TypeScript×CASLでつくるSaaSの認可

株式会社PeopleX 坂津 潤平 / 芹澤 和也

はじめに

BtoB SaaS開発において「認可 (Authorization)」は複雑でセキュリティリスクの高い領域

- **UI** では、権限がないボタンを「見せない」
- **API** では、権限がないリソース操作を「させない」
- **DB** では、権限がないデータを「引かない」

これらがバラバラに実装されると、セキュリティホールやバグの温床に。

-> PeopleX AI面接における**CASL** を中心に据えた認可アーキテクチャの実践を紹介

BtoB SaaSの代表認可パターン

パターン	概要	具体例
ロールベース	役割（Role）に基づく大枠の制御	「管理者」は設定変更可、「一般」は閲覧のみ
属性ベース	所属に基づく動的な制御	自分の部署のリソースのみ閲覧可
関係性ベース	所有に基づく動的な制御	自分の所有・担当するリソースのみ閲覧可
フィールドレベル	特定の項目（列）ごとの制御	給与額や評価コメントなど、センシティブな情報のマスク

認可の代表的なモデルのおさらい

SaaS要件に応じて適切なモデルを選択する必要がある

モデル	判定基準	特徴
RBAC	**役割**	・シンプルで直感的 ・「誰が何をできるか」を定義
ABAC	**属性**	・「AかつBなら許可」のような複雑な条件を表現できる ・きめ細かな制御が可能
ReBAC	**関係性**	・グラフ構造や階層構造に強い ・親リソースの権限を子へ継承させやすい
PBAC	**ポリシー**	・認可ロジックを分離・コード管理できる (Policy as Code) ・RBAC/ABAC/ReBACを包括的に表現可能

デモ

題材: PeopleX AI面接

代表的な要件	認可モデルの適用	内容
テナント管理者はテナント設定の操作のみ可能	**RBAC**	「テナント管理者」というロールに適切な権限を定義
評価担当者は「自部署」の応募者のみ閲覧可能	**ABAC**	企業ユーザーの属性（部署ID）とリソースの属性を比較
評価担当者には個人情報（氏名・生年月日など）を見せない	**Field-Level**	特定のフィールド（列）へのアクセス制御

要求に合わせてどの認可パターンで実装するかを決定
コストパフォーマンスやメンテナンス性などを考慮

技術選定: SaaS vs OSS

「AaaS（Authorization as a Service）」

Auth0 のような IDaaS は広く普及している
BtoB SaaS における認可機能の提供
SaaSとOSSどちらを選定するべき？

主要な認可サービス

1. **Auth0 FGA**
    - Auth0 が提供する、Google Zanzibar モデルに基づいた認可サービス。
    - **特徴:** Auth0 (Okta) エコシステムの一部であり、多機能。
2. **Oso Cloud**
    - 開発者体験 (DX) を最優先に設計された、BtoB SaaS 開発で人気のあるサービスの一つ。
    - **特徴:** 独自のポリシー言語「Polar」を使用。

主要な認可サービス

3. Permit.io
    - OPA (Open Policy Agent) をベースにしつつ、UIベースの管理に強みを持つサービス。
    - **特徴:** 「ローコード」での権限管理を重視。権限設定を変更できる 管理UI を提供。

OSSライブラリの採用

SaaSは強力だが、外部通信によるレイテンシや、ローカル開発・CI 環境の複雑化といったオーバーヘッドを伴う。

PeopleX AI面接では、**Node.js プロセス内で完結するライブラリベースのアプローチ**を採用。

OSSライブラリの採用

- **コスト:** 従量課金の SaaS モデルを避け、スケーリングしてもコストが増加しないようにしたい。
- **開発体験:** 外部依存を減らし、`npm install` だけで開発環境が整うシンプルさを保ちたい。
- **DB連携:** ビジネスロジックと認可ロジックのWhere句を分離しつつ、効率的なクエリを発行したい。
- **低レイテンシ:** 認可判定のたびにネットワーク通信を発生させたくない。

Why CASL?

1. **Isomorphic:** 定義したポリシーをフロントエンドとバックエンドの両方で再利用可能
2. **Type Safety:** ポリシーをTypeScript で宣言的に記述可能
3. **API Integration:** デコレータによるAPIレベルでの宣言的な認可設定が可能
4. **Database Integration:** SQLクエリ条件をポリシーとして宣言的に定義可能
5. **Frontend Integration:** 直感的・宣言的な認可判定が可能

Ability（TypeScript によるポリシー定義）

Ability とは？

CASL における **Ability** は、認可ルールの心臓部となるクラス
これは「ユーザーがどのような操作（Action）を、どのリソース（Subject）に対して行えるか」というルールを詰め込んだオブジェクト

アプリケーション側では、`ability.can('update', article)` と問いかけるだけで、複雑な条件分岐なしに権限判定を行うことが可能

型定義とAbilityの構築

Prismaの型をそのままSubjectとして利用可能

```
import { Candidate } from '@prisma/client';
import { PureAbility } from '@casl/ability';
import { PrismaQuery, Subjects } from '@casl/prisma';

// アクションとリソースの定義
export type Action = 'manage' | 'create' | 'read' | 'update' | 'delete';

// Prismaモデルと紐付け
export type AppSubjects = Subjects<{ Candidate: Candidate }> | 'all';

export type AppAbility = PureAbility<[Action, AppSubjects], PrismaQuery>;
```

