Vamos reorganizar a estrutura do banco de dados e a lógica da aplicação para torná-la mais eficiente e evitar duplicação.

Objetivos:

1. Cada loja deve ter seus próprios dados.
2. As rotas devem ser organizadas por funcionalidade.
3. Evitar duplicação de código e dados.

Estrutura sugerida para o banco de dados:

- Loja (Store)
  - codigo: String (único)
  - nome: String
  - cidade: String
- Auditoria (Audit)
  - loja: ObjectId (ref: 'Store')
  - tipo: String (etiqueta, presenca, ruptura)
  - data: Date
  - usuario: ObjectId (ref: 'User') [opcional, se for importante ter usuários separados por loja?]
  - itens: [{
  - codigo: String,
  - produto: String,
  - local: String,
  - situacao: String, // Atualizado, Não lido, etc.
  - estoque: String,
  - // ... outros campos conforme o tipo de auditoria
  - }]
- User (User)
  - id: String (pode ser o mesmo id do sistema da empresa)
  - nome: String
  - lojas: [ObjectId] (ref: 'Store') [se um usuário pode ter acesso a múltiplas lojas]
  - auditorias: [ObjectId] (ref: 'Audit') [opcional, para rastrear as auditorias que o usuário realizou]
- Planilha (Spreadsheet)
  - loja: ObjectId (ref: 'Store')
  - tipo: String (etiqueta, presenca, ruptura)
  - data: Date
  - nomeArquivo: String
  - totalItens: Number
  - totalItensLidos: Number

No entanto, note que atualmente temos modelos separados para Setor, Ruptura, Presenca, etc.
Uma abordagem mais unificada seria ter um único modelo de Auditoria com um campo 'tipo' e um schema flexível para os itens.

Considerando a complexidade atual, talvez seja melhor manter modelos separados para cada tipo de auditoria, mas com uma estrutura comum.

Estrutura de pastas e rotas:

- routes/
  - index.js (ou server.js) - centraliza todas as rotas
  - lojas.js (para obter listagem de lojas e selecionar loja)
  - upload.js (upload de planilhas - unificado para os três tipos, mas com tratamento específico)
  - estatisticas.js (para dados de estatísticas)
  - ranking.js (para ranking de usuários)
  - relatorios.js (para relatórios)

Melhorias:

1. Middleware de loja: em todas as rotas que precisam da loja, usar um middleware para verificar a loja (via header, session, ou parâmetro) e anexar ao request.
2. Modelos:
   - Store: para lojas
   - Audit: modelo base para auditorias (pode ser estendido ou não)
   - User: para usuários
3. Rotas de upload: processar o arquivo e salvar na coleção correspondente (etiqueta, presenca, ruptura) com o campo loja.
4. Rotas de relatórios/estatísticas: sempre filtrar pela loja.
5. Session: manter a loja selecionada na session do usuário (se for um sistema com login) ou usar tokens com a loja embutida.
6. Frontend: em todas as requisições, enviar o código da loja no header (x-loja).

Passos para reestruturar:

1. Criar modelo Store (se não existir).
2. Modificar os modelos de auditoria (Setor, Ruptura, Presenca) para incluir campo loja (ObjectId ref Store).
3. Modificar as rotas de upload para extrair a loja do header e salvar o ObjectId da loja no documento.
4. Modificar as rotas de relatórios, estatísticas, ranking para filtrar pela loja (usando o header x-loja).

No entanto, note que o sistema atual já tem um modelo de loja no frontend e no store do Pinia. Precisamos conectar isso.

Como a aplicação é grande, talvez seja melhor fazer uma transição gradual.

Alternativa: Manter a estrutura atual, mas garantir que todas as operações de banco de dados incluam o campo loja (código string) e modificar as consultas para usar esse campo.

