Como consequência, o *front-end* não conseguirá mais lidar com a rotina de *logoff* sozinho. O processo de *logoff*, de agora em diante, será constituído pelas seguintes etapas:
   a. o *front-end* enviará uma requisição de *logout* para o *back-end*. Ao processar essa requisição, o *back-end* apagará o *cookie* com o *token* enviado ao *front-end*.
   b. Ao receber uma resposta de sucesso (**`HTTP 204: No Content`**) da requisição enviada, aí sim o *front-end* limpa as informações do usuário autenticado da memória e redireciona para a página de *login*.
7. Vamos acrescentar um método **`logout()`** ao final do *controller* de usuários do *back-end*:
   a. No final do arquivo **`back-end/src/controllers/users.js`**, acrescente as linhas em azul:

```
// (...código existente...)
controller.logout = function(req, res) {
  // Apaga no front-end o cookie que armazena o token
  res.clearCookie(process.env.AUTH_COOKIE_NAME)
  // HTTP 204: No Content
  res.status(204).end()
}

export default controller
```

8. Reorganize o arquivo **`back-end/src/routes/users.js`**. A linha referente ao *logout* foi acrescentada, e outras rotas mudaram de ordem.

```
import { Router } from 'express'
import controller from '../controllers/users.js'

const router = Router()

router.post('/login', controller.login)
router.post('/logout', controller.logout)
router.get('/me', controller.me)

router.post('/', controller.create)
```

```
router.get('/', controller.retrieveAll)
router.get('/:id', controller.retrieveOne)
router.put('/:id', controller.update)
router.delete('/:id', controller.delete)

export default router
```

9. Vamor retornar ao arquivo **front-end/src/ui/AuthControl.jsx** e alterar a forma como o *front-end* processa o *logout*:

```
// (...código existente...)
  async function handleLogoutButtonClick() {
    if(await askForConfirmation('Deseja realmente sair?')) {
      showWaiting(true)
      try {
        // Faz uma requisição ao back-end solicitando a
        // exclusão do cookie com o token de autorização
        await myfetch.post('/users/logout')

        // Apaga as informações em memória sobre o usuário
        // autenticado
        setAuthUser(null)

        // Redireciona para a página de login
        navigate('/login')
      }
      catch(error) {

      }
      finally {
        showWaiting(false)
      }
    }
  }
```

```
// (...código existente...)
```

a. Agora, ao fazer o *logout*, primeiramente é enviada uma requisição ao *back-end* solicitando a exclusão do *cookie* com o *token*. Se você fizer o *logout* no *front-end* enquanto mantém a aba correspondente das Ferramentas de Desenvolvedor aberta, verá o *cookie* desaparecer ao vivo.

10. Com isso, concluímos todo o processo necessário para transferir o armazenamento do *token* de autorização do *local storage* para um *cookie* HTTP seguro.
11. Vamos terminar as atividades do dia com um *commit* + *push* no repositório do GitHub.
    a. Feche todos os terminais abertos usando o ícone da lixeira e abra um terminal novo. Nele, execute:

```
git add .
git commit -m "(18/10) Término da transferência do armazenamento do token JWT para cookie HTTP seguro"
git push
```

    b. **IMPORTANTE: o Trabalho 2 da disciplina será constituído pelo conjunto de *commits* no repositório do aluno, que deve ser um *fork* do repositório do professor. Serão considerados, para fins avaliativos, apenas os *commits* feitos entre o dia da aula e o dia anterior à aula seguinte. Por exemplo, o *commit* de uma aula de sábado deve ser feito, no máximo, até a terça-feira seguinte.**

22/10

1. No VS Code, abra um terminal e use o botão para duplicar o terminal.
   a. No terminal da esquerda:

```
cd back-end
npm run dev
```

   b. No terminal da direita:

```
cd front-end
npm run dev
```

2. Acesse o *front-end* em http://localhost:5173.
3. Abra um terceiro terminal na pasta **`back-end`** e execute **`npx prisma studio`**. Acesse a tabela de usuários para ver os usuários cadastrados.

4. Escolha um usuário ***que não seja administrador*** e faça *login* na aplicação Vulcom. A senha de ***todos*** os usuários de *id* 2 a 5 é **senha123**.
5. Abra uma nova requisição no RapidAPI:
    a. Método: **GET**
    b. URL: http://localhost:8080/users
    c. Na aba "Auth/Bearer", cole o *token* do usuário logado no item 4 (Vá em Ferramentas de Desenvolvedor > Aba "Application" > Cookies > http://localhost:5173 para copiar)
    d. Clique sobre o botão "Send".
6. Se tudo estiver correto, você verá uma listagem com TODOS os usuários cadastrados. No entanto, isso é uma grave falha de segurança, pois um usuário não adminstrador pode acessar (e modificar, e até excluir) dados de outros usuários, inclusive aqueles com *status* de administrador. É necessário corrigir isso.
7. Vamos modificar os métodos do *controller* de usuários. Abra o arquivo **back-end/src/controllers/users.js** e faça as alterações marcadas em azul:
    a. no método **create()**, vamos impedir a respectiva execução caso o usuário não seja administrador. Ou seja, apenas um usuário com *status* de administrador poderá criar outros usuários:

```
controller.create = async function(req, res) {
  try {

    // Somente usuários administradores podem acessar este recurso
    // HTTP 403: Forbidden(
    if(! req?.authUser?.is_admin) return res.status(403).end()

    // Verifica se existe o campo "password" em "req.body".
    // Caso positivo, geramos o hash da senha antes de enviá-la
    // ao BD
    // (12 na chamada a bcrypt.hash() corresponde ao número de
    // passos de criptografia utilizados no processo)
    if(req.body.password) {
      req.body.password = await bcrypt.hash(req.body.password, 12)
    }

    await prisma.user.create({ data: req.body })
```

```
    // HTTP 201: Created
    res.status(201).end()
  }
  catch(error) {
    console.error(error)

    // HTTP 500: Internal Server Error
    res.status(500).end()
  }
}
```

b. o mesmo deve ocorrer no método **`retrieveAll()`**:

```
controller.retrieveAll = async function(req, res) {
  try {

    // Somente usuários administradores podem acessar este recurso
    // HTTP 403: Forbidden(
    if(! req?.authUser?.is_admin) return res.status(403).end()

    const result = await prisma.user.findMany(
      // Omite o campo "password" do resultado
      // por questão de segurança
      { omit: { password: true } }
    )

    // HTTP 200: OK (implícito)
    res.send(result)
  }
  catch(error) {
    console.error(error)

    // HTTP 500: Internal Server Error
```

```
    res.status(500).end()
  }
}
```

c. Já no método **`retrieveOne()`**, se o usuário for administrador, poderá acessar os dados de qualquer usuário; caso contrário, poderá acessar ***seus próprios*** dados.

```
controller.retrieveOne = async function(req, res) {
  try {

    // Somente usuários administradores ou o próprio usuário
    // autenticado podem acessar este recurso
    // HTTP 403: Forbidden
    if(! (req?.authUser?.is_admin ||
      Number(req?.authUser?.id) === Number(req.params.id)))
      return res.status(403).end()

    const result = await prisma.user.findUnique({
      // Omite o campo "password" do resultado
      // por questão de segurança
      omit: { password: true },
      where: { id: Number(req.params.id) }
    })

    // Encontrou ~> retorna HTTP 200: OK (implícito)
    if(result) res.send(result)
    // Não encontrou ~> retorna HTTP 404: Not Found
    else res.status(404).end()
  }
  catch(error) {
    console.error(error)

    // HTTP 500: Internal Server Error
    res.status(500).end()
```

```
  }
}
```

d. Nos métodos **update()** e **delete()**, retornamos à lógica de acesso exclusivo a usuários administradores:

```
controller.update = async function(req, res) {
  try {

    // Somente usuários administradores podem acessar este recurso
    // HTTP 403: Forbidden(
    if(! req?.authUser?.is_admin) return res.status(403).end()

    // Verifica se existe o campo "password" em "req.body".
    // Caso positivo, geramos o hash da senha antes de enviá-la
    // ao BD
    // (12 na chamada a bcrypt.hash() corresponde ao número de
    // passos de criptografia utilizados no processo)
    if(req.body.password) {
      req.body.password = await bcrypt.hash(req.body.password, 12)
    }

    const result = await prisma.user.update({
      where: { id: Number(req.params.id) },
      data: req.body
    })

    // Encontrou e atualizou ~> HTTP 204: No Content
    if(result) res.status(204).end()
    // Não encontrou (e não atualizou) ~> HTTP 404: Not Found
    else res.status(404).end()
  }
  catch(error) {
    console.error(error)
```

```
    // HTTP 500: Internal Server Error
    res.status(500).end()
  }
}
```

```
controller.delete = async function(req, res) {
  try {

    // Somente usuários administradores podem acessar este recurso
    // HTTP 403: Forbidden(
    if(! req?.authUser?.is_admin) return res.status(403).end()

    await prisma.user.delete({
      where: { id: Number(req.params.id) }
    })

    // Encontrou e excluiu ~> HTTP 204: No Content
    res.status(204).end()
  }
  catch(error) {
    if(error?.code === 'P2025') {
      // Não encontrou e não excluiu ~> HTTP 404: Not Found
      res.status(404).end()
    }
    else {
      // Outros tipos de erro
      console.error(error)

      // HTTP 500: Internal Server Error
      res.status(500).end()
    }
  }
```

```
}
```

