
FROM ZERO TO
HERO
隐蔽

ライブラリ

そのまま使うと

しんどなるで

大変な理由

- **変更時**
  - ライブラリを変えるとき、全部のコードを修正する必要がある
  - 1 箇所の変更が複数箇所に影響する
- **テスト・ユニットテスト**
  - 外部ライブラリに依存していると、テストが書きにくい

例：moment.js から date-fns に変更するとき

- 関数名が変わる
- 書き方が変わる

関数名が変わる

`fromNow()` → `formatDistanceToNow()`

```
// components/UserProfile.tsx
import moment from "moment";

export function UserProfile({ user }) {
  return (
    <div>
      <p>登録日: {moment(user.createdAt).format("YYYY-MM-DD")}</p>
      <p>最終ログイン: {moment(user.lastLogin).fromNow()}</p>
    </div>
  );
}
```

書き方が変わる

`moment(date)` → `new Date(date)`

```
import moment from "moment";

export function EventCard({ event }) {
  const isToday = moment(event.date).isSame(moment(), "day");
  return (
    <div>
      <h3>{event.title}</h3>
      <p>{moment(event.date).format("MM/DD HH:mm")}</p>
      {isToday && <span>今日のイベント！</span>}
    </div>
  );
}
```

axios のタイムアウト設定

各コンポーネントで個別に修正する必要がある

```
import axios from "axios";

export const useBlogData = () => {
  const fetchData = async () => {
    const axiosConfig = {
      timeout: 10000, // 10秒
      headers: {
        "Content-Type": "application/json",
        "User-Agent": "BlogViewer/1.0",
      },
    };
    const response = await axios.get("https://jsonplaceholder.typicode.com/posts", axiosConfig);
  };
```

```
  const refreshPosts = async () => {
    // 別の場所で異なるタイムアウト値
    const response = await axios.get(
      "https://jsonplaceholder.typicode.com/posts",
      {
        timeout: 5000, // 5秒
      }
    );
  };
};
```

MUI はもっと大変

例：MUI を直接使用したログイン画面

```
import { Button, TextField, Alert, Box, Typography } from "@mui/material";

export function LoginPage() {
  return (
    <Box>
      <Typography>ログイン</Typography>
      <TextField label="メールアドレス" />

      <TextField label="パスワード" type="password" />

      <Button onClick={handleLogin} variant="contained">
        "ログイン"
      </Button>
    </Box>
  );
}
```

MUI を辞めたい場合ほぼ全てのコンポーネントが修正対象になる

PJ が成長すると MUI が足枷になるときがくる

ほな、どうすんねん？！

カプセル化（隠蔽）で解決しよう！

カプセル化って何？

外部ライブラリを直接使わずに、自分で作った関数を経由して使うこと

具体的にはどうするの？

ライブラリ → 自分で作った関数 → コンポーネント

```
axios    → apiClient.getUser()           → LoginPage
MUI      → components/Button.tsx         → LoginPage
date-fns → formatDate() / diffFromNow()  → LoginPage
```

どんないいことがあるの？

- ライブラリを変えても 1 か所だけ修正すれば OK
- テストが簡単に書ける
- 自分のアプリ専用の機能を作りやすい
- チーム全員が同じ書き方で統一できる

axios を apiClient で隠蔽する

```
// lib/apiClient.ts
import axios from "axios";

const API_BASE_URL = "https://api.example.com";

const apiClient = axios.create({
  baseURL: API_BASE_URL,
  timeout: 10000,
  headers: {
    "Content-Type": "application/json",
    "User-Agent": "BlogViewer/1.0",
  },
});
```

```
apiClient.interceptors.request.use((config) => {
  const token = localStorage.getItem("authToken");
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});

export { apiClient };
```

コンポーネントからの使用

```
// useBlogData.ts
import { apiClient } from "@/lib/apiClient";

export const useBlogData = () => {
  const fetchData = async () => {
    const postsResponse = await apiClient.get("/posts");
    const usersResponse = await apiClient.get("/users");

    // タイムアウトやトークンは自動で設定される！
    setPosts(postsResponse.data);
    setUsers(usersResponse.data);
  };
};
```

