TYPE-DRIVEN UI

Designing React Components with Pattern Matching

with for M.U.G. - Marca User Group
by Filippo Zancanaro

QUALCHE INFO

Full Stack Developer

React, Angular, .NET, Elixir & Phoenix and Flutter

Bartender

Mi potete trovare al Vigna Colada

Padre adottivo di uno Shiba Inu

Il suo Instagram è @batsugram, se siete curiosi/e

@batsugram

CONTATTI

> linkedin.com/in/filippozancanaro

> zancanaro.filippo.97@gmail.com

> github.com/filippozancanaro

Filippo Zancanaro

Software Engineer

sibill.

AVETE MAI VISTO QUALCOSA DEL GENERE?

un esempio iper semplificato di un classico scenario in cui ci ritroviamo a dover usare una serie di condizioni per gestire diversi stati del componente

```tsx
function UserCard({ user }: { user: User }) {
  if (user.isLoading) {
    return <Spinner />;
  }
  if (user.error) {
    return <ErrorBanner message={user.error} />;
  }
  if (!user.data) {
    return <EmptyState />;
  }
  if (user.data.role === "admin") {
    return <AdminCard data={user.data} />;
  }
  return <DefaultCard data={user.data} />;
}
```

```tsx
type User = {
  isLoading: boolean;
  error: string | null;
  data: UserData | null;
};
```

+

certo,

QUESTO APPROCCIO FUNZIONA

e se lo vorrete usare non sarò di certo io a fermarvi :)

+

```tsx
function UserCard({ user }: { user: User }) {
  if (user.isLoading) {
    return <Spinner />;
  }
  if (user.error) {
    return <ErrorBanner message={user.error} />;
  }
  if (!user.data) {
    return <EmptyState />;
  }
  if (user.data.role === "admin") {
    return <AdminCard data={user.data} />;
  }
  return <DefaultCard data={user.data} />;
}
```

MA HA DEI PROBLEMI

+

```tsx
function UserCard({ user }: { user: User }) {
  if (user.isLoading) {
    return <Spinner />;
  }
  if (user.error) {
    return <ErrorBanner message={user.error} />;
  }
  if (!user.data) {
    return <EmptyState />;
  }
  if (user.data.role === "admin") {
    return <AdminCard data={user.data} />;
  }
  return <DefaultCard data={user.data} />;
}
```

01

Non è scalabile

All’aumentare della logica aumenta la complessità, fino a diventare illeggibile e poco manutenibile.

02

Non è esaustivo

Il compilatore non ci avviserà se dimentichiamo un caso (o due, tre, ecc...).

03

Non è mutualmente esclusivo

Questo è un esempio semplice, ma come vedremo in seguito lascia spazio a combinazioni logicamente errate.

04

Rischio di bug

Aumentando la dispersività e senza esaustività, il rischio di bug e di reiterazioni sul codice è dietro l’angolo.

IL VERO PROBLEMA?

il nostro tipo non riflette la realtà del dominio

```ts
type User = {
  isLoading: boolean;
  error: string | null;
  data: UserData | null;
};
```

Con un oggetto piatto come questo, possiamo potenzialmente rappresentare combinazioni di stati che non hanno senso

Esempio: *isLoading: true* assieme a *data: { /* something */ }* è una combinazione tecnicamente valida per TypeScript, ma impossibile e scorretta da un punto di vista logico.

LA BASE DELLA NOSTRA SOLUZIONE: DISCRIMINATED UNIONS

```ts
type UserState =
  | { status: "loading" }
  | { status: "error"; message: string }
  | { status: "empty" }
  | { status: "ready"; data: UserData; role: "admin" | "user" }
```

Le **Discriminated Unions** sono lo strumento di TypeScript per modellare stati mutuamente esclusivi.
La proprietà ***status*** è il discriminante: TypeScript la userà per restringere il tipo in ogni combinazione e ci darà errore in caso di condizioni che prevedono stati incoerenti.