8. Repita a requisição do item 5. Agora você deve receber uma resposta 403 Forbidden. No entanto, caso você altere a URL para **`http://localhost:8080/users/X`**, onde **X** é o *id* do usuário logado, verá normalmente os respectivos dados.
9. No menu do Projeto Vulcom, o item que dá acesso ao cadastro de usuários não aparece para usuários não administradores. Isso é implementado no arquivo **`front-end/src/ui/MainMenu.jsx`**.
10. No entanto, caso um usuário não administrador acesse diretamente a rota http://localhost:5173/users, ainda terá acesso ao cadastro de usuários, embora não consiga mais ver os dados devido à proteção que fizemos no item 7 letra b). Mesmo assim, é importante bloquear o acesso também à rota, já que, dessa forma, estaremos diminuindo a superfície de ataque da aplicação.
11. A proteção de rotas no *front-end* é feita por meio de guardiões de rotas (*route guards*). Já temos toda a lógica dos guardiões no arquivo **`front-end/src/routes/AuthGuard.jsx`**. Falta agora aplicá-los às rotas.
12. As modificações que precisamos fazer para ativar os guardiões devem ser feitas no arquivo **`front-end/src/routes/AppRoutes.jsx`**.
    a. Importe o **`AuthGuard`** no arquivo **`AppRoutes`**:

```
import AuthGuard from './AuthGuard'
```

    b. Em seguida, precisamos conhecer o nível de acesso de cada rota para implementar (ou não) o AuthGuard para cada uma delas, conforme segue:
        i. Elementos que podem ser acessados por **qualquer usuário** (ou mesmo sem um usuário logado): continuam como estão, **sem `AuthGuard`**;
        ii. Elementos que **necessitam de um usuário logado** para serem acessados: **devem ser envolvidas por um `AuthGuard`** (**`<AuthGuard> <Elemento /> </AuthGuard>`**);
        iii. Elementos a que **só o usuário administrador pode ter acesso**: **uso do `AuthGuard` com a *prop* `adminOnly` ativada** (**`<AuthGuard adminOnly={true}> <Elemento /> </AuthGuard>`**).
    c. Assim, vamos alterar o código do componente AppRoutes de acordo com o esquema:

```
// (...código existente...)
export default function AppRoutes() {
  return <Routes>
    <Route path="/" element={ <Homepage /> } />

    <Route path="/login" element={ <Login /> } />
```

```
    <Route path="/cars" element={ <AuthGuard> <CarList /> </AuthGuard> } />
    <Route path="/cars/new" element={ <AuthGuard> <CarForm /> </AuthGuard> } />
    <Route path="/cars/:id" element={ <AuthGuard> <CarForm /> </AuthGuard> } />

    <Route path="/customers" element={
      <AuthGuard> <CustomerList /> </AuthGuard>
    } />

    <Route path="/customers/new" element={
      <AuthGuard> <CustomerForm /> </AuthGuard>
    } />
    <Route path="/customers/:id" element={
      <AuthGuard> <CustomerForm /> </AuthGuard>
    } />

    <Route path="/users" element={
      <AuthGuard adminOnly={true}> <UserList /> </AuthGuard> }
    />
    <Route path="/users/new" element={
      <AuthGuard adminOnly={true}> <UserForm /> </AuthGuard> }
    />
    <Route path="/users/:id" element={
      <AuthGuard adminOnly={true}> <UserForm /> </AuthGuard> }
    />

  </Routes>
}
```

13. Agora, caso um usuário não administrador tente acessar diretamente a URL http://localhost:5173/users, verá apenas uma mensagem de Acesso Negado.
14. Vamos terminar as atividades do dia com um *commit* + *push* no repositório do GitHub.
    a. Feche todos os terminais abertos usando o ícone da lixeira e abra um terminal novo. Nele, execute:
       `git add .`

```
git commit -m "(22/10) Proteção de rotas no back-end e no front-end"
git push
```

b. **IMPORTANTE: o Trabalho 2 da disciplina será constituído pelo conjunto de *commits* no repositório do aluno, que deve ser um *fork* do repositório do professor. Serão considerados, para fins avaliativos, apenas os *commits* feitos entre o dia da aula e o dia anterior à aula seguinte. Por exemplo, o *commit* de uma aula de sábado deve ser feito, no máximo, até a terça-feira seguinte.**

25/10

1. No VS Code, abra um terminal e use o botão ⫾ para duplicar o terminal.
   a. No terminal da esquerda:
   ```
   cd back-end
   npm run dev
   ```
   b. No terminal da direita:
   ```
   cd front-end
   npm run dev
   ```
2. Acesse o *front-end* em http://localhost:5173.
3. Com o trabalho feito na última aula, as rotas de *front-end* foram protegidas contra acessos de usuários não autorizados ou com autorização insuficiente. No entanto, as definições de níveis de acesso foram feitas de forma separada nos arquivos **`front-end/src/routes/AppRoutes.jsx`** e **`front-end/src/ui/MainMenu.jsx`**. A cada nova página/rota, é necessário inserir as informações de acesso nesses dois arquivos, levando ao risco de introduzir inconsistências caso informações diferentes sejam colocadas neles.
4. Por isso, o ideal é centralizar a definição das rotas e dos níveis de permissão em um único local, que passará a funcionar como **fonte única de verdade** sobre essas informações. Vamos começar criando o arquivo **`front-end/src/routes/routes.jsx`** com o seguinte conteúdo:

```jsx
/*
 Define as rotas e suas informações, servindo como fonte única
 de verdade para AppRoutes.jsx e MainMenu.jsx.
*/

import Homepage from '../pages/Homepage'
```

```
import Login from '../pages/Login'

import CustomerList from '../pages/customer/CustomerList'
import CustomerForm from '../pages/customer/CustomerForm'

import CarList from '../pages/car/CarList'
import CarForm from '../pages/car/CarForm'

import UserList from '../pages/user/UserList'
import UserForm from '../pages/user/UserForm'

/*
 Os níveis de acesso foram definidos como segue:
 0 ~> qualquer usuário (incluindo quando não há usuário autenticado)
 1 ~> qualquer usuário autenticado
 2 ~> somente usuário administrador
*/
const UserLevel = {
 ANY: 0,
 AUTHENTICATED: 1,
 ADMIN: 2
}

const routes = [
 {
   route: '/',
   description: 'Início',
   element: <Homepage />,
   userLevel: UserLevel.ANY,
   divider: true
 },
 {
```

```
  route: '/login',
  description: 'Entrar',
  element: <Login />,
  userLevel: UserLevel.ANY,
  omitFromMainMenu: true
},
{
  route: '/customers',
  description: 'Listagem de clientes',
  element: <CustomerList />,
  userLevel: UserLevel.AUTHENTICATED
},
{
  route: '/customers/new',
  description: 'Cadastro de clientes',
  element: <CustomerForm />,
  userLevel: UserLevel.AUTHENTICATED,
  divider: true
},
{
  route: '/customers/:id',
  description: 'Alterar cliente',
  element: <CustomerForm />,
  userLevel: UserLevel.ADMIN,
  omitFromMainMenu: true
},
{
  route: '/cars',
  description: 'Listagem de veículos',
  element: <CarList />,
  userLevel: UserLevel.AUTHENTICATED
},
{
```

```
    route: '/cars/new',
    description: 'Cadastro de veículos',
    element: <CarForm />,
    userLevel: UserLevel.AUTHENTICATED,
    divider: true
  },
  {
    route: '/cars/:id',
    description: 'Alterar veículo',
    element: <CarForm />,
    userLevel: UserLevel.ADMIN,
    omitFromMainMenu: true
  },
  {
    route: '/users',
    description: 'Listagem de usuários',
    element: <UserList />,
    userLevel: UserLevel.ADMIN
  },
  {
    route: '/users/new',
    description: 'Cadastro de usuários',
    element: <UserForm />,
    userLevel: UserLevel.ADMIN
  },
  {
    route: '/users/:id',
    description: 'Alterar usuário',
    element: <UserForm />,
    userLevel: UserLevel.ADMIN,
    omitFromMainMenu: true
  },
]
```

```
export { routes, UserLevel }
```

