Code Documentatie - Fetch Custom Fields Functionaliteit

Overzicht

Deze documentatie beschrijft de implementatie van het ophalen van custom fields in de Fleet Management applicatie, waarbij we kijken naar zowel de frontend component als de bijbehorende backend endpoint.

Frontend: fetchCustomFields Functie

```
const fetchCustomFields = async () => {
    if (!tenantId) return;

    try {
      const response = await
axios.get(`http://localhost:5054/api/tenant/${tenantId}`);
      const customFieldsData = response.data.customFields?.$values || [];
      console.log('Custom fields data:', customFieldsData);

      // Ensure proper casing and filter out invalid custom fields
      const validCustomFields = customFieldsData
        .filter(field => !field.IsDeleted)
        .map(field => ({
          CustomFieldId: field.customFieldId || field.CustomFieldId,
          FieldName: field.fieldName || field.FieldName,
          ValueType: field.valueType || field.ValueType,
          IsDeleted: field.isDeleted || field.IsDeleted
        }));

      setCustomFields(validCustomFields);
      setError('');
    } catch (error) {
      console.error('Error fetching custom fields:', error);
      setError('Failed to load custom fields. Please try again later.');
      setCustomFields([]);
    }
};
```

Gedetailleerde Uitleg

1. **Functie Doel**
    - Deze functie is verantwoordelijk voor het ophalen en verwerken van custom fields data voor een specifieke tenant
    - Het is een asynchrone functie die communiceert met de backend API
    - Zorgt voor de juiste verwerking van de data voordat deze wordt weergegeven in de gebruikersinterface

2. **Initiële Controle**
    - `if (!tenantId) return;`
    - Voorkomt onnodige API-aanroepen als er geen tenant ID beschikbaar is
    - Dient als beveiliging voor data-integriteit
    - Voorkomt foutmeldingen door ontbrekende tenant ID

3. **API Aanroep**
    - Gebruikt axios om een GET-verzoek te doen naar het tenant endpoint
    - De URL bevat het tenant ID om tenant-specifieke data op te halen
    - `response.data.customFields?.$values` gebruikt optional chaining om veilig met mogelijke null-waarden om te gaan
    - Vangt eventuele fouten op tijdens het ophalen van de data

4. **Data Verwerking**
    - Filtert verwijderde custom fields uit met `.filter(field => !field.IsDeleted)`
    - Zet de data om naar een consistent formaat met hoofdletters/kleine letters
    - Gebruikt OR-operatoren (`||`) om zowel camelCase als PascalCase eigenschappen te ondersteunen
    - Standaardiseert de data voor gebruik in de frontend
    - Zorgt voor consistente naamgeving van eigenschappen

5. **State Beheer**
    - Werkt de component state bij met de verwerkte custom fields
    - Wist eventuele eerdere foutmeldingen bij succes
    - Zet een lege array en foutmelding bij mislukte aanvragen
    - Zorgt voor een goede gebruikerservaring door duidelijke feedback