ポリシーファクトリ

ポリシーを動的に定義（実際にはロールと権限のペアを別ファイルで宣言的に管理）

```
export function defineAbilityFor(user: UserContext): AppAbility {
  const { can, cannot, build } = new AbilityBuilder(createPrismaAbility);

  if (user.role === 'Recruiter') {
    // ABAC: 自部署の候補者のみ操作可能
    can(['read', 'update'], 'Candidate', {
      departmentId: { in: user.departmentIds },  // Prismaのクエリ構文で条件を記述
    });
  }
  if (user.role === 'Interviewer') {
    // Field-Level: 個人情報は見せない
    cannot('read', 'Candidate', ['salary', 'address']);
  }
  return build();
}
```

BE実装: APIレベルの制御 (NestJS)

Guardとデコレータで宣言的に記述可能

```
@Get()
@UseGuards(PoliciesGuard)
// 読む権限があるかチェック
@CheckPolicies((ability) => ability.can('read', 'Candidate'))
async findAll() {
  return this.candidatesService.findAll();
}
```

BE実装: Prisma

CASL 公式の `@casl/prisma` プラグインを使用することで、定義した Ability を Prisma の `where` 句として呼び出すだけ

```
import { accessibleBy } from '@casl/prisma';

const candidates = await prisma.candidate.findMany({
  where: {
    AND: [
      // AbilityからWhere句を参照
      accessibleBy(ability, 'read').Candidate,
    ]
  }
});
```

BE実装: フィールドレベルの認可

`permittedFieldsOf` ヘルパーが利用可能

```
import pick from 'lodash/pick';
import { permittedFieldsOf } from '@casl/ability/extra';

// 取得したオブジェクトから、見せてはいけないフィールドを除外する
const candidate = await prisma.candidate.findUnique({ ... });
const fields = permittedFieldsOf(ability, 'read', candidate, { fieldsFrom: rule => rule.fields || [] });

// Lodashのpickなどでフィルタリング
const sanitizedCandidate = pick(candidate, fields);
res.json(sanitizedCandidate);
```

FE実装: Canコンポーネント

```
export function CandidateCard({ candidate }) {
  return (
    <div className="card">
      <h1>{candidate.name}</h1>

      {/* 権限がある場合のみボタンを表示 */}
      <Can I="update" this={candidate}>
        <button onClick={handleEdit}>編集</button>
      </Can>
    </div>
  );
}
```

FE実装: 関数呼び出し

```
export function CandidateCard({ candidate }) {
  const ability = useAppAbility();  // AppAbilityを参照するカスタムフック

  return (
    <div className="card">
      <h1>{candidate.name}</h1>

      {/* 権限がない場合はボタンを非活性 */}
      <button onClick={handleEdit} disabled={ability.cannot('update', candidate)}>編集</button>
    </div>
  );
}
```

導入してよかったポイント

1. **TypeScriptによる認可ロジックの宣言的定義**
   認可のロジックが `Ability`（ポリシー定義）として一箇所に集約できる点。仕様変更があっても、ここだけ直せば全レイヤーに反映される安心感がある。

2. **Prisma 連携が強力**
   開発者は認可のロジックとビジネスロジックを混合せず、独立してクエリを書くことができる点。ロジックが混合せず、メンテナンス性の高いコードを維持することができている。

3. **開発者体験の向上**
   実装が宣言的・直感的にできる点。認可基盤の具体まで詳細に理解できていなくとも、適切に適用できる基盤を整えることができた。

工夫が必要なポイント

1. **複雑な条件の制御**
   Prisma の `WhereInput` に変換できる条件しか書けないため、あまりに複雑になるケースにおいては、無理に CASL に押し込まずサービス層で補完するなどの割り切りも必要と感じた。

2. **型定義の複雑さ**
   TypeScript の型推論をフル活用しようとすると、ジェネリクスや型定義がやや複雑。
   例えば、`PureAbility<[Action, Subjects<{ Candidate: Candidate }>], PrismaQuery>` のような定義は、初見では直感的に理解しづらい。

工夫が必要なポイント

3. **SQLの複雑化**

   `accessibleBy` は便利な反面、生成される SQL が暗黙的で意図せず複雑で冗長になってしまうことがある。インデックスが効きにくいクエリが生成されていないか、定期的なスロークエリの監視や開発時のデバッグは必要と感じている。

まとめ

- BtoB SaaSの認可は複雑
- 認可の導入をサポートしてくれるSaaSやOSSなどアプローチが複数ある
- 組織やアーキテクチャに沿った技術選定をすることで、すばやく実装し、メンテナンス性高く維持することができる
- TypeScript / Next.js / NestJS / Prisma を採用しているアーキテクチャにおいては、CASLが有力候補

正しい知識を元に、状況に応じた最適な技術選定を行い、プロダクトのコアバリューの開発に集中しよう