5. Todas as rotas da aplicação *front-end* foram definidas nesse arquivo com seu respectivo nível de acesso, inclusive aquelas não acessíveis diretamente por intermédio do menu principal (itens marcados com **`omitFromMainMenu = true`**).
6. O passo seguinte é adaptar o **`AuthGuard`** (**`front-end/src/routes/AuthGuard.jsx`**) para consumir informações do arquivo **`routes.jsx`**:

```
import React from 'react'
import { Navigate, useLocation, useNavigate } from 'react-router-dom'
import myfetch from '../lib/myfetch'
import AuthUserContext from '../contexts/AuthUserContext'
import useWaiting from '../ui/useWaiting'
import Box from '@mui/material/Box'
import Typography from '@mui/material/Typography'

import { UserLevel } from './routes'

export default function AuthGuard({ children, userLevel = UserLevel.ANY }) {

 const { setAuthUser, authUser, setRedirectLocation } =
React.useContext(AuthUserContext)
  const [status, setStatus] = React.useState('IDLE')

 const location = useLocation()
 const { showWaiting, Waiting } = useWaiting()
 const navigate = useNavigate()

 async function checkAuthUser() {
   if(setStatus) setStatus('PROCESSING')
   showWaiting(true)
   try {
     const authUser = await myfetch.get('/users/me')
     setAuthUser(authUser)
```

```
  }
  catch(error) {
    setAuthUser(null)
    console.error(error)
    navigate('/login', { replace: true })
  }
  finally {
    showWaiting(false)
    setStatus('DONE')
  }
}

React.useEffect(() => {
  // Salva a rota atual para posterior redirecionamento,
  // caso a rota atual não seja o próprio login
  if(! location.pathname.includes('login')) setRedirectLocation(location)

  checkAuthUser()
}, [location])

// Enquanto ainda não temos a resposta do back-end para /users/me,
// exibimos um componente Waiting
if(status === 'PROCESSING') return <Waiting />

/*
  Se não há usuário autenticado e o nível de acesso assim o
  exige, redirecionamos para a página de login
*/
if(!authUser && userLevel > UserLevel.ANY) {
  console.log({authUser, userLevel})
  return <Navigate to="/login" replace />
}
```

```
  /*
    Senão, se há um usuário não administrador tentando acessar uma
    rota exclusiva para esse nível, mostramos uma mensagem de acesso negado
  */
  if(!(authUser?.is_admin) && userLevel === UserLevel.ADMIN) return (
    <Box>
      <Typography variant="h2" color="error">
        Acesso negado
      </Typography>
    </Box>
  )

    /*
      Se chegou até aqui, é porque a rota é liberada para qualquer
      um ou o usuário possui autorização para acessar o
      nível
    */
    console.log('AUTHGUARD:', authUser)
    return children
  }
```

7. Agora, vamos alterar o arquivo **front-end/src/routes/AppRoutes** para consumir informações do arquivo **routes.jsx**. Note que o novo conteúdo se torna até mais simples do que o original.

```
import { Routes, Route } from 'react-router-dom'

import AuthGuard from './AuthGuard'

import { routes, UserLevel } from './routes'

export default function AppRoutes() {
 return (
   <Routes>
     {
```

```
        routes.map(route => {
          let element
          if(route.userLevel > UserLevel.ANY) {
            element = <AuthGuard userLevel={route.userLevel}>
              {route.element}
            </AuthGuard>
          }
          else element = route.element

          return <Route
            key={route.route}
            path={route.route}
            element={element}
          />
        })
      }
    </Routes>
  )
}
```

8. **Devido a um erro em uma linha de `import`, copie novamente o código do `AuthGuard` no item 6. O erro já foi corrigido lá.**
9. **Copie também novamente o conteúdo do arquivo `routes.jsx` do item 4. Erros foram corrigidos.**
10. Por fim, vamos modificar **`front-end/src/ui/MainMenu.jsx`** para que também busque informações no arquivo **`routes.jsx`**:

```
import * as React from 'react';
import Menu from '@mui/material/Menu';
import MenuItem from '@mui/material/MenuItem';
import IconButton from '@mui/material/IconButton';
import { Link } from 'react-router-dom';
import MenuIcon from '@mui/icons-material/Menu';
import AuthUserContext from '../contexts/AuthUserContext'
import { routes, UserLevel } from '../routes/routes'

export default function MainMenu() {
```

```
const [anchorEl, setAnchorEl] = React.useState(null);
const open = Boolean(anchorEl);
const handleClick = (event) => {
  setAnchorEl(event.currentTarget);
};
const handleClose = () => {
  setAnchorEl(null);
};

const { authUser } = React.useContext(AuthUserContext)

// Determina o nível do usuário autenticado
let currentUserLevel = UserLevel.ANY

if(!authUser) currentUserLevel = UserLevel.ANY
else if(authUser.is_admin) currentUserLevel = UserLevel.ADMIN
else currentUserLevel = UserLevel.AUTHENTICATED

/*
  Filtra as rotas que se tornarão itens de menu, excluindo:
  - rotas com omitFromMainMenu === true
  - rotas com userLevel > currentUserLevel
*/
const menuRoutes = routes.filter(
  r => !(r?.omitFromMainMenu) && r.userLevel <= currentUserLevel
)

return (
  <div>
      <IconButton
          edge="start"
          color="inherit"
          aria-label="menu"
```

```
          sx={{ mr: 2 }}
          aria-controls={open ? 'basic-menu' : undefined}
          aria-haspopup="true"
          aria-expanded={open ? 'true' : undefined}
          onClick={handleClick}

        >
          <MenuIcon />
        </IconButton>
      <Menu
        id="basic-menu"
        anchorEl={anchorEl}
        open={open}
        onClose={handleClose}
        slotProps={{
          list: {
            'aria-labelledby': 'basic-button',
          }
        }}
      >
        {
          menuRoutes.map(r => (
            <MenuItem
              key={r.route}
              onClick={handleClose}
              component={Link}
              to={r.route}
              divider={r?.divider}
            >
              {r.description}
            </MenuItem>
          ))
        }
```

```
      </Menu>
    </div>
  );
}
```

11. Terminamos. De agora em diante, cada nova página/rota só precisará ser definida no arquivo **`routes.jsx`**, e será processada automaticamente pelos arquivos **`AuthGuard.jsx`**, **`AppRoutes.jsx`** e **`MainMenu.jsx`**.
12. Com isso, terminamos a reorganização das rotas de *front-end*. Na próxima aula, iniciaremos a implementação de um simulador de ataque de força bruta contra o *back-end*.
13. Vamos terminar as atividades do dia com um *commit* + *push* no repositório do GitHub.
    a. Feche todos os terminais abertos usando o ícone da lixeira e abra um terminal novo. Nele, execute:

    ```
    git add .
    git commit -m "(25/10) Finalização da reorganização das rotas do front-end"
    git push
    ```

    b. **IMPORTANTE: o Trabalho 2 da disciplina será constituído pelo conjunto de *commits* no repositório do aluno, que deve ser um *fork* do repositório do professor. Serão considerados, para fins avaliativos, apenas os *commits* feitos entre o dia da aula e o dia anterior à aula seguinte. Por exemplo, o *commit* de uma aula de sábado deve ser feito, no máximo, até a terça-feira seguinte.**

29/10

1. No VS Code, abra um terminal e use o botão para duplicar o terminal.
   a. No terminal da esquerda:

   ```
   cd back-end
   npm run dev
   ```

   b. No terminal da direita:

   ```
   cd front-end
   npm run dev
   ```

2. Acesse o *front-end* em http://localhost:5173.
3. Hoje iniciaremos a implementação de uma interface no *front-end* para efetivar um ataque de força bruta contra o *back-end*.
4. Para começar, crie a pasta **`data`** dentro de **`front-end/src`**. Baixe este arquivo (**`wordlist.js`**) e o coloque dentro da nova pasta.