Backend: GetTenantById Endpoint

```
[HttpGet("{tenantId}")]
public async Task<ActionResult<object>> GetTenantById(int tenantId)
{
    Console.WriteLine($"Fetching tenant with ID: {tenantId}");
    try
    {
        var tenant = await repo.GetTenantByIdAsync(tenantId);
        if (tenant == null)
        {
            return NotFound();
        }

        var response = new
        {
            TenantId = tenant.TenantId,
            CompanyName = tenant.CompanyName,
            VATNumber = tenant.VATNumber,
            IsDeleted = tenant.IsDeleted,
```

```
            Users = (tenant.Users ?? Enumerable.Empty<User>()).Select(u => new
            {
                UserId = u.UserId,
                Email = u.Email,
                IsDeleted = u.IsDeleted
            }),
            CustomFields = (tenant.CustomFields ?? Enumerable.Empty<CustomField>
()).
                .Where(cf => cf != null && !cf.IsDeleted)
                .Select(cf => new
                {
                    CustomFieldId = cf.CustomFieldId,
                    FieldName = cf.FieldName,
                    ValueType = cf.ValueType,
                    IsDeleted = cf.IsDeleted
                }),
            Vehicles = (tenant.Vehicles ?? Enumerable.Empty<Vehicle>())
                .Where(v => v != null && !v.IsDeleted)
                .Select(v => new
                {
                    VehicleId = v.VehicleId,
                    Brand = v.Brand,
                    Model = v.Model,
                    LicensePlate = v.LicensePlate,
                    ChassisNumber = v.ChassisNumber,
                    FuelTypeId = v.FuelTypeId,
                    Color = v.Color,
                    VehicleTypeId = v.VehicleTypeId,
                    TenantId = v.TenantId,
                    IsDeleted = v.IsDeleted,
                    CustomFieldValues = (v.CustomFieldValues ??
Enumerable.Empty<CustomFieldValue>())
                        .Where(cfv => cfv != null && !cfv.IsDeleted)
                        .Select(cfv => new
                        {
                            CustomFieldValueId = cfv.CustomFieldValueId,
                            VehicleId = cfv.VehicleId,
                            CustomFieldId = cfv.CustomFieldId,
                            Value = cfv.Value,
                            ValueType = cfv.CustomField?.ValueType,
                            IsDeleted = cfv.IsDeleted,
                            CustomField = cfv.CustomField != null ? new
                            {
                                CustomFieldId = cfv.CustomField.CustomFieldId,
                                FieldName = cfv.CustomField.FieldName,
                                ValueType = cfv.CustomField.ValueType,
                                IsDeleted = cfv.CustomField.IsDeleted
                            } : null
                        })
                })
        };

        return Ok(response);
    }
```

```
    catch (Exception ex)
    {
        Console.WriteLine(ex);
        return BadRequest();
    }
}
```

Gedetailleerde Uitleg

1. **Endpoint Definitie**
    - HTTP GET endpoint die een tenant ID als route parameter accepteert
    - Retourneert een complex object met tenant data en gerelateerde entiteiten
    - Gebruikt async/await voor niet-blokkerende database operaties
    - Zorgt voor efficiënte verwerking van grote datasets
2. **Foutafhandeling**
    - Omvat de hele operatie in een try-catch blok voor veilige uitvoering
    - Logt uitzonderingen naar de console voor debugging doeleinden
    - Retourneert passende HTTP statuscodes:
        - 200 (OK) voor succesvolle verzoeken
        - 404 (NotFound) wanneer tenant niet bestaat
        - 400 (BadRequest) voor andere fouten
    - Zorgt voor duidelijke foutmeldingen naar de client
3. **Data Ophalen**
    - Maakt gebruik van het repository pattern (`repo.GetTenantByIdAsync`) voor data toegang
    - Voert null-check uit op tenant voordat verdere verwerking plaatsvindt
    - Logt de ophaaloperatie voor monitoring doeleinden
    - Scheidt data toegang van business logica
4. **Response Structuur**
    - Creëert een anoniem object met een specifieke structuur
    - Bevat basis tenant informatie (ID, bedrijfsnaam, BTW-nummer)
    - Omvat drie hoofdcollecties:
        - Gebruikers (Users)
        - Aangepaste Velden (Custom Fields)
        - Voertuigen (Vehicles) met geneste custom field waarden
    - Zorgt voor een georganiseerde en voorspelbare data structuur
5. **Null-waarde Afhandeling en Filtering**
    - Gebruikt null-coalescing operator (`??`) voor veilige collectie handling
    - Filtert verwijderde records met Where clausules
    - Voert null-checks uit op geneste objecten
    - Voorkomt null-reference exceptions

6. **Data Projectie**
    - Gebruikt LINQ Select voor het maken van nieuwe anonieme objecten
    - Includeert alleen noodzakelijke eigenschappen in de response
    - Behoudt de data hiërarchie terwijl de payload grootte wordt beperkt
    - Handelt geneste relaties af (bijv. Vehicle -> CustomFieldValues -> CustomField)
    - Optimaliseert de response grootte

