Architecture Quick Reference Guide

Quick decision guide for daily development.

🗂️ Where Does This Code Go?

Domain (`src/domain/`)

Put here:

- ✅ Business entities (`Pregnancy`, `Menstruation`, etc.)
- ✅ Value objects (`Week`, `DateRange`)
- ✅ Repository interfaces (contracts)
- ✅ Domain-specific types

Rules:

- ❌ NO imports from other layers
- ❌ NO framework dependencies (React, Redux, Firebase)
- ✅ Pure TypeScript only

Application (`src/application/`)

Put here:

- ✅ Use cases (business workflows)
- ✅ Application services (orchestration)
- ✅ Repository interfaces
- ✅ Shared application types

Rules:

- ✅ Can import from Domain
- ❌ CANNOT import from Presentation
- ❌ CANNOT import from Infrastructure
- ❌ NO Redux imports (`createAsyncThunk`)
- ❌ NO React imports
- ✅ Framework-agnostic only

Infrastructure (`src/infrastructure/`)

Put here:

- ✅ Repository implementations (Firebase, Sanity)
- ✅ Serializers and mappers
- ✅ External service adapters

Rules:

- ✅ Can import from Domain and Application
- ❌ CANNOT import from Presentation
- ✅ Firebase/Sanity imports allowed here

Note: DI container lives at `src/container.ts` (outside layers) as it's a bootstrap concern.

Presentation (`src/presentation/`)

Put here:

- ✅ Redux state (`state/slices/`, `state/thunks/`)
- ✅ ViewModels (React hooks)
- ✅ UI components
- ✅ UI-specific utilities

Rules:

- ✅ Can import from Domain and Application
- ✅ React and Redux imports allowed here
- ✅ This is where `createAsyncThunk` lives

🔄 Common Patterns

Pattern 1: Adding a New Use Case

```
// 1. Create Use Case (Application Layer)
// src/application/useCases/myFeature/DoSomething.useCase.ts

export interface DoSomethingParams {
  userId: string;
  data: SomeData;
}

export interface DoSomethingResult {
  success: boolean;
  data: ResultData;
}

@injectable()
export class DoSomethingUseCase {
  constructor(
    @inject("SomeRepository")
    private repository: SomeRepository,
  ) {}

  async execute(params: DoSomethingParams): Promise<DoSomethingResult> {
    // Pure business logic here
    const result = await this.repository.doSomething(params.data);
    return { success: true, data: result };
  }
}
```

```
// 2. Register in DI Container (Bootstrap file)
// src/container.ts

import { DoSomethingUseCase } from "@/application/useCases/myFeature/DoSomething.useCase";

container.register<DoSomethingUseCase>(DoSomethingUseCase, {
  useClass: DoSomethingUseCase,
});
```

Pattern 2: Adding a Repository

```
// 1. Define Interface (Application Layer)
// src/application/repositories/MyDataRepository.ts

export interface MyDataRepository {
  getById(id: string): Promise<MyData | null>;
  save(data: MyData): Promise<void>;
}
```

```
// 2. Implement for Firebase (Infrastructure Layer)
// src/infrastructure/firebase/MyDataRepositoryImpl.firebase.ts

@injectable()
export class MyDataRepositoryImplFirebase implements MyDataRepository {
  async getById(id: string): Promise<MyData | null> {
    const doc = await firestore().collection("myData").doc(id).get();
    return doc.exists ? (doc.data() as MyData) : null;
  }

  async save(data: MyData): Promise<void> {
    await firestore().collection("myData").doc(data.id).set(data);
  }
}
```

```
// 3. Register in DI Container (Bootstrap file)
// src/container.ts

container.registerSingleton<MyDataRepository>(
  "MyDataRepository",
  MyDataRepositoryImplFirebase,
);
```

Pattern 3: Adding a Service (Pure Functions)

For stateless services, prefer plain functions over classes:

```
// src/application/services/myService.ts

export const calculateSomething = (input: number): number => {
  return input * 2;
};

export const transformData = (data: SomeData): TransformedData => {
  return {
    id: data.id,
    value: calculateSomething(data.value),
  };
};
```

❌ **DON'T** use DI for pure functions:

```
// ❌ Unnecessary
@injectable()
export class MyService {
  calculateSomething(input: number): number {
    return input * 2;
  }
}
```

🚫 Common Anti-Patterns

❌ Use Case with Redux Thunk

DON'T:

```
// ❌ Application layer should NOT have createAsyncThunk
@injectable()
export class GetDataUseCase {
  execute = createAsyncThunk("data/get", async () => {
    // ...
  });
}
```