5. Em seguida, crie o arquivo **BruteForce.jsx** na pasta **front-end/src/pages** com o conteúdo a seguir:

```jsx
import React from 'react'
import wordlist from '../data/wordlist'
import myfetch from '../lib/myfetch'

export default function BruteForce() {
 const [logs, setLogs] = React.useState([])
 const [isRunning, setIsRunning] = React.useState(false)
 const stopRef = React.useRef(false)
 const logsEndRef = React.useRef(null)
 const [concurrency, setConcurrency] = React.useState(5) // Número de requisições paralelas

 // Auto-scroll para o topo quando novos logs são adicionados
 React.useEffect(() => {
   logsEndRef.current?.scrollTo(0, 0)
 }, [logs])

 function tryPassword(password) {
   return myfetch.post('/users/login', {
     username: 'admin',
     password
   })
   .then(() => 'OK')
   .catch(error => error.message)
 }

 async function handleStartClick() {
   setIsRunning(true)
   stopRef.current = false
   setLogs([])

   const batchSize = concurrency
   let found = false

   for (let i = 0; i < wordlist.length && !found; i += batchSize) {
     if (stopRef.current) break
```

```
    const batch = wordlist.slice(i, i + batchSize)
    const promises = batch.map((password, index) =>
      tryPassword(password)
        .then(result => {
          const attemptNumber = i + index
          const newLog = result === 'OK'
            ? `SENHA ENCONTRADA, tentativa nº ${attemptNumber}: ${password}`
            : `Tentativa nº ${attemptNumber} (${password}) => ${result}`

          setLogs(prevLogs => [newLog, ...prevLogs])
          return result
        })
    )

    const results = await Promise.all(promises)
    found = results.some(result => result === 'OK')

    // Pequena pausa para evitar bloqueio da UI
    await new Promise(resolve => requestAnimationFrame(resolve))
  }

  setIsRunning(false)
}

function handleStopClick() {
  stopRef.current = true
}

return (
  <>
    <h1>Ataque de força bruta no <em>login</em></h1>
    <div style={{ marginBottom: '1rem' }}>
      <label>
        Concorrência:
        <input
```

```
            type="number"
            min="1"
            max="20"
            value={concurrency}
            onChange={(e) => setConcurrency(Math.min(20, Math.max(1,
parseInt(e.target.value) || 1)))}
            disabled={isRunning}
            style={{ marginLeft: '0.5rem', width: '50px' }}
          />
        </label>
      </div>
      <div style={{
        display: 'flex',
        justifyContent: 'space-around',
        marginBottom: '1rem'
      }}>
        <button onClick={handleStartClick} disabled={isRunning}>
          Iniciar
        </button>
        <button onClick={handleStopClick} disabled={!isRunning}>
          Parar
        </button>
      </div>
      <div
        ref={logsEndRef}
        style={{
          fontFamily: 'monospace',
          height: '400px',
          overflowY: 'auto',
          border: '1px solid #ccc',
          padding: '0.5rem',
          color: '#555',
          backgroundColor: '#f5f5f5'
        }}
      >
        {logs.length > 0 ? (
```

```
          <ul style={{ listStyleType: 'none', padding: 0, margin: 0 }}>
            {logs.map((log, index) => (
              <li
                key={index}
                style={{
                  padding: '0.25rem 0',
                  borderBottom: index < logs.length - 1 ? '1px solid #eee' : 'none',
                  color: log.includes('SENHA ENCONTRADA') ? 'green' : 'inherit'
                }}
              >
                {log}
              </li>
            ))}
          </ul>
        ) : (
          <div style={{ color: '#555' }}>Nenhum log disponível. Clique em Iniciar para
começar.</div>
        )}
      </div>
    </>
  )
}
```

6. Vamos criar uma nova rota para acessar a página do ataque de força bruta, no arquivo **front-end/src/routes/routes.jsx** (acrescente as linhas marcadas em amarelo):

```
/*
 Define as rotas e suas informações, servindo como fonte única
 de verdade para AppRoutes.jsx e MainMenu.jsx.
*/

import Homepage from '../pages/Homepage'

import Login from '../pages/Login'

import CustomerList from '../pages/customer/CustomerList'
import CustomerForm from '../pages/customer/CustomerForm'
```

```
import CarList from '../pages/car/CarList'
import CarForm from '../pages/car/CarForm'

import UserList from '../pages/user/UserList'
import UserForm from '../pages/user/UserForm'

import BruteForce from '../pages/BruteForce'

/*
 Os níveis de acesso foram definidos como segue:
 0 ~> qualquer usuário (incluindo quando não há usuário autenticado)
 1 ~> qualquer usuário autenticado
 2 ~> somente usuário administrador
*/
const UserLevel = {
 ANY: 0,
 AUTHENTICATED: 1,
 ADMIN: 2
}

const routes = [
 {
   route: '/',
   description: 'Início',
   element: <Homepage />,
   userLevel: UserLevel.ANY,
   divider: true
 },
 {
   route: '/login',
   description: 'Entrar',
   element: <Login />,
   userLevel: UserLevel.ANY,
   omitFromMainMenu: true
 },
```

```
{
  route: '/customers',
  description: 'Listagem de clientes',
  element: <CustomerList />,
  userLevel: UserLevel.AUTHENTICATED
},
{
  route: '/customers/new',
  description: 'Cadastro de clientes',
  element: <CustomerForm />,
  userLevel: UserLevel.AUTHENTICATED,
  divider: true
},
{
  route: '/customers/:id',
  description: 'Alterar cliente',
  element: <CustomerForm />,
  userLevel: UserLevel.ADMIN,
  omitFromMainMenu: true
},
{
  route: '/cars',
  description: 'Listagem de veículos',
  element: <CarList />,
  userLevel: UserLevel.AUTHENTICATED
},
{
  route: '/cars/new',
  description: 'Cadastro de veículos',
  element: <CarForm />,
  userLevel: UserLevel.AUTHENTICATED,
  divider: true
},
{
  route: '/cars/:id',
  description: 'Alterar veículo',
```

```
    element: <CarForm />,
    userLevel: UserLevel.ADMIN,
    omitFromMainMenu: true
  },
  {
    route: '/users',
    description: 'Listagem de usuários',
    element: <UserList />,
    userLevel: UserLevel.ADMIN
  },
  {
    route: '/users/new',
    description: 'Cadastro de usuários',
    element: <UserForm />,
    userLevel: UserLevel.ADMIN
  },
  {
    route: '/users/:id',
    description: 'Alterar usuário',
    element: <UserForm />,
    userLevel: UserLevel.ADMIN,
    omitFromMainMenu: true
  },
  {
    route: '/brute-force',
    description: 'Ataque de força bruta',
    element: <BruteForce />,
    userLevel: UserLevel.ADMIN,
    divider: true
  },
]

export { routes, UserLevel }
```

a. Havia um pequeno erro na linha marcada em verde acima, o correto é `route: '/brute-force'`.

7. Abra o Projeto Vulcom, faça *login* como o usuário administrador (**admin**/**Vulcom@DSM**) e acesse o item de menu que dá acesso à página de ataque de força bruta. Nessa página, clique sobre o botão "Iniciar" para começar um ataque de força bruta.

    a. O ataque de força bruta consiste em tentar todas as entradas constantes no arquivo **`wordlist.js`** como senhas para o usuário **`admin`**, enviando requisições em *loop* para o *endpoint* **`/users/login`** do *back-end*. Se você deixar o ataque rodando até aproximadamente 13200 tentativas, ele irá acertar a senha.