7. **Beveiliging en Data Integriteit**
    - Filtert verwijderde records op elk niveau
    - Exposeert alleen noodzakelijke data velden
    - Handhaaft data integriteit door grondige null-checking
    - Voorkomt het lekken van gevoelige informatie

Code Documentatie - Add Custom Fields Functionaliteit

Frontend: handleAddCustomField Functie

```
const handleAddCustomField = async () => {
    if (!customFieldName.trim()) {
      setError('Please enter a field name');
      return;
    }

    try {
      const newCustomField = {
        TenantId: parseInt(tenantId),
        ValueType: getValueTypeString(selectedValueType),
        FieldName: customFieldName.trim(),
        IsDeleted: false,
        CustomFieldValues: [],
        Tenant: {
          TenantId: parseInt(tenantId),
          CompanyName: "",
          VATNumber: "",
          IsDeleted: false,
          Users: [],
          Vehicles: [],
          CustomFields: []
        }
      };

      console.log('Sending custom field data:', newCustomField);

      const response = await axios.post(
        `http://localhost:5054/api/tenant/${tenantId}/customfields`,
        newCustomField
```

```
        );

        if (response.status === 200 || response.status === 201) {
          await fetchCustomFields();
          setCustomFieldName('');
          setError('');
        }
      } catch (error) {
        console.error('Error adding custom field:', error);
        if (error.response) {
          console.error('Error details:', error.response.data);
          setError(`Failed to add custom field: ${error.response.data}`);
        } else {
          setError('Failed to add custom field. Please try again.');
        }
      }
  };
```

Gedetailleerde Uitleg Frontend

1. **Validatie**
   - Controleert of er een veldnaam is ingevoerd
   - Voorkomt het verzenden van lege waardes naar de backend
   - Geeft duidelijke feedback aan de gebruiker bij ontbrekende data
2. **Object Constructie**
   - Bouwt een nieuw custom field object met alle benodigde eigenschappen
   - Zet het tenant ID om naar een integer met `parseInt()`
   - Bepaalt het waardetype via de `getValueTypeString` helper functie
   - Initialiseert lege arrays en objecten voor gerelateerde data
3. **API Communicatie**
   - Stuurt een POST-verzoek naar het custom fields endpoint
   - Logt de verzonden data voor debugging doeleinden
   - Wacht op een succesvolle response (status 200 of 201)
4. **Success Handling**
   - Roept `fetchCustomFields()` aan om de lijst te verversen
   - Reset het invoerveld voor de veldnaam
   - Wist eventuele foutmeldingen
5. **Error Handling**
   - Vangt fouten op tijdens het API-verzoek
   - Logt gedetailleerde foutinformatie
   - Toont gebruiksvriendelijke foutmeldingen
   - Maakt onderscheid tussen API-fouten en andere fouten

Backend: AddCustomField Endpoint

```
[HttpPost("{tenantId}/customfields")]
public async Task<IActionResult> AddCustomField(int tenantId, [FromBody]
CustomField customField)
{
    if (customField == null)
    {
        return BadRequest("Custom field data is null");
    }

    try
    {
        var tenant = await repo.GetTenantByIdAsync(tenantId);
        if (tenant == null)
        {
            return NotFound($"Tenant with ID {tenantId} not found");
        }

        customField.TenantId = tenantId;
        customField.IsDeleted = false;
        customField.CustomFieldValues = new List<CustomFieldValue>();
        customField.Tenant = null;

        var addedField = await repo.AddCustomFieldAsync(customField);
        return Ok(addedField);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex);
        return BadRequest($"Something went wrong while adding the custom field:
{ex.Message}");
    }
}
```