DO:

```
// ✅ Use case returns Promise
@injectable()
export class GetDataUseCase {
  async execute(): Promise<Data> {
    // ...
  }
}

// ✅ Thunk in Presentation layer
export const fetchData = createAsyncThunk("data/fetch", async () => {
  const useCase = container.resolve(GetDataUseCase);
  return await useCase.execute();
});
```

❌ Application Importing Presentation

DON'T:

```
// ❌ Application importing from Presentation
// src/application/services/MyService.ts
import { SomeType } from "@/presentation/components/SomeComponent";
```

DO:

```
// ✅ Move shared type to Application
// src/application/types/SomeType.ts
export type SomeType = { ... };

// ✅ Both layers import from Application
// src/application/services/MyService.ts
import { SomeType } from '@/application/types/SomeType';

// src/presentation/components/SomeComponent.tsx
import { SomeType } from '@/application/types/SomeType';
```

❌ Domain Importing Frameworks

DON'T:

```
// ❌ Domain should not import frameworks
// src/domain/entities/User.ts
import { firestore } from "@react-native-firebase/firestore";
```

DO:

```
// ✅ Domain is pure
// src/domain/entities/User.ts
export class User {
  constructor(
    public id: string,
    public name: string,
  ) {}
}

// ✅ Firebase in Infrastructure
// src/infrastructure/firebase/UserRepositoryImpl.firebase.ts
import { firestore } from "@react-native-firebase/firestore";
```

❌ Using DI for Pure Functions

DON'T:

```
// ❌ Unnecessary class wrapper
@injectable()
export class DateService {
  formatDate(date: Date): string {
    return date.toISOString();
  }
}
```

DO:

```
// ✅ Simple exported function
export const formatDate = (date: Date): string => {
  return date.toISOString();
};
```

✅ Quick Decision Tree

```
Is it a business entity or value object?
├─ YES → Domain Layer (entities/, types/)
└─ NO ↓

Does it orchestrate business operations?
├─ YES → Application Layer (useCases/, services/)
└─ NO ↓

Is it a repository implementation or external service?
├─ YES → Infrastructure Layer (firebase/, sanity/, di/)
└─ NO ↓

Is it UI-related (React, Redux, ViewModels)?
└─ YES → Presentation Layer (state/, viewModels/, components/)
```

📝 Checklist for Code Reviews

Domain Layer:

- [ ] No imports from other layers
- [ ] No framework dependencies
- [ ] Pure TypeScript

Application Layer:

- [ ] Use cases return `Promise<T>`, NOT `createAsyncThunk`
- [ ] No Redux imports
- [ ] No React imports
- [ ] Only imports from Domain

Infrastructure Layer:

- [ ] Implements interfaces from Application/Domain
- [ ] No imports from Presentation

Bootstrap (container.ts):

- [ ] Only wires dependencies together
- [ ] Doesn't contain business logic
- [ ] Lives outside 4-layer structure

Presentation Layer:

- [ ] Redux thunks are here
- [ ] ViewModels are here

🎯 Testing Guidelines

Unit Test Use Cases

```
describe("GetDataUseCase", () => {
  it("should fetch data", async () => {
    // Arrange
    const mockRepo = { getData: jest.fn().mockResolvedValue(mockData) };
    const useCase = new GetDataUseCase(mockRepo);

    // Act
    const result = await useCase.execute();

    // Assert
    expect(result).toEqual(mockData);
  });
});
```

Integration Test Thunks

```
describe("fetchData thunk", () => {
  it("should update Redux state", async () => {
    const store = mockStore({ data: initialState });

    await store.dispatch(fetchData());

    const actions = store.getActions();
    expect(actions[0].type).toBe("data/fetch/pending");
    expect(actions[1].type).toBe("data/fetch/fulfilled");
  });
});
```

📚 Common Questions

Q: Where do I put Redux slices?
A: `src/presentation/state/slices/`

Q: Where do I put `createAsyncThunk`?
A: `src/presentation/state/thunks/`

Q: Can Application layer call Infrastructure?
A: No. Application depends on interfaces. Infrastructure implements them.

Q: Can I use DI for everything?
A: No. Only for classes with dependencies. Use plain functions for pure logic.

Q: Where does StateManager go?
A: `src/presentation/state/` (it's being moved from `application/`)

Q: Where does the DI container go?
A: `src/container.ts` (outside the 4 layers - it's a bootstrap concern)

Q: Where does legacy code go?
A: `src/old/` (being phased out)

Q: My use case needs `getState()` from Redux?
A: Pass the required data as parameters instead. Get it from state in the thunk.

Last Updated: 2025-11-19

See Also: Full Architecture Report