No seu caso, como você está recriando do zero, vamos planejar:

1. Modelos:
   - Store:
   - const storeSchema = new mongoose.Schema({
   - codigo: { type: String, required: true, unique: true },
   - nome: { type: String, required: true },
   - cidade: String,
   - ativa: { type: Boolean, default: true }
   - });
   - User:
   - const userSchema = new mongoose.Schema({

id: String,
nome: String,
email: String,
lojas: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Store' }],
// ... outros campos
});
AuditBase (campos comuns a todos os tipos de auditoria):
const auditBaseSchema = new mongoose.Schema({
loja: { type: mongoose.Schema.Types.ObjectId, ref: 'Store', required: true },
data: { type: Date, required: true },
usuario: String, // ou ObjectId ref User
tipo: { type: String, enum: ['etiqueta', 'presenca', 'ruptura'], required: true }
}, { strict: false }); // strict: false para permitir campos adicionais
Em vez de vários modelos (Setor, Ruptura, Presenca), ter um único modelo Audit com campos dinâmicos.
Planilha (para registro de uploads):
const spreadsheetSchema = new mongoose.Schema({
loja: { type: mongoose.Schema.Types.ObjectId, ref: 'Store', required: true },
tipo: { type: String, enum: ['etiqueta', 'presenca', 'ruptura'], required: true },
data: Date,
nomeArquivo: String,
totalItens: Number,
totalItensLidos: Number,
// ...
});

2. Rotas:
/api/stores: GET (lista de lojas)
/api/upload: POST (upload de planilhas, com middleware para determinar a loja)
/api/audits: GET (para relatórios, estatísticas, etc.)
3. Middleware: em todas as rotas que precisam de loja, usar:

```
text

const getStore = async (req, res, next) => {
  const codigoLoja = req.headers['x-loja'];
  if (!codigoLoja) {
    return res.status(400).json({ error: 'Loja não especificada' });
  }
  const store = await Store.findOne({ codigo: codigoLoja });
  if (!store) {
    return res.status(404).json({ error: 'Loja não encontrada' });
  }
  req.store = store;
  next();
}
```

E nas rotas:

```
text

router.post('/upload', getStore, upload.single('file'), (req, res) => {
  const lojaId = req.store._id;
  // processar planilha e salvar com lojaId
});
```

4. Frontend: sempre que o usuário selecionar uma loja, salvar no Pinia e incluir o código da loja no header das requisições.
5. Processamento de planilhas: a função de processamento deve ser genérica o suficiente para lidar com os três tipos, mas pode ter tratamentos específicos por tipo.
6. Indexes: criar índices nos campos loja e data para consultas rápidas.

Esta é uma visão geral. A implementação real exigirá ajustes.

Dada a complexidade, você pode querer fazer isso gradualmente, começando por criar o modelo Store e modificar as rotas de upload para usar a loja do header e salvar o ObjectId da loja.

Se você preferir manter modelos separados para cada tipo de auditoria, pode criar:

```
text

- Etiqueta: que herda de AuditBase (usando discriminator do mongoose)
- Presenca: que herda de AuditBase
- Ruptura: que herda de AuditBase
```

Mas a query fica mais complexa.

Alternativamente, pode ter um único modelo Audit com um campo 'tipo' e um schema flexível.

Vou deixar a decisão com você. O importante é que todas as auditorias tenham uma referência para a loja (ObjectId) e que as consultas filtrem por essa loja.

Vamos agora modificar as rotas existentes para usar a loja do header e salvar o ObjectId da loja.

Passos:
1. Criar modelo Store e popular com as lojas.
2. Modificar o middleware verificarLoja para buscar a loja pelo código e anexar o objeto da loja (com _id) ao request.
3. Modificar todas as funções de processamento para salvar o ObjectId da loja (req.store._id) nos documentos.
4. Modificar as consultas em estatísticas, relatórios, etc. para filtrar por esse ObjectId.

Isso é um trabalho significativo, mas tornará o sistema mais organizado.

Se você quiser, podemos começar criando o modelo Store e modificando uma rota de cada vez.

Vamos começar pelo modelo Store:

```
// models/Store.js
import mongoose from 'mongoose';
```

```
const storeSchema = new mongoose.Schema({
codigo: { type: String, required: true, unique: true },
nome: { type: String, required: true },
cidade: String,
ativa: { type: Boolean, default: true }
});

export default mongoose.model('Store', storeSchema);
```

Depois, popule o banco com as lojas que você tem.

Em seguida, modifique o middleware verificarLoja para usar o modelo Store.