Gedetailleerde Uitleg Backend

1. **Endpoint Definitie**
   - HTTP POST endpoint voor het toevoegen van custom fields
   - Accepteert tenant ID als route parameter
   - Verwacht een CustomField object in de request body
   - Gebruikt [FromBody] attribuut voor automatische model binding
2. **Input Validatie**
   - Controleert of het custom field object niet null is
   - Verifieert het bestaan van de tenant
   - Retourneert duidelijke foutmeldingen bij ongeldige input
3. **Data Voorbereiding**

- Koppelt het custom field aan de juiste tenant
- Initialiseert de IsDeleted status op false
- Maakt een lege lijst voor custom field values
- Verwijdert de circulaire referentie naar tenant

4. **Database Operatie**

- Gebruikt het repository pattern voor data persistentie
- Voert de toevoeging asynchroon uit
- Retourneert het toegevoegde veld bij succes

5. **Foutafhandeling**

- Vangt alle exceptions tijdens de uitvoering
- Logt de foutdetails voor debugging
- Retourneert gebruiksvriendelijke foutmeldingen
- Gebruikt appropriate HTTP status codes:
  - 200 (OK) bij succes
  - 400 (BadRequest) bij ongeldige input
  - 404 (NotFound) als tenant niet bestaat

Code Documentatie - UpdateCustomField Functionaliteit

Frontend: handleUpdateCustomField Functie

```
const handleUpdateCustomField = async (updatedCustomField) => {
    try {
      const completeCustomField = {
        CustomFieldId: updatedCustomField.CustomFieldId,
        TenantId: parseInt(tenantId),
        FieldName: updatedCustomField.FieldName,
        ValueType: updatedCustomField.ValueType,
        IsDeleted: false,
        CustomFieldValues: [],
        Tenant: {
          TenantId: parseInt(tenantId),
          CompanyName: "",
          VATNumber: "",
          IsDeleted: false,
          Users: [],
          Vehicles: [],
          CustomFields: []
        }
      };

      const response = await axios.put(

`http://localhost:5054/api/tenant/${tenantId}/customfields/${updatedCustomField.Cu
```

```
stomFieldId}`,
        completeCustomField
      );

      if (response.status === 200) {
        await fetchCustomFields(); // Refresh the list
        setError('');
      }
    } catch (error) {
      console.error('Error updating custom field:', error);
      setError('Failed to update custom field. Please try again.');
    }
};
```

Gedetailleerde Uitleg Frontend

1. **Object Voorbereiding**
   - Creëert een volledig custom field object voor de update
   - Behoudt het originele CustomFieldId voor identificatie
   - Zet het tenant ID om naar een integer
   - Initialiseert lege arrays voor gerelateerde data
   - Zorgt voor een consistente datastructuur

2. **API Communicatie**
   - Gebruikt axios voor een PUT-verzoek naar het update endpoint
   - Construeert de URL met tenant ID en custom field ID
   - Stuurt het complete object mee in de request body
   - Wacht op een succesvolle response (status 200)

3. **Success Handling**
   - Vernieuwt de lijst met custom fields via `fetchCustomFields()`
   - Wist eventuele foutmeldingen uit de state
   - Zorgt voor directe UI updates na succesvolle wijziging

4. **Error Handling**
   - Vangt fouten op tijdens het update proces
   - Logt fouten voor debugging doeleinden
   - Toont gebruiksvriendelijke foutmeldingen
   - Handhaaft de applicatie state bij fouten

Backend: UpdateCustomField Endpoint

```
[HttpPut("{tenantId}/customfields/{customFieldId}")]
public async Task<IActionResult> UpdateCustomField(int tenantId, int
customFieldId, [FromBody] CustomField customField)
{
```

```
    if (customField == null || customField.CustomFieldId != customFieldId)
    {
        return BadRequest("Invalid custom field data");
    }

    try
    {
        var tenant = await repo.GetTenantByIdAsync(tenantId);
        if (tenant == null)
        {
            return NotFound($"Tenant with ID {tenantId} not found");
        }

        var existingCustomField = tenant.CustomFields.FirstOrDefault(cf =>
cf.CustomFieldId == customFieldId);
        if (existingCustomField == null)
        {
            return NotFound($"Custom field with ID {customFieldId} not found");
        }

        var updatedField = await repo.UpdateCustomFieldAsync(customField);
        return Ok(updatedField);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex);
        return BadRequest($"Something went wrong while updating the custom field:
{ex.Message}");
    }
}
```