questo è il fondamento su cui ci baseremo per il resto del talk

CHE COS'È IL PATTERN MATCHING

Il **pattern matching** non è un concetto nuovo, esiste da decenni nei linguaggi funzionali.
L'idea è semplice: si confronta un valore con una serie di pattern, e si esegue il codice del primo pattern corrispondente.

```
// 1> Confronta un valore con una struttura attesa
// 2> Estrai le parti che ti servono
// 3> Esegui il branch corrispondente
```

A differenza di **switch**, il pattern matching può operare su strutture annidate, tipi complessi, e il compilatore garantisce che tutti i casi siano coperti (**exhaustiveness checking**).

```
case user_state do
  :loading ->
    render_spinner()

  {:error, msg} ->
    render_error(msg)

  {:ready, data} ->
    render_card(data)
end
```

← Un esempio di pattern matching in Elixir adattato al codice delle slide precedenti.
Il pattern matching esiste anche in altri linguaggi come Rust, Scala ed Haskell.

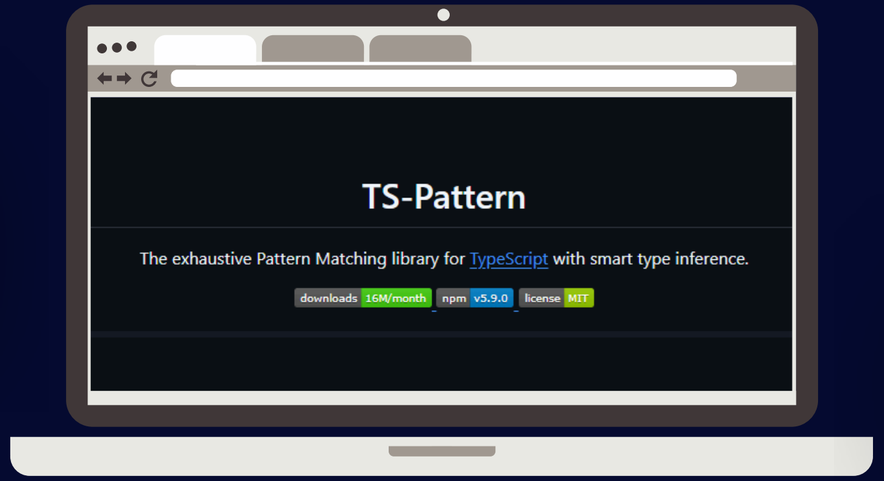
E JAVASCRIPT / TYPESCRIPT?

Ad oggi, Javascript e Typescript non supportano nativamente questa feature.*

Tuttavia, possiamo implementarla in modo facile ed intuitivo con una libreria:

STATS

⭐ ~15k stars su GitHub
📦 0 deps
🔒 100% type_safe
✅ Exhaustiveness checking a compile time




```
npm install ts-pattern
```

```
import { match, P } from 'ts-pattern';
```

**NOTA: Esiste una proposta ufficiale (TC39) per aggiungere il pattern matching a JavaScript (e di conseguenza a TypeScript).*
Attualmente è allo Stage 1 (o Stage 2 a seconda delle specifiche sezioni), il che significa che è in fase di discussione e il design non è ancora definitivo.

CORE FEATURES: FUNCTIONS

Match & Exhaustive
Pattern annidati

Pattern helpers
Select

Ts-Pattern

MATCH( ) + .EXHAUSTIVE( )

```ts
import { match } from 'ts-pattern';

// discriminated union: ogni forma ha delle info diverse
type Shape =
  | { kind: 'circle'; radius: number }
  | { kind: 'square'; side: number }
  | { kind: 'triangle'; base: number; height: number };

// usiamo ts-pattern per calcolare l'area
// in base alla forma dovremo applicare logiche differenti
const area = (shape: Shape): number =>
  match(shape)
    .with({ kind: 'circle' }, (s) => Math.PI * s.radius ** 2)
    .with({ kind: 'square' }, (s) => s.side ** 2)
    .with({ kind: 'triangle' }, (s) => (s.base * s.height) / 2)
    .exhaustive(); // <- darà errore a compile time se manca un caso
```

```ts
const area = (shape: Shape): number =>
  match(shape)
    .with({ kind: 'circle' }, (s) => Math.PI * s.radius ** 2)
    // ecc...
    .otherwise(() => {
      throw new Error('Forma non riconosciuta');
    });
```