MUI を独自コンポーネントで隠蔽する

コンポーネント構成

```
src/components/
├── Button/
│   └── index.tsx
├── TextInput/
│   └── index.tsx
└── Alert/
    └── index.tsx
```

必要な分だけコンポーネントを作っていく

Button コンポーネント

```tsx
// components/Button/index.tsx
import {
  Button as MuiButton,
  ButtonProps as MuiButtonProps,
} from "@mui/material";

type Props = {
  children: React.ReactNode;
  onClick?: () => void;
  variant?: "primary" | "secondary" | "danger";
  disabled?: boolean;
  fullWidth?: boolean;
};

export const Button = ({
  children,
  onClick,
  variant = "primary",
  ...props
}: Props) => {
  const muiVariant = variant === "danger" ? "outlined" : "contained";
  const muiColor = variant === "danger" ? "error" : "primary";

  return (
    <MuiButton
      variant={muiVariant}
      color={muiColor}
      onClick={onClick}
      {...props}
    >
      {children}
    </MuiButton>
  );
};
```

TextInput コンポーネント

```tsx
// components/TextInput/index.tsx
import { TextField } from "@mui/material";

type Props = {
  label: string;
  value: string;
  onChange: (value: string) => void;
  type?: "text" | "email" | "password";
  required?: boolean;
};

export const TextInput = ({ label, value, onChange, ...props }: Props) => {
  return (
    <TextField
      label={label}
      value={value}
      onChange={(e) => onChange(e.target.value)}
      fullWidth
      margin="normal"
      variant="outlined"
      {...props}
    />
  );
};
```

使用例

```tsx
// LoginPage.tsx
import { Button } from "@/components/Button";
import { TextInput } from "@/components/TextInput";
import { Alert } from "@/components/Alert";

export function LoginPage() {
  return (
    <>
      <TextInput
        label="メールアドレス"
        value={email}
        onChange={setEmail}
        type="email"
        required
      />

      <Button onClick={handleLogin} variant="primary">
        ログイン
      </Button>
    </>
  );
}
```

date-fns をクラスで隠蔽する

クラス構成

```
src/utils/
├── DateFormatter.ts    // フォーマット用
├── DateCalculator.ts   // 日付計算用
└── DateComparator.ts   // 日付比較用
```

なぜ機能単位で分割するの？

- **単一責任の原則** - 各クラスが 1 つの責任だけを持つ
- **必要な機能だけ import できる** - フォーマットだけ使いたいときは DateFormatter だけ
- **テストが書きやすい** - 機能ごとに独立してテストできる
- **見通しが良い** - どのクラスが何をするか一目瞭然

DateFormatter インターフェース

```
// utils/IDateFormatter.ts
export interface IDateFormatter {
  format(date: Date | string, pattern?: string): string;
  fromNow(date: Date | string): string;
  time(date: Date | string): string;
  datetime(date: Date | string): string;
}
```

DateFormatter クラス（1/2）

```
// utils/DateFormatter.ts
import { format, formatDistanceToNow } from 'date-fns';
import { ja } from 'date-fns/locale';
import { IDateFormatter } from './IDateFormatter';

export class DateFormatter implements IDateFormatter {
  // 日付を指定フォーマットで表示
  static format(date: Date | string, pattern: string = 'yyyy/MM/dd'): string {
    const targetDate = typeof date === 'string' ? new Date(date) : date;
    return format(targetDate, pattern, { locale: ja });
  }

  // 相対時間を表示（例：3日前）
  static fromNow(date: Date | string): string {
    const targetDate = typeof date === 'string' ? new Date(date) : date;
    return formatDistanceToNow(targetDate, {
      addSuffix: true,
      locale: ja
    });
  }
```

DateFormatter クラス（2/2）

```
  // 時刻を表示
  static time(date: Date | string): string {
    return this.format(date, 'HH:mm');
  }

  // 日付と時刻を表示
  static datetime(date: Date | string): string {
    return this.format(date, 'yyyy/MM/dd HH:mm');
  }
}
```