Gedetailleerde Uitleg Backend

1. **Endpoint Definitie**
   - HTTP PUT endpoint voor het bijwerken van custom fields
   - Accepteert tenant ID en custom field ID als route parameters
   - Verwacht een CustomField object in de request body
   - Gebruikt [FromBody] attribuut voor model binding

2. **Input Validatie**
   - Controleert of het custom field object niet null is
   - Verifieert of het CustomFieldId overeenkomt met de route parameter
   - Voorkomt onbedoelde updates van verkeerde records
   - Zorgt voor data integriteit

3. **Tenant en Veld Verificatie**
   - Controleert het bestaan van de tenant
   - Zoekt het bestaande custom field binnen de tenant context
   - Gebruikt FirstOrDefault voor efficiënte zoekopdracht

- Voorkomt updates van niet-bestaande velden

4. **Database Operatie**
   - Voert de update asynchroon uit via het repository pattern
   - Retourneert het bijgewerkte veld bij succes
   - Handhaaft data consistentie tijdens de update
   - Scheidt database logica van de controller

5. **Foutafhandeling**
   - Vangt alle exceptions tijdens de uitvoering
   - Logt foutdetails voor debugging
   - Retourneert duidelijke foutmeldingen
   - Gebruikt passende HTTP status codes:
     - 200 (OK) bij succesvolle update
     - 400 (BadRequest) bij ongeldige input
     - 404 (NotFound) bij niet-bestaande tenant of veld

Code Documentatie - Login Functionaliteit

Frontend: handleLogin Functie

```
const handleLogin = async (e) => {
    e.preventDefault();
    const username = document.getElementById('username').value;
    const password = document.getElementById('password').value;

    try {
      const response = await axios.post('http://localhost:5054/api/user/login', {
        Email: username,
        Password: password
      });

      const data = response.data;
      console.log("Response data:", data);

      if (data && data.success) {
        const userData = data.user;
        console.log("User data:", userData);
        const user = {
          username: userData.userName,
          userId: userData.userId,
          clearanceId: userData.clearanceId,
          tenantId: userData.tenantId
        };
        console.log("Created user object:", user);
        login(user);
        console.log("After login call");
```

```
        console.log("Attempting navigation with clearanceId:",
userData.clearanceId);

        switch (userData.clearanceId) {
          case 1:
            console.log("Navigating to superadmin");
            navigate('/superadmin/dashboard');
            break;
          case 2:
            console.log("Navigating to bedrijfsadmin");
            navigate('/bedrijfsadmin/dashboard');
            break;
          case 3:
            console.log("Navigating to werknemer");
            navigate('/werknemer/dashboard');
            break;
          default:
            console.log("Navigating to default");
            navigate('/');
            break;
        }
        console.log("After navigation attempt");
      } else {
        setErrorMessage('Invalid email or password');
      }
    } catch (error) {
      console.error('Error during login:', error);
      setErrorMessage('Failed to login');
    }
};
```

Gedetailleerde Uitleg Frontend

1. **Form Handling**
   - Voorkomt standaard form submit gedrag met `e.preventDefault()`
   - Haalt gebruikersnaam en wachtwoord op uit de form elementen
   - Zorgt voor een gecontroleerde form submission

2. **API Communicatie**
   - Stuurt een POST-verzoek naar het login endpoint
   - Verzendt email en wachtwoord in het juiste formaat
   - Wacht op de server response
   - Bevat uitgebreide logging voor debugging doeleinden