.exhaustive() → se aggiungiamo un nuovo tipo alla union ma dimentichiamo di gestirlo nel match, TypeScript ci darà un errore a compile time (prima ancora di eseguire il codice), trasformando errori di runtime in errori di compilazione (ed impedendoci di dimenticare casi).

.otherwise() → è l'alternativa quando vogliamo un fallback default, rinunciando però alla garanzia di exhaustiveness.

PATTERN HELPER E P._ WILDCARD

```ts
import { match, P } from 'ts-pattern';

const describe = (value: unknown): string =>
  match(value)
    // tipi semplici
    .with(P.string, (s) => `È una stringa: "${s}"`)
    .with(P.number, (n) => `È un numero: ${n}`)
    .with(P.boolean, (b) => `È un booleano: ${b}`)

    // array di numeri
    .with(P.array(P.number), () => 'Array di numeri')

    // tuple (sequenze finite e ordinate)
    .with([P.string, P.number], ([s, n]) => `Una tupla stringa + numero: ${s}, ${n}`)

    // array di oggetti
    .with(P.array({ kind: 'item', id: P.number }), () => 'Array di oggetti')

    // oggetti chiave-valore
    .with(P.record(P.string, P.number), () => 'Record key string e value number')
    .with(P.record(P.string, { active: P.boolean }), () => 'Record key string e value oggetto')

    // classi
    .with(P.instanceOf(MyAppUser), (u) => `È una classe custom MyAppUser: ${u.name}`)

    // null e undefined
    .with(null, () => 'È null')
    .with(undefined, () => 'È undefined')

    // fallback wildcard
    .with(P._, () => 'Tipo non riconosciuto')

    // non ci serve exhaustiveness checking per via del P._
    .run();
```

P è il namespace dei **pattern helper**.
Invece di matchare valori esatti, possiamo matchare per tipo o struttura.

P._ invece funziona come il case default di uno switch, matchando per type any tutto ciò che non abbiamo verificato prima.

Wildcards disponibili:

- P._ o P.any
- P.string, P.number, P.boolean, P.nullish
- P.array(...), P.record(...)
- P.instanceOf(Class)

PATTERN ANNIDATI

```ts
type ApiResponse =
  | { status: 'success'; data: { user: { role: 'admin' | 'user' } } }
  | { status: 'error'; code: 401 | 403 | 500 }
  | { status: 'loading' };

const getMessage = (res: ApiResponse) =>
  match(res)
    .with({ status: 'success', data: { user: { role: 'admin' } } },
      () => 'Benvenuto, amministratore!')
    .with({ status: 'success', data: { user: { role: 'user' } } },
      () => 'Benvenuto!')
    .with({ status: 'error', code: 401 },
      () => 'Non autorizzato')
    .with({ status: 'error', code: P.union(403, 500) },
      ({ code }) => `Errore server: ${code}`)
    .with({ status: 'loading' },
      () => 'Caricamento...')
    .exhaustive();
```

Questa è una delle cose impossibili da gestire elegantemente con lo **switch**:
il **deep matching**.

Il pattern si applica ricorsivamente alla struttura dell'oggetto, gestendo casistiche complesse con estrema facilità.

Nell’esempio a sinistra vediamo inoltre come possiamo utilizzare **P.union()** per raggruppare più valori in un unico branch, evitando ripetizioni.