Vamos modificar o middleware:

```
// middleware/verificarLoja.js
import Store from '../models/Store.js';

async function verificarLoja(req, res, next) {
try {
let codigoLoja = req.headers['x-loja'] || req.body.loja || req.session.loja;
// Se não tem código, usa padrão (não recomendado, mas para não quebrar)
if (!codigoLoja) {
codigoLoja = '056';
}

  text

   const store = await Store.findOne({ codigo: codigoLoja });
   if (!store) {
     return res.status(400).json({ error: 'Loja não encontrada' });
   }

   req.store = store;
   next();
```

```
} catch (error) {
res.status(500).json({ error: 'Erro ao verificar loja' });
}
}

export default verificarLoja;
```

Então, use esse middleware nas rotas que precisam.

Nas rotas de upload, após processar, salve o ObjectId da loja (req.store._id) no documento.

Exemplo em processarEtiqueta:

```
const setorData = {
...,
loja: req.store._id
};
```

E nas consultas, use:

```
await Setor.find({ loja: req.store._id });
```

Isso garante que cada loja veja apenas seus dados.

Recomendo também modificar os modelos para usar referência para Store:

```
// No modelo Setor, por exemplo:
loja: { type: mongoose.Schema.Types.ObjectId, ref: 'Store', required: true }
```

E o mesmo para os outros modelos.

Espero que isso ajude a dar uma direção. Se precisar de mais detalhes em alguma parte, me avise.

## Reestruturação do Sistema de Auditorias - Banco de Dados e Arquitetura

**Analisando seu código, vejo que há bastante duplicação e complexidade desnecessária. Vou propor uma estrutura mais organizada e eficiente.**

### Estrutura Ideal do Banco de Dados

#### 1. Modelo de Lojas (Stores)

```javascript
// models/Store.js
const storeSchema = new mongoose.Schema({
  code: { type: String, required: true, unique: true },
  name: { type: String, required: true },
  city: { type: String, required: true },
  image: { type: String, default: '/images/lojas/default.jpg' },
  isActive: { type: Boolean, default: true },
  createdAt: { type: Date, default: Date.now }
});
```

#### 2. Modelo de Usuários (Users)

```javascript
// models/User.js
const userSchema = new mongoose.Schema({
```

```
  userId: { type: String, required: true, unique: true },
  name: { type: String, required: true },
  email: { type: String },
  stores: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Store' }],
  isActive: { type: Boolean, default: true },
  createdAt: { type: Date, default: Date.now }
});
```

### 3. Modelo Unificado de Auditorias (Audits)

```
javascript

// models/Audit.js
const auditSchema = new mongoose.Schema({
  store: { type: mongoose.Schema.Types.ObjectId, ref: 'Store', required: true },
  user: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
  type: {
    type: String,
    required: true,
    enum: ['etiqueta', 'presenca', 'ruptura']
  },
  date: { type: Date, required: true },
  items: [{
    code: String,
    product: String,
    location: String,
    status: {
      type: String,
      enum: ['Atualizado', 'Não lido', 'Com problema', 'Com presença', 'Sem presença'],
      default: 'Não lido'
    },
    stock: { type: Number, default: 0 },
    additionalData: mongoose.Schema.Types.Mixed
  }],
  summary: {
    totalItems: { type: Number, default: 0 },
    itemsRead: { type: Number, default: 0 },
    completionRate: { type: Number, default: 0 }
  },
  file: {
    name: String,
    uploadDate: { type: Date, default: Date.now },
    size: Number
  }
});
```

### 4. Modelo de Processamento de Arquivos (FileProcess)

```
javascript

// models/FileProcess.js
const fileProcessSchema = new mongoose.Schema({
  store: { type: mongoose.Schema.Types.ObjectId, ref: 'Store', required: true },
  user: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
  originalName: String,
  processedName: String,
  type: {
    type: String,
    enum: ['etiqueta', 'presenca', 'ruptura']
  },
  status: {
    type: String,
    enum: ['pending', 'processing', 'completed', 'error'],
    default: 'pending'
  },
  result: {
    totalItems: Number,
    processedItems: Number,
    errors: [String]
  },
  processingTime: Number,
  createdAt: { type: Date, default: Date.now }
});
```