8. No caso em questão, a senha correta foi inserida propositalmente no arquivo **`wordlist.js`** para fins didáticos. No entanto, mesmo que a senha correta não estivesse no arquivo, a aplicação *back-end* não deveria permitir um número indefinido de tentativas de *login*.
    a. Se você abrir a aba "Console" das Ferramentas de Desenvolvedor enquanto o ataque de força bruta está sendo executado, verá que as requisições retornam com o código de erro **`HTTP 401: Unauthorized`**, indicando que a senha está incorreta.
9. As abordagens mais usadas para resolver a vulnerabilidade de número indefinido de tentativas de *login* são as seguintes:
    a. **Bloquer o acesso do usuário após um número predeterminado de tentativas inválidas**. Nesse caso, seria necessária a intervenção de um outro usuário, administrador, para a retirada do bloqueio. É o que acontece, por exemplo, no SIGA, após 5 tentativas infrutíferas.
    b. **Limitar a quantidade de requisições (tentativas) que um usuário pode fazer durante uma janela de tempo predeterminada**. Essa alternativa é especialmente adequada contra ataques automatizados, como o ataque de força bruta que estamos efetuando. Um ataque de força bruta eficiente fará o maior número de tentativas no menor tempo possível, com o objetivo de descobrir a senha correta o mais rápido possível. Limitando a quantidade de tentativas que um usuário pode realizar em um certo período de tempo acaba atrasando o ataque de força bruta, tornando-o inviável na prática.
10. Iremos adotar uma solução baseada na alternativa b). O *framework* Express.js conta com um *plugin* chamado **`express-rate-limit`**, o qual podemos instalar e configurar para proteger nosso *back-end* contra ataques de força bruta.
    a. Caso o *back-end* esteja em execução, interrompa-o teclando Ctrl+C no terminal. Em seguida, assegure-se de estar na pasta back-end e execute os comandos abaixo, um por vez:

    ```
    npm install express-rate-limit
    npm run dev
    ```

    b. Em seguida, vamos configurar o *plugin* no arquivo **`back-end/src/app.js`**:

```
// (...código existente)

app.use(logger('dev'))
app.use(json())
app.use(urlencoded({ extended: false }))
app.use(cookieParser())

// Rate limiter: limita a quantidade de requisições que cada usuário/IP
// pode efetuar dentro de um determinado intervalo de tempo
```

```
import { rateLimit } from 'express-rate-limit'

const limiter = rateLimit({
 windowMs: 60 * 1000,    // Intervalo: 1 minuto
 limit: 20               // Máximo de 20 requisições
})

app.use(limiter)

// (...código existente)
```

11. Reinicie o ataque de força bruta. Agora, após a vigésima tentativa, o back-end passará a retornar o erro **`HTTP 429: Too Many Requests`**, indicando que aquela tentativa sequer chegou a ser processada, fazendo com que o atacante perca tempo enviando várias senhas que não serão testadas.
12. Na próxima aula, vamos refinar o ataque de força bruta, para que ele seja mais informativo, e vamos testar também outros cenários.
13. Vamos terminar as atividades do dia com um *commit* + *push* no repositório do GitHub.
    a. Feche todos os terminais abertos usando o ícone da lixeira e abra um terminal novo. Nele, execute:
    ```
    git add .
    git commit -m "(29/10) Início da implementação de ataque de força bruta"
    git push
    ```
    b. **IMPORTANTE: o Trabalho 2 da disciplina será constituído pelo conjunto de *commits* no repositório do aluno, que deve ser um *fork* do repositório do professor. Serão considerados, para fins avaliativos, apenas os *commits* feitos entre o dia da aula e o dia anterior à aula seguinte. Por exemplo, o *commit* de uma aula de sábado deve ser feito, no máximo, até a terça-feira seguinte.**

01/11

1. No VS Code, abra um terminal e use o botão para duplicar o terminal.
    a. No terminal da esquerda:
    ```
    cd back-end
    npm run dev
    ```
    b. No terminal da direita:
    ```
    cd front-end
    ```

```
npm run dev
```

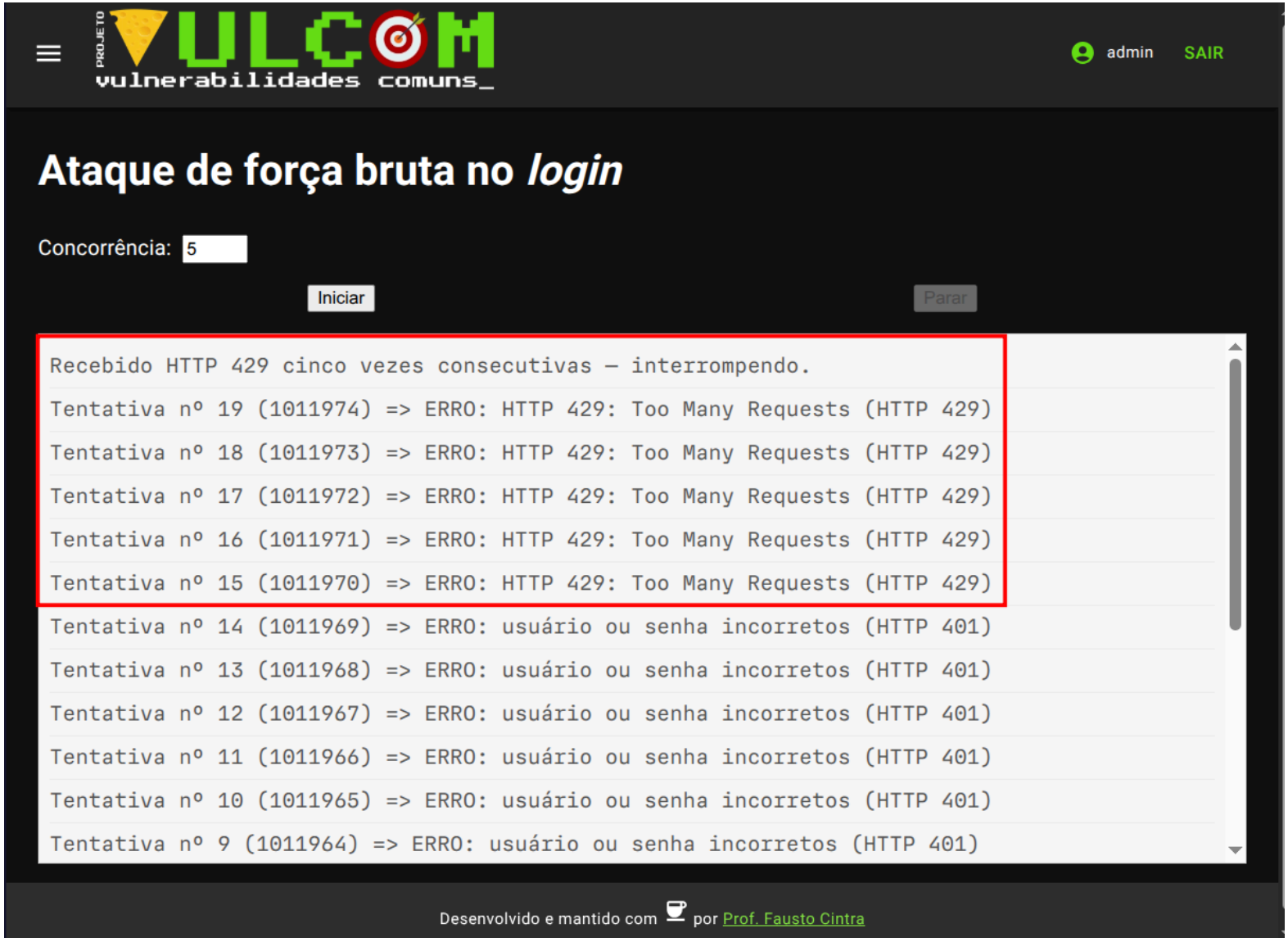
2. Acesse o *front-end* em http://localhost:5173.
3. No ataque de força bruta que implementamos na última aula, o atacante vigia o código de status HTTP de cada tentativa de *login* para monitorar a situação do ataque. Ele espera receber um dos seguintes códigos:
    a. **`HTTP 401: Unauthorized`** ~> significa que as credenciais enviadas na tentativa foram processadas, mas a senha está incorreta. O ataque deve continuar.
    b. **`HTTP 201: No Content`** ~> as credenciais enviadas foram processadas, e a senha enviada na tentativa corresponde à senha correta. O ataque pode terminar, pois a senha correta foi encontrada.
4. No entato, a partir do momento que implementamos o limite de requisições por usuário e tempo com express-rate-limit, o *back-end* passa a retornar o código **`HTTP 429: Too Many Attempts`**. Isso significa que as credenciais enviadas na tentativa sequer foram processadas, de modo que não há como ter certeza de que a senha da vez é a correta ou não. Em outras palavras, a continuação do ataque representa uma perda de tempo para o atacante.
5. Portanto, vamos modificar o método **`handleStartClick()`** do arquivo **`front-end/src/pages/BruteForce.jsx`** para monitorar o recebimento de códigos de *status* **`HTTP 429`** e interromper o ataque caso cinco desses *status* sejam recebidos consecutivamente:

```
// (...código existente)

async function handleStartClick() {
   setIsRunning(true)
   stopRef.current = false
   setLogs([])

   const batchSize = concurrency
   let found = false
   let consecutive429 = 0

   for (let i = 0; i < wordlist.length && !found && !stopRef.current; i += batchSize) {
     if (stopRef.current) break

     const batch = wordlist.slice(i, i + batchSize)
     const promises = batch.map(password =>
       myfetch.post('/users/login', { username: 'admin', password })
         .then(() => ({ status: 200, result: 'OK', password }))
         .catch(error => {
```

```
          const status = error?.response?.status ?? error?.status ?? null
          return { status, result: error?.message ?? String(error), password }
        })
    )

    const results = await Promise.all(promises)

    for (let idx = 0; idx < results.length; idx++) {
      const res = results[idx]
      const attemptNumber = i + idx

      if (res.result === 'OK') {
        const newLog = `SENHA ENCONTRADA, tentativa nº ${attemptNumber}: ${res.password}`
        setLogs(prevLogs => [newLog, ...prevLogs])
        found = true
        break
      } else {
        const statusText = res.status ? ` (HTTP ${res.status})` : ''
        const newLog = `Tentativa nº ${attemptNumber} (${res.password}) =>
${res.result}${statusText}`
        setLogs(prevLogs => [newLog, ...prevLogs])

        if (res.status === 429) {
          consecutive429++
          if (consecutive429 >= 5) {
            setLogs(prev => ['Recebido HTTP 429 cinco vezes consecutivas —
interrompendo.', ...prev])
            stopRef.current = true
            break
          }
        } else {
          consecutive429 = 0
        }
      }
    }
```

```
      // Pequena pausa para evitar bloqueio da UI
      await new Promise(resolve => requestAnimationFrame(resolve))
    }

    setIsRunning(false)
}

// (...código existente)
```