P.SELECT( ) – ESTRAI VALORI DAL PATTERN

```ts
import { match, P } from 'ts-pattern';

type Event =
  | { type: 'click'; target: { id: string; label: string } }
  | { type: 'keydown'; key: string; modifiers: string[] };

const handleEvent = (event: Event) =>
  match(event)
    // Estrae target.id come variabile "id"
    .with(
      { type: 'click', target: { id: P.select('id') } },
      ({ id }) => `Cliccato elemento: ${id}`
    )
    // Estrae l'intero oggetto con select implicito
    .with(
      { type: 'keydown', key: P.select() },
      (key) => `Tasto premuto: ${key}`
    )
    .exhaustive();
```

P.select() è estremamente comodo quando vogliamo estrarre una parte specifica del valore matchato, senza dover fare destructuring manuale nella callback.

La sintassi di base è:
P.select('NOME')
dove NOME è il nome della variabile che passeremo al callback

Con il nome opzionale possiamo estrarre più valori contemporaneamente e riceverli come oggetto nella callback.
Se non esplicitiamo il nome, la callback riceverà direttamente il valore.

Di seguito alcuni esempi più o meno complessi per comprendere meglio gli usi e la sintassi.

P.SELECT( ) - ESEMPI EXTRA

```ts
import { match, P } from 'ts-pattern';

const user = { name: 'Tony Stark', age: 30 };

const result = match(user)
  .with({ age: P.select('eta') }, ({ eta }) => `Età: ${eta}`)
  .exhaustive();

console.log(result); // "Età: 30"
```

1- Estrazione di un valore semplice

```ts
const result = match(['Tony Stark', 'Steve Rogers'])
  .with([P.select('first'), P._], ({ first }) => `Primo Avenger: ${first}`)
  .exhaustive();
```

2- Estrazione da un array

```ts
const point = { x: 10, y: 20 };

const result = match(point)
  .with(
    { x: P.select('a'), y: P.select('b') },
    ({ a, b }) => `Somma: ${a + b}`
  )
  .exhaustive();
```

3- Estrazione multipla

P.SELECT( ) - NOME SI O NOME NO?

```ts
match(value)
  .with(
    { user: { id: P.select(), name: P.select() }, active: P.boolean },
    (selected) => selected
  )
  .run();

// input
value = { user: { id: 1, name: "Bruce Wayne" }, active: true }

// output
selected === [1, "Bruce Wayne"]
```

P.select() **senza** nome, inserisce i valori come valori indicizzati di un **array numerico**

P.select() **con** il nome, inserisce i valori come **proprietà di un oggetto**

```ts
.with(
  { a: P.select(), b: P.select("nomePersonalizzato"), c: P.select() },
  (selected) => selected
)

// output
selected === {
  nomePersonalizzato: valoreB,
  0: valoreA,
  1: valoreC
}
```

P.select() in casi **ibridi con** e **senza** nome, inserisce i valori come **proprietà di un oggetto** assegnando ai campi anonimi un nome **numerico indicizzato**

TS-PATTERN IN RENDERING E ROUTING

GESTIONE DI UN ASYNC STATE

Prima

```tsx
function UserProfile({ state }: { state: UserState }) {
  if (state.isLoading) return <Spinner />;
  if (state.error) return <ErrorMessage error={state.error} />;
  if (!state.data) return <p>Nessun utente trovato.</p>;

  return (
    <div>
      <h1>{state.data.name}</h1>
      {state.data.isAdmin && <Badge>Admin</Badge>}
    </div>
  );
}
```

Dopo

```tsx
type UserState =
  | { status: 'loading' }
  | { status: 'error'; error: Error }
  | { status: 'empty' }
  | { status: 'ready'; data: User };

// ...

function UserProfile({ state }: { state: UserState }) {
  return match(state)
    .with({ status: 'loading' }, () => <Spinner />)
    .with({ status: 'error' }, ({ error }) => (
      <ErrorMessage message={error.message} />
    ))
    .with({ status: 'empty' }, () => <p>Nessun utente trovato.</p>)
    .with({ status: 'ready' }, ({ data }) => (
      <div>
        <h1>{data.name}</h1>
        {data.isAdmin && <Badge>Admin</Badge>}
      </div>
    ))
    .exhaustive();
}
```