3. **Response Verwerking**
   - Controleert op succesvolle login via `data.success`
   - Extraheert gebruikersgegevens uit de response
   - Creëert een gestandaardiseerd user object voor de applicatie
   - Bewaart de login status via de `login()` functie

4. **Rolgebaseerde Navigatie**
    - Gebruikt `clearanceId` voor het bepalen van de juiste route
    - Implementeert verschillende routes voor verschillende gebruikersrollen:
        - 1: Superadmin dashboard
        - 2: Bedrijfsadmin dashboard
        - 3: Werknemer dashboard
    - Valt terug op de homepage bij onbekende rollen

5. **Error Handling**
    - Vangt mislukte login pogingen af
    - Toont gebruiksvriendelijke foutmeldingen
    - Logt fouten voor debugging doeleinden
    - Handhaaft de applicatie state bij fouten

Backend: LoginUser Endpoint

```
[HttpPost("login")]
public async Task<ActionResult<UserDTO>> LoginUser([FromBody] LoginDTO
loginRequest)
{
    // Validate the request
    if (loginRequest == null || string.IsNullOrEmpty(loginRequest.Email) ||
string.IsNullOrEmpty(loginRequest.Password))
    {
        return BadRequest("Email and password are required.");
    }

    // Query the database for the user
    var user = await _repo.LogInUserAsync(loginRequest.Email,
loginRequest.Password);
    if (user == null)
    {
        return Unauthorized("Invalid email or password.");
    }

    // Ensure Clearance is not null
    if (user.Clearance == null)
    {
        return BadRequest("User clearance information is missing.");
    }

    // Create and return the UserDTO
    UserDTO userDTO = new UserDTO(user.UserId, user.TenantId, user.UserName,
user.Email, user.Clearance.ClearanceId);
    return Ok(new { success = true, user = userDTO });
}
```

Gedetailleerde Uitleg Backend

1. **Endpoint Definitie**
    - HTTP POST endpoint voor gebruikersauthenticatie
    - Verwacht een LoginDTO object met email en wachtwoord
    - Retourneert een UserDTO bij succesvolle authenticatie
    - Gebruikt [FromBody] voor automatische model binding
2. **Request Validatie**
    - Controleert of het loginRequest object niet null is
    - Valideert de aanwezigheid van email en wachtwoord
    - Voorkomt onnodige database queries bij ontbrekende data
    - Retourneert duidelijke foutmeldingen bij invalide requests
3. **Authenticatie**
    - Gebruikt het repository pattern voor gebruikersverificatie
    - Roept `LogInUserAsync` aan met de verstrekte credentials
    - Controleert op null response voor niet-gevonden gebruikers
    - Retourneert Unauthorized bij ongeldige credentials
4. **Autorisatie Verificatie**
    - Controleert de aanwezigheid van clearance informatie
    - Voorkomt toegang voor gebruikers zonder juiste rechten
    - Zorgt voor veilige toegangscontrole
    - Retourneert BadRequest bij ontbrekende clearance
5. **Response Constructie**
    - Creëert een UserDTO met alleen noodzakelijke gebruikersinformatie
    - Includeert success flag in de response
    - Verbergt gevoelige gebruikersgegevens
    - Zorgt voor een consistente response structuur

Tenant Data Flow na Login

1. Login en TenantId Doorgifte

```
// AuthContext.js
const login = (userData) => {
    setUser(userData);
    localStorage.setItem('loggedInUser', JSON.stringify(userData));
    setIsAuthenticated(true);
};

// LoginPage.js
if (data && data.success) {
    const userData = data.user;
    const user = {
        username: userData.userName,
```

```
            userId: userData.userId,
            clearanceId: userData.clearanceId,
            tenantId: userData.tenantId  // TenantId wordt hier opgeslagen
        };
        login(user);  // User data wordt opgeslagen in AuthContext en localStorage
    }
```

