DVMM: Domain-View Model Mapper

A Frontend Architecture Pattern for Scalable and Maintainable TypeScript Applications

Abstract

This paper presents **DVMM (Domain-View Model Mapper)**, an architectural pattern tailored for frontend development in TypeScript. DVMM advocates for a clear separation between backend data structures and frontend UI models, bridged by deterministic mappers. This separation offers benefits in scalability, testability, and collaboration across backend and frontend teams. The pattern is especially valuable in large codebases where APIs evolve independently from UI requirements.

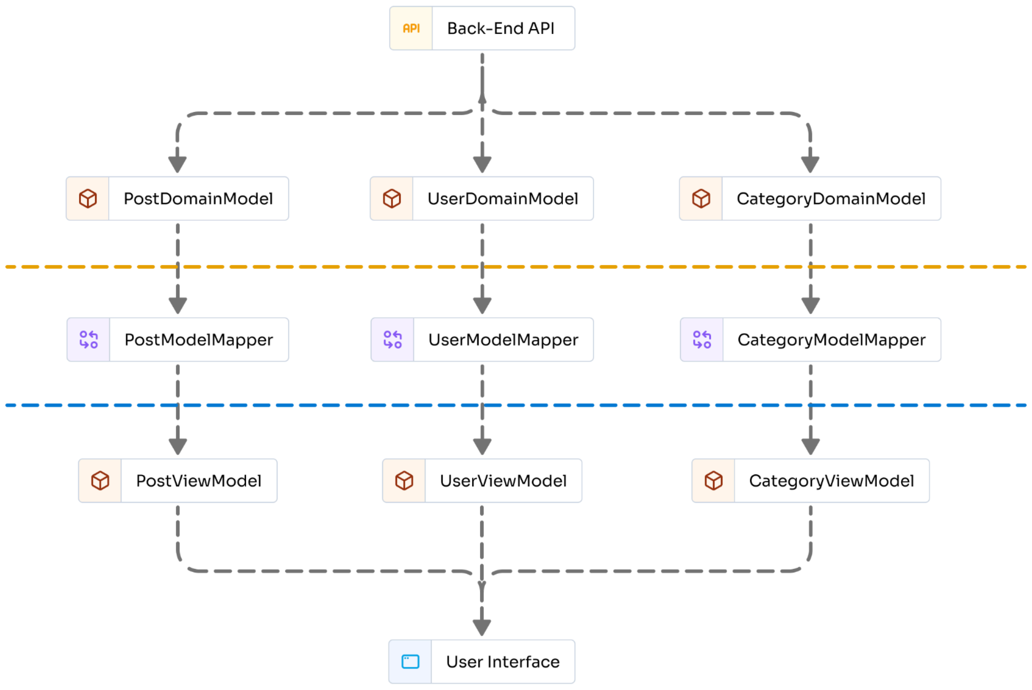
1. Introduction

Modern frontend applications often rely on dynamic APIs that evolve rapidly. In many cases, UI components consume API responses directly, leading to fragile codebases that break with every backend change. The lack of clear boundaries between backend and frontend responsibilities leads to tight coupling, making maintenance and scaling difficult.

DVMM introduces a structural pattern for frontend teams to:

- Mirror backend contracts exactly using **Domain Models**.
- Define UI-specific representations using **View Models**.
- Convert between them using pure, **deterministic Mappers**.