Nel “dopo”: esaustività, narrowing automatico (tipizzazione corretta senza necessità di cast) e niente stati insensati

Ts-Pattern

COMBINAZIONE DI STATI MULTIPLI

```tsx
// combinazione di due stati asincroni:
function Dashboard({ userState, dataState }) {
  if (userState.isLoading || dataState.isLoading) {
    return <GlobalSpinner />;
  }
  if (userState.error) {
    return <ErrorPage error={userState.error} />;
  }
  if (dataState.error) {
    return <ErrorPage error={dataState.error} />;
  }
  if (!userState.data || !dataState.data) {
    return <EmptyState />;
  }
  // resto della UI qui, una volta caricato tutto
}
```

Prima

```tsx
// Match su tupla: gestiamo ogni combinazione esplicitamente

function Dashboard({ userState, dataState }: {
  userState: UserState;
  dataState: DataState;
}) {
  return match([userState, dataState] as const)
    // Entrambi in errore: mostriamo solo il primo
    .with(
      [{ status: 'error' }, P._],
      ([{ error }]) => <ErrorPage error={error} />
    )
    // Loading di qualsiasi tipo
    .with(
      [{ status: 'loading' }, P._],
      [P._, { status: 'loading' }],
      () => <GlobalSpinner />
    )
    // Entrambi pronti: la UI vera
    .with(
      [{ status: 'ready' }, { status: 'ready' }],
      ([{ data: user }, { data }]) => (
        <DashboardContent user={user} data={data} />
      )
    )
    // In tutti gli altri casi, placeholder
    .otherwise(() => <EmptyState />);
}
```

Dopo

Possiamo gestire stati multipli passando un array a match() e verificando tramite ts-pattern su diverse **tuple** di valori.

Possiamo anche raggruppare branch multipli sullo stesso .with() separandoli con virgola: se uno dei pattern fa match, entra in quel branch (sotto un esempio semplice).

```tsx
match(value)
  .with(1, 2, 3, () => "È un numero piccolo")
  .otherwise(() => "Altro");
```

TS-PATTERN NEL ROUTING?

Assolutamente si!

```ts
type Route =
  | { path: '/'; }
  | { path: '/users'; }
  | { path: '/users/:id'; id: string }
  | { path: '/settings'; tab: 'profile' | 'security' | 'billing' }
  | { path: '/404' };

function Router({ route }: { route: Route }) {
  return match(route)
    .with({ path: '/' }, () => <HomePage />)
    .with({ path: '/users' }, () => <UserListPage />)
    .with({ path: '/users/:id' }, ({ id }) => (
      <UserDetailPage userId={id} />
    ))
    .with({ path: '/settings', tab: 'profile' }, () => <ProfileSettings />)
    .with({ path: '/settings', tab: 'security' }, () => <SecuritySettings />)
    .with({ path: '/settings', tab: 'billing' }, () => <BillingSettings />)
    .with({ path: '/404' }, () => <NotFoundPage />)
    .exhaustive();
}
```

Il pattern matching ci permette di fare deep matching anche sui parametri.

Inoltre, aggiungere una nuova route al tipo Route senza gestirla nel Router darà un errore.

ESEMPI IN FUNZIONI LOGICHE

ES. GESTIONE DI EVENTI E AZIONI

```ts
// PRIMA: classico switch
type Action =
  | { type: 'INCREMENT'; amount: number }
  | { type: 'DECREMENT'; amount: number }
  | { type: 'RESET' }
  | { type: 'SET'; value: number };

function counterReducer(state: number, action: Action): number {
  switch (action.type) {
    case 'INCREMENT': return state + action.amount;
    case 'DECREMENT': return state - action.amount;
    case 'RESET':     return 0;
    case 'SET':       return action.value;
    default:
      // TypeScript non ci avviserà se aggiungiamo un'azione nel type
      // e dimentichiamo il case
      return state;
  }
}
```