2. Tenant Data Ophalen

```
// BedrijfsAdminDashboard.js
useEffect(() => {
    if (!tenantId) return;

    const fetchTenantInfo = async () => {
      try {
        const response = await
axios.get(`http://localhost:5054/api/tenant/${tenantId}`);
        console.log('Tenant info:', response.data);
        setTenantInfo(response.data);

        // Filter out deleted users
        const users = response.data.users?.$values || [];
        const activeUsers = users.filter(user => !user.isDeleted);
        setUserData(activeUsers);

        // Fetch fuel types from the API
        const fuelTypesResponse = await
axios.get('http://localhost:5054/api/vehicles/fueltypes');
        const fuelTypesData = fuelTypesResponse.data.$values || [];
        const fuelTypesMap = fuelTypesData.reduce((acc, type) => {
          acc[type.fuelTypeId] = type.fuelTypeName;
          return acc;
        }, {});

        // Extract vehicles and filter out deleted ones
        const vehicles = response.data.vehicles?.$values || [];
        const activeVehicles = vehicles.filter(vehicle => !vehicle.isDeleted);
        setVehicleData(activeVehicles);
      } catch (error) {
        console.error('Error fetching tenant info:', error);
      }
    };

    fetchTenantInfo();
}, [tenantId]);
```

Gedetailleerde Uitleg van het Proces

1. **Login Flow**

- Bij succesvolle login wordt user data ontvangen van de backend
- TenantId wordt meegestuurd in de UserDTO
- User data (inclusief tenantId) wordt opgeslagen in:
  - AuthContext state
  - localStorage voor persistentie
- **Beveiligingsprobleem:** Geen verificatie van tenantId op backend niveau

2. **AuthContext Beheer**

- Bewaart de ingelogde gebruiker's gegevens
- Maakt tenantId beschikbaar voor alle componenten
- Handhaaft de login status
- **Beveiligingsprobleem:** Vertrouwt volledig op frontend data

3. **Data Ophalen**

- `fetchTenantInfo` wordt aangeroepen met opgeslagen tenantId
- Haalt alle tenant-gerelateerde data op in één request
- Filtert verwijderde records client-side
- **Beveiligingsprobleem:** Geen rolgebaseerde filtering op backend

Code Documentatie - Fouten bij de Clearance Implementatie

Overzicht van het Probleem

De applicatie moest oorspronkelijk drie verschillende toegangsniveaus ondersteunen:

1. Superadmin: Overzicht van alle bedrijven, hun admins, bedrijfsnaam en BTW-nummer
2. Bedrijfsadmin: Toegang tot alleen data van eigen bedrijf
3. Werknemer: Alleen inzage in voertuigen van eigen bedrijf

Geïdentificeerde Problemen

1. Ontbrekende Autorisatie Attributen

- **Huidige Situatie:**
  - Beperkt gebruik van `[Authorize]` attributen
  - Alleen enkele endpoints in TenantController hebben `[Authorize(Roles = "Admin")]`
  - Geen systematische rolgebaseerde toegangscontrole
- **Impact:**
  - Endpoints zijn onvoldoende beveiligd
  - Geen garantie dat gebruikers alleen toegang hebben tot hun eigen data
  - Mogelijke beveiligingsrisico's

2. Incorrecte Rol-mapping

- **Huidige Situatie:**
  - Clearance IDs worden wel opgeslagen:
    - 1: Superadmin
    - 2: Bedrijfsadmin
    - 3: Werknemer
  - Geen vertaling van clearance naar daadwerkelijke systeem rollen
  - Alleen frontend routing op basis van clearanceId
- **Impact:**
  - Autorisatie wordt alleen op frontend niveau afgehandeld
  - Backend heeft geen kennis van gebruikersrollen
  - Eenvoudig te omzeilen beveiliging