2. Architecture Overview



2.1 Components

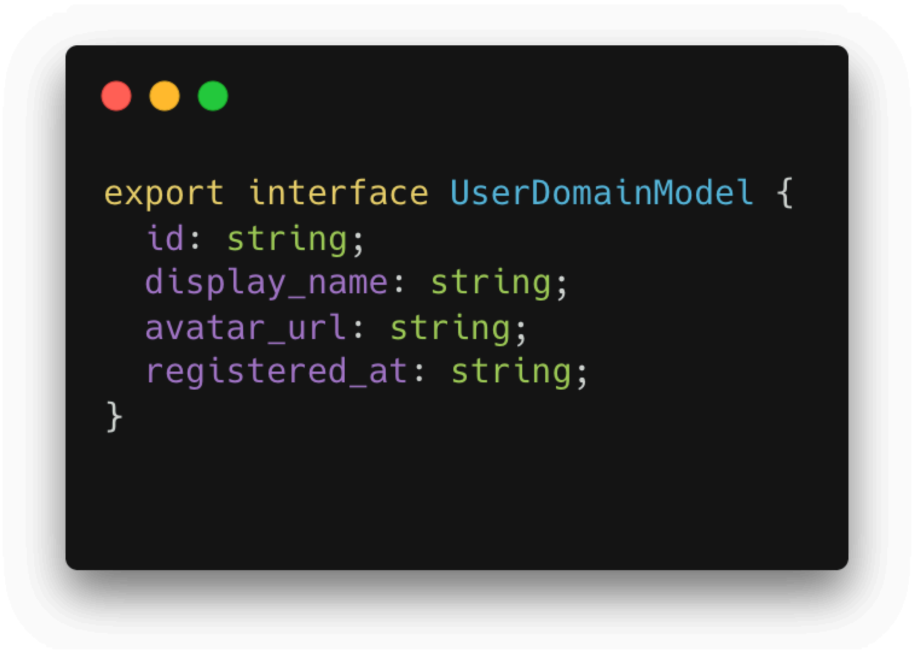
- **Domain Model:** Represents the exact API response structure. These are not to be used directly in UI logic.
- **View Model:** Represents the data tailored for UI — reformatted, renamed, or enriched for rendering.
- **Model Mapper:** A pure function or class responsible for converting between domain and view models, and vice versa.

2.2 Folder Structure

```
📦 src
├── 📂 data
│   ├── 📂 domain
│   │   ├── 📂 models         🧩 Domain Models (API response types)
│   │   ├── 📂 functions      🔌 API Call Functions
│   │   ├── 📄 client.ts      🌐 API Client Setup (Axios / Fetch / Ky)
│   │   └── 📄 index.ts       🧠 Domain Logic Entry Point
│   ├── 📂 view
│   │   ├── 📂 models         🎨 View Models (UI-specific data)
│   │   ├── 📂 mappers        🔁 Domain ↔ View Model Mappers
│   │   └── 📄 index.ts       🎯 View Logic Entry Point
├── 📂 components             🧱 Reusable UI Components
└── 📂 pages                  📄 Application Pages (Routes)
```