Prima

```ts
type Action =
  | { type: 'INCREMENT'; amount: number }
  | { type: 'DECREMENT'; amount: number }
  | { type: 'RESET' }
  | { type: 'SET'; value: number };

// DOPO: match con exhaustive
function counterReducer(state: number, action: Action): number {
  return match(action)
    .with({ type: 'INCREMENT' }, ({ amount }) => state + amount)
    .with({ type: 'DECREMENT' }, ({ amount }) => state - amount)
    .with({ type: 'RESET' },    () => 0)
    .with({ type: 'SET' },      ({ value }) => value)
    .exhaustive(); // Se aggiungiamo un'azione nel type, exhaustive darà errore
}
```

Dopo

ES. TRASFORMAZIONE DI DATI

```ts
// Normalizzare risposte API eterogenee verso un tipo comune
type RawProduct =
  | { source: 'shopify'; product_id: string; title: string; price_cents: number }
  | { source: 'woocommerce'; ID: number; post_title: string; regular_price: string }
  | { source: 'internal'; id: number; name: string; price: number };

type Product = { id: string; name: string; priceEuro: number };

const normalize = (raw: RawProduct): Product =>
  match(raw)
    .with({ source: 'shopify' }, (p) => ({
      id: `shopify-${p.product_id}`,
      name: p.title,
      priceEuro: p.price_cents / 100,
    }))
    .with({ source: 'woocommerce' }, (p) => ({
      id: `woo-${p.ID}`,
      name: p.post_title,
      priceEuro: parseFloat(p.regular_price),
    }))
    .with({ source: 'internal' }, (p) => ({
      id: `int-${p.id}`,
      name: p.name,
      priceEuro: p.price,
    }))
    .exhaustive();
```

ES. GESTIONE ERRORI API

```ts
type ApiErrorResponse =
  | {
      status: 400;
      body: {
        error: "VALIDATION_ERROR";
        fields: Record<string, string>;
      };
    }
  | {
      status: 401;
      body: {
        error: "AUTH_ERROR";
        code: "TOKEN_EXPIRED" | "INVALID_TOKEN" | "MISSING_TOKEN";
      };
    }
  | {
      status: 403;
      body: {
        error: "FORBIDDEN";
      };
    }
  | {
      status: 404;
      body: {
        error: "NOT_FOUND";
        resource: string;
      };
    }
  | {
      status: 500;
      body: {
        error: "INTERNAL_ERROR";
        traceId: string;
      };
    }
  | {
      status: number;
      body: {
        error: string;
        message?: string;
      };
    };
```

STEP 1:
gestiamo un mapping da
ApiErrorResponse a AppError

```ts
type AppError =
  | {
      kind: "validation";
      fields: Record<string, string>;
    }
  | {
      kind: "auth";
      reason: "expired" | "invalid" | "missing";
    }
  | {
      kind: "forbidden";
    }
  | {
      kind: "not_found";
      resource: string;
    }
  | {
      kind: "server";
      traceId?: string;
    }
  | {
      kind: "network";
      status: number;
    }
  | {
      kind: "unknown";
      message: string;
    };
```

ES. GESTIONE ERRORI API

```ts
import { match, P } from "ts-pattern";

const mapApiError = (error: ApiErrorResponse): AppError =>
  match(error)
    .with({ status: 400, body: { error: "VALIDATION_ERROR" } }, ({ body }) => ({
      kind: "validation",
      fields: body.fields,
    }))
    .with(
      { status: 401, body: { error: "AUTH_ERROR", code: "TOKEN_EXPIRED" } },
      () => ({ kind: "auth", reason: "expired" }),
    )
    .with(
      { status: 401, body: { error: "AUTH_ERROR", code: "INVALID_TOKEN" } },
      () => ({ kind: "auth", reason: "invalid" }),
    )
    .with(
      { status: 401, body: { error: "AUTH_ERROR", code: "MISSING_TOKEN" } },
      () => ({ kind: "auth", reason: "missing" }),
    )
    .with({ status: 403 }, () => ({ kind: "forbidden" }))
    .with({ status: 404, body: { error: "NOT_FOUND" } }, ({ body }) => ({
      kind: "not_found",
      resource: body.resource,
    }))
    .with({ status: 500 }, ({ body }) => ({
      kind: "server",
      traceId: body.traceId,
    }))
    .with({ status: P.when((s) => s >= 500) }, () => ({ kind: "server" }))
    .otherwise((e) => ({
      kind: "unknown",
      message: e.body?.message ?? "Errore sconosciuto",
    }));
```