6. Agora, ao efetuar o ataque de força bruta, ele será interrompido automaticamente caso o *back-end* passe a enviar *status* **HTTP 429**:



7. Vamos agora mudar o foco do nosso estudo para a validação da entrada de dados dos usuários. Se você acessar qualquer um dos três cadastros do Projeto Vulcom (clientes, veículos e usuários), será capaz de inserir novos registros ou modificar os existentes com praticamente qualquer informação. No momento, não há qualquer validação da entrada de dados inseridos pelo usuários do sistema.
8. Uma questão comum é se a validação de dados de entrada deve ser feita no *front-end* ou no *back-end*. Vejamos os cenários:

a. Validação no *front-end*: mais rápida, com *feedback* instantâneos sem necessidade de enviar uma requisição ao *back-end*. No entanto, a validação vale apenas para o *front-end* - requisições feitas para a API por outros meios, como, por exemplo, o RapidAPI Client, não passarão pela validação.
b. Validação no *back-end*: tem a vantagem de ser acionada, independentemente da origem da requisição (*front-end* ou RapidAPI Client). Contudo, é necessário processar uma requisição para verificar a validade dos dados, atrasando o *feedback* ao usuário e potencialmente sobrecarregando o tráfego de rede, caso a largura de banda seja limitada.

9. Portanto, **o ideal é que a validação da entrada de dados de usuário seja feita tanto no *front-end* quanto no *back-end***. O desafio é como manter o mesmo conjunto de regras de validação em ambas as plataformas, para evitar inconsistências.
10. O Projeto Vulcom foi desenvolvido com Node.js + Express.js no *back-end* e React no *front-end*; em outras palavras, ele utiliza a linguagem JavaScript em ambas as frentes. Isso torna possível usar uma única biblioteca da linguagem para definir as regras de validação tanto para o *front-end* quanto para o *back-end*. Entre as muitas bibliotecas disponíveis para esse fim, utilizaremos a Zod, que se destaca por sua definição de regras de forma declarativa e por oferecer *feedback* detalhado para cada tipo de erro.
11. Vamos iniciar a validação de dados pelo *back-end*.
    a. Interrompa a execução do *back-end* teclando Ctrl+C no terminal. Em seguida, instale a biblioteca Zod e do validador de CPF/CNPJ e reinicie o o projeto:

       ```
       npm install zod cpf-cnpj-validator
       npm run dev
       ```
12. O Zod trabalha com *models*, que representam o conjunto de regras de validação para os campos de uma determinada entidade. Como exemplo, vamos criar o *model* para validar o cadastro de clientes.
    a. Crie a pasta **`models`** sob **`back-end/src`**.
    b. Nessa pasta, crie o arquivo **`Customer.js`** (**note a inicial maiúscula no nome do arquivo**) com o seguinte conteúdo:

```
import { z } from 'zod'
import { cpf } from 'cpf-cnpj-validator'

/*
 O cliente deve ser maior de 18 anos.
 Por isso, para validar a data de nascimento, calculamos
 a data máxima até a qual o cliente pode ter nascido (no
 passado) para ter, pelo menos, 18 anos na data atual
*/
const maxBirthDate = new Date()   // Hoje
maxBirthDate.setFullYear(maxBirthDate.getFullYear() - 18)
```

```
// O cliente pode ter, no máximo, 120 anos de idade
const minBirthDate = new Date()
minBirthDate.setFullYear(minBirthDate.getFullYear() - 120)

// Unidades da Federação
const unidadesFederacao = [
 'AC', 'AL', 'AP', 'AM', 'BA', 'CE', 'DF', 'ES', 'GO',
 'MA', 'MT', 'MS', 'MG', 'PA', 'PB', 'PR', 'PE', 'PI',
 'RJ', 'RN', 'RS', 'RO', 'RR', 'SC', 'SP', 'SE', 'TO'
]

const Customer = z.object({
 name: z.string()
   .trim()   // Retira eventuais espaços em branco das extremidades
   .min(5, { message: 'O nome deve ter, no mínimo, 5 caracteres. '})
   .max(100, { message: 'O nome deve ter, no máximo, 100 caracteres.' })
   .includes(' ', { message: 'O nome teve ter um espaço em branco separando prenome e
sobrenome.'}),

 ident_document: z.string()
   // Remove eventuais sublinhados (da máscara do campo usada no
   // front-end), caso o CPF não tenha sido completamente preenchido
   .transform(val => val.replace('_', ''))
   .refine(val => val.length === 14, {
     message: 'O CPF deve ter, exatamente, 14 caracteres.'
   })
   .refine(val => cpf.isValid(val), {
     message: 'CPF inválido.'
   }),

 birth_date:
   // Força a conversão para o tipo Date
   z.coerce.date()
   .min(minBirthDate, {
     message: 'Data de nascimento está muito no passado.'
   })
```

```
  .max(maxBirthDate, {
    message: 'O cliente deve ser maior de 18 anos.'
  })
  .nullish(),    // O campo é opcional

street_name: z.string()
  .trim()
  .min(1, { message: 'Logradouro deve ter, pelo menos, 1 caractere.' })
  .max(40, { message: 'Logradouro pode ter, no máximo, 40 caracteres.' }),

house_number: z.string()
  .trim()
  .min(1, { message: 'O número do imóvel deve ter, pelo menos, 1 caractere. '})
  .max(10, { message: 'O número do imóvel pode ter, no máximo, 10 caracteres.'}),

complements: z.string()
  .trim()
  .max(20, { message: 'Complemento pode ter, no máximo, 20 caracteres.' })
  .nullish(),

district: z.string()
  .trim()
  .min(1, { message: 'Bairro deve ter, no mínimo, 1 caractere.' })
  .max(25, { message: 'Bairro pode ter, no máximo, 25 caracteres.' }),

municipality: z.string()
  .trim()
  .min(1, { message: 'Município deve ter, no mínimo, 1 caractere.' })
  .max(40, { message: 'Município pode ter, no máximo, 40 caracteres.' }),

state: z.enum(unidadesFederacao, {
  message: 'Unidade da Federação inválida.'
}),

phone: z.string()
  .transform(val => val.replace('_', ''))
```

```
    // Depois de transform(), o Zod não permite usar length(). Por isso,
    // precisamos usar uma função personalizada com refine() para validar
    // o comprimento do valor
    .refine(val => val.length === 15, {
      message: 'O número do telefone/celular deve ter exatas 15 posições.'
    }),

  email: z.string()
    .email({ message: 'E-mail inválido.' })
})
export default Customer
```