### Estrutura de Pastas e Arquivos Recomendada

```
text

src/
├── models/
│   ├── Store.js
│   ├── User.js
│   ├── Audit.js
│   └── FileProcess.js
├── routes/
│   ├── index.js
│   ├── stores.js
│   ├── uploads.js
```

```
│   ├── audits.js
│   ├── reports.js
│   └── statistics.js
├── middleware/
│   ├── auth.js
│   ├── store.js
│   └── upload.js
├── services/
│   ├── fileProcessor.js
│   ├── auditService.js
│   └── reportService.js
├── utils/
│   ├── excelParser.js
│   ├── helpers.js
│   └── validators.js
└── app.js
```

## Fluxo de Processamento Unificado

### 1. Rota de Upload Única

javascript

```javascript
// routes/uploads.js
router.post('/upload',
  storeMiddleware,
  upload.single('file'),
  async (req, res) => {
    try {
      const { type } = req.body;
      const { store } = req;

      // Registrar processamento
      const processRecord = await FileProcess.create({
        store: store._id,
        user: req.user._id,
        originalName: req.file.originalname,
        type
      });

      // Processar em segundo plano
      processFileBackground(processRecord._id, req.file, type, store);

      res.json({
        success: true,
        message: 'Arquivo recebido e em processamento',
        processId: processRecord._id
      });
    } catch (error) {
      res.status(500).json({ error: error.message });
    }
  }
);
```

### 2. Serviço de Processamento Centralizado

javascript

```javascript
// services/fileProcessor.js
async function processFile(processId, file, type, store) {
  try {
    // Atualizar status para processando
    await FileProcess.findByIdAndUpdate(processId, { status: 'processing' });

    const workbook = xlsx.readFile(file.path);
    const data = parseWorkbook(workbook, type);

    // Processar de acordo com o tipo
    const result = await processAuditData(data, type, store);

    // Salvar auditoria
    const audit = await Audit.create({
      store: store._id,
      user: result.userId,
      type,
      date: new Date(),
      items: result.items,
      summary: {
        totalItems: result.totalItems,
        itemsRead: result.itemsRead,
        completionRate: result.completionRate
      },
      file: {
        name: file.originalname,
        size: file.size
      }
    });

    // Atualizar processamento como concluído
```

```
      await FileProcess.findByIdAndUpdate(processId, {
        status: 'completed',
        result: {
          totalItems: result.totalItems,
          processedItems: result.itemsRead,
          errors: result.errors
        },
        processingTime: Date.now() - startTime
      });

      return audit;
    } catch (error) {
      await FileProcess.findByIdAndUpdate(processId, {
        status: 'error',
        result: { errors: [error.message] }
      });
      throw error;
    }
  }
```

### Middleware de Loja Simplificado

```
javascript

// middleware/store.js
const getStore = async (req, res, next) => {
  try {
    const storeCode = req.headers['x-store'] || req.session.storeCode;

    if (!storeCode) {
      return res.status(400).json({
        error: 'Nenhuma loja selecionada'
      });
    }

    const store = await Store.findOne({ code: storeCode });

    if (!store) {
      return res.status(404).json({
        error: 'Loja não encontrada'
      });
    }

    req.store = store;
    next();
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
};
```

### Vantagens desta Abordagem

1. **Desacoplamento**: Separação clara de responsabilidades
2. **Manutenibilidade**: Código mais fácil de manter e modificar
3. **Escalabilidade**: Estrutura que permite crescimento
4. **Consistência**: Dados uniformes em todo o sistema
5. **Performance**: Processamento em background não bloqueia requisições
6. **Rastreabilidade**: Histórico completo de processamento de arquivos

### Migração Progressiva

**Se você não pode reescrever tudo de uma vez:**

1. Comece criando os novos modelos
2. Implemente a rota de upload unificada primeiro
3. Migre gradualmente as funcionalidades existentes
4. Mantenha compatibilidade com versão anterior durante a transição
5. Use a nova estrutura para novos desenvolvimentos

**Esta estrutura resolve o problema de dados salvos na loja errada e proporciona uma base sólida para expansões futuras.**