ES. GESTIONE ERRORI API

```ts
const getErrorMessage = (error: AppError): string =>
  match(error)
    .with({ kind: "validation" }, ({ fields }) => {
      const messages = Object.entries(fields).map(
        ([field, msg]) => `${field}: ${msg}`,
      );
      return `Errori nei campi:\n${messages.join("\n")}`;
    })
    .with(
      { kind: "auth", reason: "expired" },
      () => "Sessione scaduta. Rifai login.",
    )
    .with({ kind: "auth", reason: "invalid" }, () => "Token non valido.")
    .with({ kind: "auth", reason: "missing" }, () => "Devi autenticarti.")
    .with({ kind: "forbidden" }, () => "Non hai i permessi per questa azione.")
    .with({ kind: "not_found" }, ({ resource }) => `${resource} non trovato.`)
    .with(
      { kind: "server" },
      ({ traceId }) => `Errore server. ${traceId ? `Codice: ${traceId}` : ""}`,
    )
    .with({ kind: "network" }, ({ status }) => `Errore di rete (${status})`)
    .with({ kind: "unknown" }, ({ message }) => message)
    .exhaustive();
```

STEP 2:
Creiamo una funzione error handling custom

```ts
try {
  const user = await fetchUser();
} catch (err) {
  const message = getErrorMessage(err as AppError);
  toast.error(message);
}
```

STEP 3:
La usiamo per gestire le eccezioni delle chiamate API

ES. GESTIONE DI UNA STATE MACHINE

```ts
type OrderStatus =
  | { state: 'pending' }
  | { state: 'processing'; startedAt: Date }
  | { state: 'shipped'; trackingId: string }
  | { state: 'delivered'; deliveredAt: Date }
  | { state: 'cancelled'; reason: string };

type OrderEvent =
  | { type: 'CONFIRM' }
  | { type: 'SHIP'; trackingId: string }
  | { type: 'DELIVER' }
  | { type: 'CANCEL'; reason: string };

const transition = (
  status: OrderStatus,
  event: OrderEvent
): OrderStatus =>
  match([status, event] as const)
    .with([{ state: 'pending' }, { type: 'CONFIRM' }],
      () => ({ state: 'processing', startedAt: new Date() }))
    .with([{ state: 'processing' }, { type: 'SHIP' }],
      ([, { trackingId }]) => ({ state: 'shipped', trackingId }))
    .with([{ state: 'shipped' }, { type: 'DELIVER' }],
      () => ({ state: 'delivered', deliveredAt: new Date() }))
    .with([P._, { type: 'CANCEL' }],
      ([, { reason }]) => ({ state: 'cancelled', reason }))
    .otherwise(([s]) => s); // Transizione non valida: stato invariato
```

RICAPITOLANDO...

QUANDO USARE TS-PATTERN

Gestione **discriminated unions complesse** (con più di 3 casistiche)

Deep matching su oggetti annidati

Combinazione di stati indipendenti

Quando serve **exhaustiveness checking**

QUANDO NON USARE TS-PATTERN

Gestione di **discriminated unions semplici** (es 2 stati gestibili con ternary)

Matching su **valori primitivi** (switch is the way)

Se aggiunge **overhead non necessario**

Il **team non** è **familiare** con **TS avanzato** o con **approcci funzionali**

QUESTION TIME!

...o **pizza time**?

GRAZIE DELL'ATTENZIONE

il vostro amichevole Software Engineer di quartiere


zancanaro.filippo.97@gmail.com


linkedin.com/in/filippozancanaro