DateCalculator インターフェース

```
// utils/IDateCalculator.ts
export interface IDateCalculator {
  addDays(date: Date | string, days: number): Date;
  addMonths(date: Date | string, months: number): Date;
  subDays(date: Date | string, days: number): Date;
  diffInDays(date1: Date | string, date2: Date | string): number;
}
```

DateCalculator クラス（1/2）

```
// utils/DateCalculator.ts
import { addDays, addMonths, subDays, differenceInDays } from 'date-fns';
import { IDateCalculator } from './IDateCalculator';

export class DateCalculator implements IDateCalculator {
  // 日数を加算
  static addDays(date: Date | string, days: number): Date {
    const targetDate = typeof date === 'string' ? new Date(date) : date;
    return addDays(targetDate, days);
  }

  // 月数を加算
  static addMonths(date: Date | string, months: number): Date {
    const targetDate = typeof date === 'string' ? new Date(date) : date;
    return addMonths(targetDate, months);
  }
```

DateCalculator クラス（2/2）

```
  // 日数を減算
  static subDays(date: Date | string, days: number): Date {
    const targetDate = typeof date === 'string' ? new Date(date) : date;
    return subDays(targetDate, days);
  }

  // 日数の差分を計算
  static diffInDays(date1: Date | string, date2: Date | string): number {
    const d1 = typeof date1 === 'string' ? new Date(date1) : date1;
    const d2 = typeof date2 === 'string' ? new Date(date2) : date2;
    return differenceInDays(d1, d2);
  }
}
```

使用例

```tsx
// components/UserProfile.tsx
import { DateFormatter } from "@/utils/DateFormatter";
import { DateCalculator } from "@/utils/DateCalculator";

export function UserProfile({ user }) {
  const memberDays = DateCalculator.diffInDays(new Date(), user.createdAt);

  return (
    <div>
      <p>登録日: {DateFormatter.format(user.createdAt)}</p>
      <p>最終ログイン: {DateFormatter.fromNow(user.lastLogin)}</p>
      <p>メンバー歴: {memberDays}日</p>
    </div>
  );
}
```

カプセル化でモックが簡単に！

なぜモックが必要なの？

- **外部ライブラリに依存しないテスト**を書きたい
- **決まった値を返す**ことでテストを安定させたい
- **ライブラリの動作を気にせず**ビジネスロジックをテストしたい

直接使用 vs カプセル化の比較

date-fns を直接使った場合

```
// ❌ 大変：ライブラリ全体をモックする必要
jest.mock("date-fns", () => ({
  format: jest.fn(),
  formatDistanceToNow: jest.fn(),
  addDays: jest.fn(),
  subDays: jest.fn(),
  differenceInDays: jest.fn(),
  // まだまだ他にも...
}));

jest.mock("date-fns/locale", () => ({
  ja: {},
}));
```

問題点：

- ライブラリの内部構造を知る必要がある
- 使わない関数もモックしないといけない

カプセル化した場合

```
// ✅ 簡単：自作クラスだけモック
jest.mock("@/utils/DateFormatter", () => ({
  DateFormatter: {
    format: jest.fn(() => "2024/01/01"),
    fromNow: jest.fn(() => "3日前"),
  },
}));
```

メリット：

- 自分で作ったクラスだけモックすれば OK
- 必要な関数だけモック
- ライブラリの更新に影響されない

実際のテストコード

```
test("ユーザー情報の表示", () => {
  const user = { createdAt: "2024-01-01" };
  const result = render(<UserProfile user={user} />);

  // モックで指定した値が表示される
  expect(result.getByText("3日前")).toBeInTheDocument();
});
```

さらに柔軟な DI パターン

```
// コンポーネントで依存性注入
export function UserProfile({
  user,
  dateFormatter = DateFormatter, // デフォルトは本物
}) {
  return <p>登録日: {dateFormatter.format(user.createdAt)}</p>;
}
```

テストではモックを注入

```
test("ユーザープロフィール", () => {
  const mockFormatter = {
    format: jest.fn(() => "テスト日付"),
  };

  render(<UserProfile user={user} dateFormatter={mockFormatter} />);

  // 呼び出されたかも確認できる
  expect(mockFormatter.format).toHaveBeenCalledWith(user.createdAt);
});
```