13. Como dito anteriormente, as regras de validação do Zod são declarativas - isso significa que você define o que deve ser válido, e deixa a execução das validações por conta da bilbioteca.
    a. As regras são definidas em um objeto da linguagem JavaScript, na qual cada campo a ser validado corresponde a uma propriedade do objeto.
    b. Cada regra começa definindo o tipo de dados esperado (**`z.string()`**, **`z.number()`**, etc., mais na documentação da biblioteca), seguida de restrições mais específicas.
    c. Cada restrição pode ter sua própria mensagem de erro, possibilitando oferecer ao usuário *feedbacks* bastante específicos.
    d. O Zod se integra bem com outras bibliotecas de validação, por meio do método **`refine()`**, como foi usado para validar o CPF do cliente.
    e. A biblioteca fornece também validações prontas para os casos mais comuns, como a validação de *e-mail* do último campo.
14. Definidas as regras de validação por meio do *model*, precisamos alterar o *controller* de clientes de modo a aplicar as regras antes de inserções ou alterações. Para isso, no arquivo **`back-end/src/controllers/customers.js`**:
    a. Importe o *model* e o objeto de erros do Zod no topo to arquivo:

```
import prisma from '../database/client.js'
import Customer from '../models/Customer.js'
import { ZodError } from 'zod'

// (...código existente)
```

    b. Altere o método **`create()`** conforme segue. As partes mais relevantes estão destacadas:

```
controller.create = async function(req, res) {
  try {
```

```
    // Sempre que houver um campo que represente uma data,
    // precisamos garantir sua conversão para o tipo Date
    // antes de passá-lo ao Zod para validação
    if(req.body.birth_date) req.body.birth_date = new Date(req.body.birth_date)

    // Invoca a validação do modelo do Zod para os dados que
    // vieram em req.body
    Customer.parse(req.body)

    await prisma.customer.create({ data: req.body })

    // HTTP 201: Created
    res.status(201).end()
  }
  catch(error) {
    console.error(error)

    // Se for erro de validação do Zod, retorna
    // HTTP 422: Unprocessable Entity
    if(error instanceof ZodError) res.status(422).send(error.issues)

    // Senão, retorna o habitual HTTP 500: Internal Server Error
    else res.status(500).end()
  }
}
```

i. A validação propriamente dita é acionada pelo método **`parse()`** do *model*. A validação gera erros que forçam a execução a cair no bloco **`catch`**; tais erros podem ser identificados e separados de outros tipos de erro verificando se são instâncias da classe **`ZodError`**.

c. De modo semelhante, precisamos alterar também o método **`update()`**:

```
controller.update = async function(req, res) {
  try {

    // Sempre que houver um campo que represente uma data,
    // precisamos garantir sua conversão para o tipo Date
```

```
    // antes de passá-lo ao Zod para validação
    if(req.body.birth_date) req.body.birth_date = new Date(req.body.birth_date)

    // Invoca a validação do modelo do Zod para os dados que
    // vieram em req.body
    Customer.parse(req.body)

    await prisma.customer.update({
      where: { id: Number(req.params.id) },
      data: req.body
    })

    // Encontrou e atualizou ~> HTTP 204: No Content
    res.status(204).end()

  }
  catch(error) {
    console.error(error)

    // Não encontrou e não atualizou ~> HTTP 404: Not Found
    if(error?.code === 'P2025') res.status(404).end()

    // Erro do Zod ~> HTTP 422: Unprocessable Entity
    else if(error instanceof ZodError) res.status(422).send(error.issues)

    // Outros erros ~> HTTP 500: Internal Server Error
    else res.status(500).end()
  }
}
```

d. Para testar:

   i. No Projeto Vulcom, vá até o cadastro de clientes. Ao mesmo tempo, deixe abertas as Ferramentas de Desenvolvedor na aba "Network".

   ii. Tente fazer um cadastro, com informações inconsistentes (tente, por exemplo, colocar como data de nascimento uma data futura) e salve. Você deve receber uma mensagem de erro **`HTTP 422: Unprocessable Entity`**.

iii. Para ver as mensagens de erro específicas enviadas pelo *back-end*, clique sobre a requisição na aba "Network" e, em seguida, sobre a sub-aba "Response".

15. A validação do cadastro de clientes está funcionando, embora ainda seja difícil ver o *feedback*. Isso porque a validação feita no *back-end* é pensada para ser acionada quando a requisição vem de ferramentas de teste de API como o RapidAPI Client. Para um *feedback* mais imediato no *front-end*, este deve implementar também sua validação. É o que faremos na próxima aula.
16. Vamos terminar as atividades do dia com um *commit* + *push* no repositório do GitHub.
    a. Feche todos os terminais abertos usando o ícone da lixeira e abra um terminal novo. Nele, execute:

```
git add .
git commit -m "(01/11) Término do ataque de força bruta e início da implementação de validação de dados de usuário"
git push
```

    b. **IMPORTANTE: o Trabalho 2 da disciplina será constituído pelo conjunto de *commits* no repositório do aluno, que deve ser um *fork* do repositório do professor. Serão considerados, para fins avaliativos, apenas os *commits* feitos entre o dia da aula e o dia anterior à aula seguinte. Por exemplo, o *commit* de uma aula de sábado deve ser feito, no máximo, até a terça-feira seguinte.**

05/11

1. No VS Code, abra um terminal e use o botão para duplicar o terminal.
    a. No terminal da esquerda:

```
cd back-end
npm run dev
```

    b. No terminal da direita:

```
cd front-end
npm run dev
```

2. Acesse o *front-end* em http://localhost:5173.
3. Uma forma mais prática de vermos os erros de validação emitidos pelo Zod no *back-end* é desabilitar temporariamente o *middleware* de autenticação e efetuar uma requisição via RapidAPI Client.
    a. No arquivo **back-end/src/app.js**, comente a linha indicada:

```
// (...código existente...)
```

```
// Middleware de verificação do token de autorização
import auth from './middleware/auth.js'
// app.use(auth)

// (...código existente...)
```

b. Em seguida, crie uma nova requisição no RapidAPI Client com as especificações:
   i. Method: **POST**
   ii. URL: http://localhost:8080/customers
   iii. Body/JSON (varie o preenchimento dos campos para observar os diferentes tipos de erro):

```
{
  "name": "",
  "ident_document": "",
  "birth_date": null,
  "street_name": "",
  "house_number": "",
  "complements": "",
  "district": "",
  "municipality": "",
  "state": "",
  "phone": "",
  "email": ""
}
```

   iv. Clique sobre o botão "Send". Caso haja alguma inconsistência nos dados, a resposta será o status **HTTP 422: Unprocessable Entity**, com o detalhamento dos erros listados na resposta.