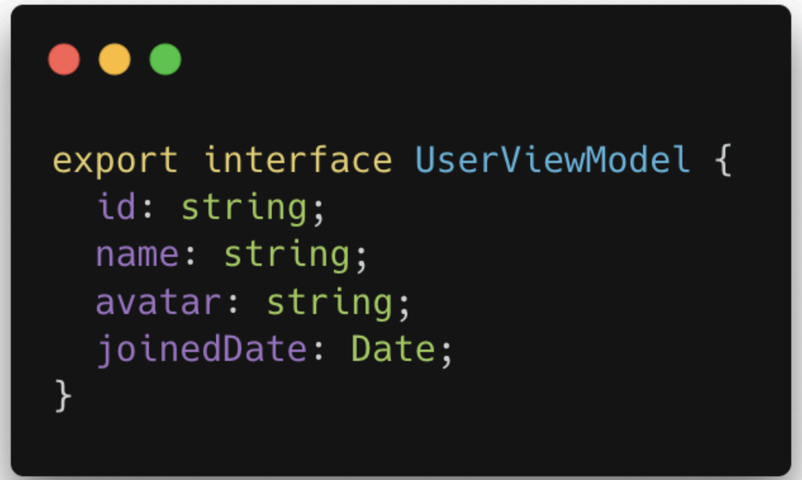
3. Implementation

3.1 Domain Model



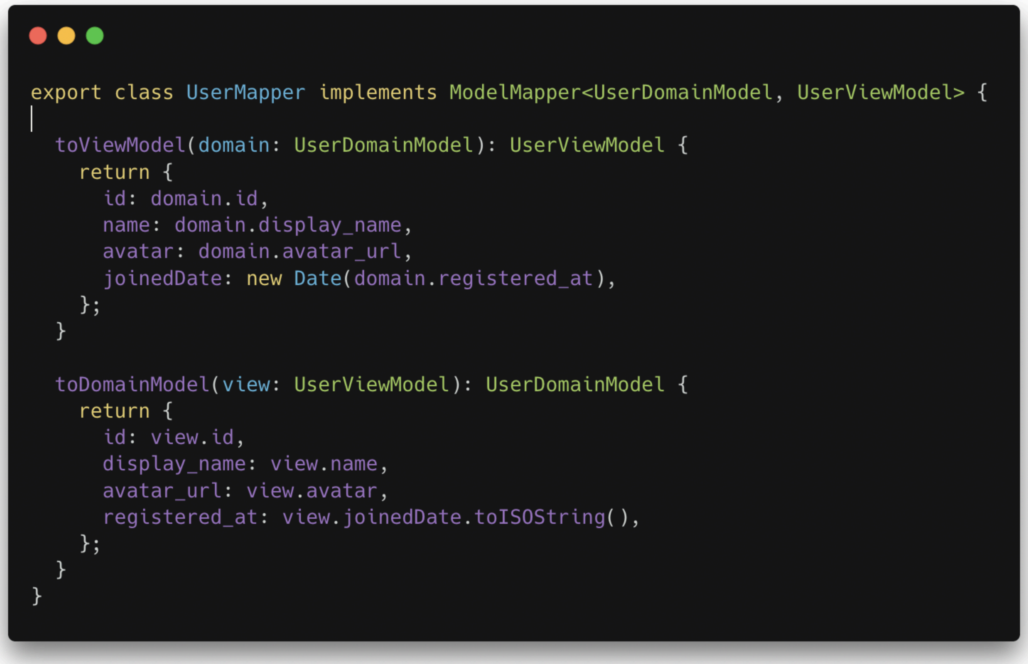
```
export interface UserDomainModel {
  id: string;
  display_name: string;
  avatar_url: string;
  registered_at: string;
}
```

3.2 View Model



```
export interface UserViewModel {
  id: string;
  name: string;
  avatar: string;
  joinedDate: Date;
}
```

3.3 Model Mapper



```
export class UserMapper implements ModelMapper<UserDomainModel, UserViewModel> {

  toViewModel(domain: UserDomainModel): UserViewModel {
    return {
      id: domain.id,
      name: domain.display_name,
      avatar: domain.avatar_url,
      joinedDate: new Date(domain.registered_at),
    };
  }

  toDomainModel(view: UserViewModel): UserDomainModel {
    return {
      id: view.id,
      display_name: view.name,
      avatar_url: view.avatar,
      registered_at: view.joinedDate.toISOString(),
    };
  }
}
```

4. Benefits

4.1 Type Safety

Each layer has well-defined types, reducing runtime bugs and improving IDE support and auto-completion.

4.2 Scalability

Teams can work independently on domain logic and UI without stepping on each other's toes.

4.3 Refactor-Friendly

API response changes are isolated within the mappers.

4.4 Testability

Mappers can be independently tested using unit tests.

4.5 Developer Experience

The structure encourages separation of concerns, improves onboarding, and boosts productivity.

5. Use Cases

- SaaS dashboards consuming multiple APIs.
- Mobile and web front-ends sharing a common API.
- Enterprise systems with strict versioning between frontend and backend.
- Teams practicing Backend-for-Frontend (BFF) strategies.

6. Limitations & Trade-offs

- Slightly higher initial boilerplate.
- Developers must maintain two sets of models and mappers.
- Overhead may not be justified for small-scale apps with minimal UI logic.

7. Conclusion

DVMM enforces a clear architectural boundary between backend and frontend data models. It provides a robust foundation for frontend teams building scalable and maintainable applications in TypeScript. While it introduces an additional layer of abstraction, the trade-offs are justified in most mid to large-scale applications.

By introducing deterministic mapping and encouraging discipline around data transformations, DVMM helps developers ship reliable and testable frontend code — faster.

8. References

- Martin Fowler – Patterns of Enterprise Application Architecture
- Flutter Docs - Guide to app architecture
- TypeScript Handbook

For questions or collaboration, feel free to reach out.