3. Ontbrekende Data Filtering

- **Huidige Situatie:**
  - Geen filtering van data op basis van gebruikersrol
  - Alle data wordt ongeacht rol teruggegeven
  - Geen tenant-specifieke toegangscontroles
- **Impact:**
  - Werknemers kunnen potentieel alle bedrijfsdata zien
  - Bedrijfsadmins kunnen data van andere bedrijven inzien
  - Geen data-isolatie tussen tenants

Voorgestelde Oplossingen

1. Implementatie van Correcte Autorisatie

```
// Correcte attributen per rol
[Authorize(Roles = "Superadmin")]
[Authorize(Roles = "BedrijfsAdmin")]
[Authorize(Roles = "Werknemer")]

// Voorbeeld van een beveiligde endpoint
[Authorize(Roles = "Superadmin,BedrijfsAdmin")]
public async Task<ActionResult<IEnumerable<Vehicle>>> GetVehicles(int tenantId)
{
    // Implementatie
}
```

2. Rol-mapping Systeem

```
// In LoginUser endpoint
private string MapClearanceToRole(int clearanceId)
```

```
{
    return clearanceId switch
    {
        1 => "Superadmin",
        2 => "BedrijfsAdmin",
        3 => "Werknemer",
        _ => throw new Exception("Ongeldige clearance")
    };
}
```

3. Data Filtering Implementatie

```
// Voorbeeld van tenant-specifieke filtering
public async Task<ActionResult<IEnumerable<Vehicle>>> GetVehicles(int tenantId)
{
    // Basis autorisatie check
    if (User.IsInRole("Werknemer") && userTenantId != tenantId)
    {
        return Forbid();
    }

    var vehicles = await repo.GetVehiclesAsync(tenantId);

    // Rol-specifieke filtering
    if (!User.IsInRole("Superadmin"))
    {
        vehicles = vehicles.Where(v => v.TenantId == userTenantId);
    }

    return Ok(vehicles);
}
```

Best Practices voor Correcte Implementatie

1. Systematische Autorisatie

- Implementeer autorisatie op alle endpoints
- Gebruik role-based access control (RBAC)
- Voeg tenant-validatie toe aan alle relevante endpoints

2. Data Isolatie

- Filter data op basis van gebruikersrol en tenant
- Implementeer data access layers met ingebouwde filtering
- Voeg extra validatie toe voor cross-tenant requests

3. Security Headers

- Implementeer proper token-based authentication

- Voeg security headers toe aan alle responses
- Valideer alle inkomende requests op tenant-niveau

4. Logging en Monitoring

- Log alle toegangspogingen
- Monitor ongeautoriseerde toegangspogingen
- Implementeer audit trails voor gevoelige operaties

Impact van de Huidige Implementatie

Beveiligingsrisico's

- Onvoldoende scheiding tussen verschillende gebruikersrollen
- Mogelijke data lekkage tussen verschillende tenants
- Onvoldoende logging van toegangspogingen

Gebruikerservaring

- Frontend routing werkt wel, maar biedt schijnveiligheid
- Geen echte data-isolatie tussen verschillende tenants
- Inconsistente toegangscontroles

Onderhoud

- Moeilijk te debuggen autorisatieproblemen
- Geen centrale plek voor rolbeheer
- Lastig uit te breiden met nieuwe rollen of permissies

4. **Huidige Problemen**
    - Backend vertrouwt blindelings de meegestuurde tenantId
    - Geen verificatie of gebruiker recht heeft op opgevraagde tenant data
    - Alle data wordt ongeacht rol teruggestuurd
    - Client-side filtering is onveilig
5. **Gewenste Situatie**
    - Backend valideert of gebruiker toegang heeft tot opgevraagde tenant
    - Data wordt gefilterd op basis van gebruikersrol
    - Aparte endpoints voor verschillende autorisatieniveaus
    - Proper token-based authentication met rol-informatie
6. **Impact**
    - Gebruikers kunnen potentieel data van andere tenants zien
    - Geen echte data-isolatie tussen tenants
    - Frontend routing biedt schijnveiligheid
    - Gevoelige data is onvoldoende beschermd