4. Feito esse teste, vamos reabilitar o *middleware* de autenticação no arquivo **back-end/src/app.js**:

```
// (...código existente...)

// Middleware de verificação do token de autorização
import auth from './middleware/auth.js'
app.use(auth)

// (...código existente...)
```

5. Precisamos habilitar a mesma validação feita no *back-end* também no *front-end*. Assim, vamos criar a pasta **`models`** dentro de **`front-end/src`** e copiar o arquivo **`back-end/src/models/Customer.js`** para lá.
    a. **IMPORTANTE: em um ambiente real de produção (um servidor rodando Linux), não seria feita uma cópia, e sim um *link* simbólico (*symlink*) da definição do *model* do *back-end* para o *front-end*. Assim, teríamos apenas uma cópia do arquivo no back-end, e o front-end se referindo a ele por meio do *symlink*.**
6. A validação no *front-end* será processada no arquivo **`front-end/src/pages/customer/CustomerForm.jsx`**.
    a. Primeiramente, vamos importar o *model* e o **`ZodError`** no topo do arquivo:

```
import Customer from '../../models/Customer.js'
import { ZodError } from 'zod'
```

    b. Em seguida, alteramos o método **`handleFormSubmit()`** para introduzir a chamada à validação e o processamento dos eventuais erros.

```
async function handleFormSubmit(e) {
   e.preventDefault()    // Evita o recarregamento da página
   // Exibir a tela de espera
   showWaiting(true)
   try {
      // Invoca a validação do Zod
      Customer.parse(customer)

      // Envia os dados para o back-end para criar um novo cliente
      // no banco de dados
      // Se houver parâmetro na rota, significa que estamos editando.
      // Portanto, precisamos enviar os dados ao back-end com o verbo PUT
      if(params.id) await myfetch.put(`/customers/${params.id}`, customer)

      // Senão, os dados serão enviados com o método POST para a criação de
      // um novo cliente
      else await myfetch.post('/customers', customer)

      // Deu certo, vamos exibir a mensagem de feedback que, quando fechada,
      // vai nos mandar de volta para a listagem de clientes
      notify('Item salvo com sucesso.', 'success', 4000, () => {
        navigate('..', { relative: 'path', replace: true })
      })
```

```
    }
    catch(error) {
      console.error(error)

      // Em caso de erro do Zod, preenchemos a variável de estado
      // inputErrors com os erros para depois exibir abaixo de cada
      // campo de entrada
      if(error instanceof ZodError) {
        const errorMessages = {}
        for(let i of error.issues) errorMessages[i.path[0]] = i.message
        setState({ ...state, inputErrors: errorMessages })
        notify('Há campos com valores inválidos. Verifique.', 'error')
      }
      else notify(error.message, 'error')
    }
    finally {
      showWaiting(false)
    }
  }
```

c. Os erros gerados pelo Zod serão exibidos abaixo de cada campo do cadastro na cor vermelha. Isso é feito por duas *props* já presentes em cada campo: **`error`** e **`helperText`**. Elas estão preparadas para serem exibidas apenas quando erros ocorrem. Veja exemplo abaixo:

```
// (...código existente...)

          <TextField
            name="name"
            label="Nome completo"
            variant="filled"
            required
            fullWidth
            autoFocus
            value={customer.name}
            onChange={handleFieldChange}
            error={inputErrors?.name}
            helperText={inputErrors?.name}
```

```
          />

// (...código existente...)
```

   d. Antes de efetivamente podermos testar a validação no *front-end*, precisamos instalar nele as mesmas bibliotecas que instalamaos para validação no *back-end*: o Zod e também **`cpf-cnpj-validator`**.
      i. Interrompa a execução do *front-end* teclando Ctrl+C no terminal correspondente.
      ii. Execute nesse terminal:
      ```
      npm install zod cpf-cnpj-validator
      npm run dev
      ```
7. Acesse o Projeto Vulcom (http://localhost:5173) e faça *login* (**`admin`**/**`Vulcom@DSM`**). Acesse o cadastro de clientes e tente inserir um novo cliente. Quaisquer inconsistências gerarão mensagens de erro em vermelho abaixo dos campos correspondentes.
8. Assim terminamos por hoje. Na próxima aula, construiremos o *model* de validação para a entidade **`Car`**.
9. Vamos terminar as atividades do dia com um *commit* + *push* no repositório do GitHub.
   a. Feche todos os terminais abertos usando o ícone da lixeira e abra um terminal novo. Nele, execute:
   ```
   git add .
   git commit -m "(05/11) Validação de Customer com Zod no front-end"
   git push
   ```
   b. IMPORTANTE: o Trabalho 2 da disciplina será constituído pelo conjunto de *commits* no repositório do aluno, que deve ser um *fork* do repositório do professor. Serão considerados, para fins avaliativos, apenas os *commits* feitos entre o dia da aula e o dia anterior à aula seguinte. Por exemplo, o *commit* de uma aula de sábado deve ser feito, no máximo, até a terça-feira seguinte.

08/11

1. No VS Code, abra um terminal e use o botão para duplicar o terminal.
   a. No terminal da esquerda:
   ```
   cd back-end
   npm run dev
   ```
   b. No terminal da direita:
   ```
   cd front-end
   ```

```
npm run dev
```

2. Acesse o *front-end* em http://localhost:5173.
3. Agora é com você. Desenvolva um *model* de validação do Zod para a entidade **`Car`**, com regras e mensagens de erro, segundo as especificações abaixo. Consulte a documentação da biblioteca, sempre que necessário.
    a. Campo **brand**: no mínimo 1 e no máximo 25 caracteres.
    b. Campo **`model`**: no mínimo 1 e no máximo 25 caracteres.
    c. Campo **`color`**: exatamente um dos seguintes: 'AMARELO', 'AZUL', 'BRANCO', 'CINZA', 'DOURADO', 'LARANJA', 'MARROM', 'PRATA', 'PRETO', 'ROSA', 'ROXO', 'VERDE', 'VERMELHO'.
    d. Campo **`year_manufacture`**: número inteiro, entre 1960 (inclusive) e o ano corrente (inclusive). Calcule o ano corrente a partir da data atual.
    e. Campo **`imported`**: deve ser um valor booleano (true ou false).
    f. Campo **`plates`**: deve ter exatamente 8 caracteres.
    g. Campo **`selling_date`**: data, PREENCHIMENTO OPCIONAL. Caso informada, não pode ser anterior à data de abertura da loja (20/03/2020) nem posterior à data de hoje.
    h. Campo **`selling_price`**: PREENCHIMENTO OPCIONAL. Caso informado, o valor deve estar entre R$ 5.000,00 e R$ 5.000.000,00.
4. Faça as modificações necessárias tanto no *back-end* quanto no *front-end* para efetivar o acionamento da validação antes do salvamento ou envio dos dados.
5. Efetue os testes necessários para verificar o funcionamento das validações.
6. Termine as atividades do dia com um *commit* + *push* no repositório do GitHub.
    a. Feche todos os terminais abertos usando o ícone da lixeira e abra um terminal novo. Nele, execute:
    ```
    git add .
    git commit -m "(08/11) Validação de Car com Zod no back-end e no front-end"
    git push
    ```
    b. IMPORTANTE: o Trabalho 2 da disciplina será constituído pelo conjunto de *commits* no repositório do aluno, que deve ser um *fork* do repositório do professor. Serão considerados, para fins avaliativos, apenas os *commits* feitos entre o dia da aula e o dia anterior à aula seguinte. Por exemplo, o *commit* de uma aula de sábado deve ser feito, no máximo, até a terça-feira seguinte.

12/11

FINALIZAÇÃO DO TRABALHO 2 (T2)

1. No arquivo **`front-end/src/ui/FooterBar.jsx`**, coloque seu nome completo e seu e-mail no lugar dos dados do professor.
2. Crie uma pasta na ***raiz do repositório*** chamada **`PRINTS TRABALHO 2`**.
3. Desabilite temporariamente o *middleware* de autenticação no *back-end* (conforme já feito anteriormente). No RapidAPI Client, gere uma requisição para inserção de um veículo, com dados errôneos.
    a. Tire um *print* da tela, com o terminal aberto, mostrando a resposta da requisição com o código **`422: Unprocessable Entity`**, mais as mensagens de erro retornadas pelo Zod. Salve o *print* na pasta criada no item 2).
    b. Reabilite o *middleware* de autenticação no *back-end*.
4. No cadastro de veículos do Projeto Vulcom, tire um *print* da tela com mensagens de erro quando dados são informados em desacordo com as regras de validação definidas. Salve o *print* na pasta criada no item 2).
5. Entrega:
    a. Feche todos os terminais abertos usando o ícone da lixeira e abra um terminal novo.
    b. Faça *commit + pus*h de suas alterações para o GitHub:

    ```
    git add .
    git commit -m "TRABALHO 2"
    git push
    ```

    c. Acesse github.com e faça *login*.
    d. Acesse github.com/<SEU_USUARIO>/sa-dsm-2025-2 e clique sobre a aba "Pull requests". Na página que será aberta, clique sobre o botão "New pull request" e informe:
        i. no título: TRABALHO 2;
        ii. na descrição: seu nome completo.
    e. Conclua clicando sobre o botão "Create pull request".
6. **Prazo de entrega: até 23/11 (domingo)**. A aula de hoje (12/11) e a da próxima quarta-feira (19/11) estarão livres para a finalização do trabalho.