spaceteams

hellostavanger

Marc Brehmer

Developing a clean architecture-inspired React application with MVVM

Architecture

About me

- Software Engineer since 2013
- @Spaceteams
- Technology agnostics - No boundaries or biases
- Staying ahead of the game - It’s Spacetime

marc-brehmer marc_brehmer disco-joern

Agenda

spaceteams

The Multifaceted Issues of Software Development

🚨 Issues incoming...

🚨 The Problems We Face

- 🏗️ **Structural Problems**
  - 🔄 Code Duplication
  - 🔗 Tight Coupling
  - 📈 Scalability Challenges
- 🛠️ **Development Issues**
  - 🧪 Testing Complexity
  - 🔧 Reduced Maintainability
  - ⚠️ Inconsistencies
- 💼 **Business Impact**
  - 🚚 Difficulty in Migration
  - 🤸 Lack of Flexibility
  - 💰 Cost & Time Implications

💸 The Real Cost

- 📈 Development Speed: 124% slower in poor quality code
- 🐛 Defect Rate: 15x more bugs in unhealthy codebases
- ⏰ Wasted Time: 23-42% of developer time lost to technical debt
- 📊 Unpredictability: 9x longer maximum task durations

Source: CodeScene "Code Red: The Business Impact of Code Quality" (2022)

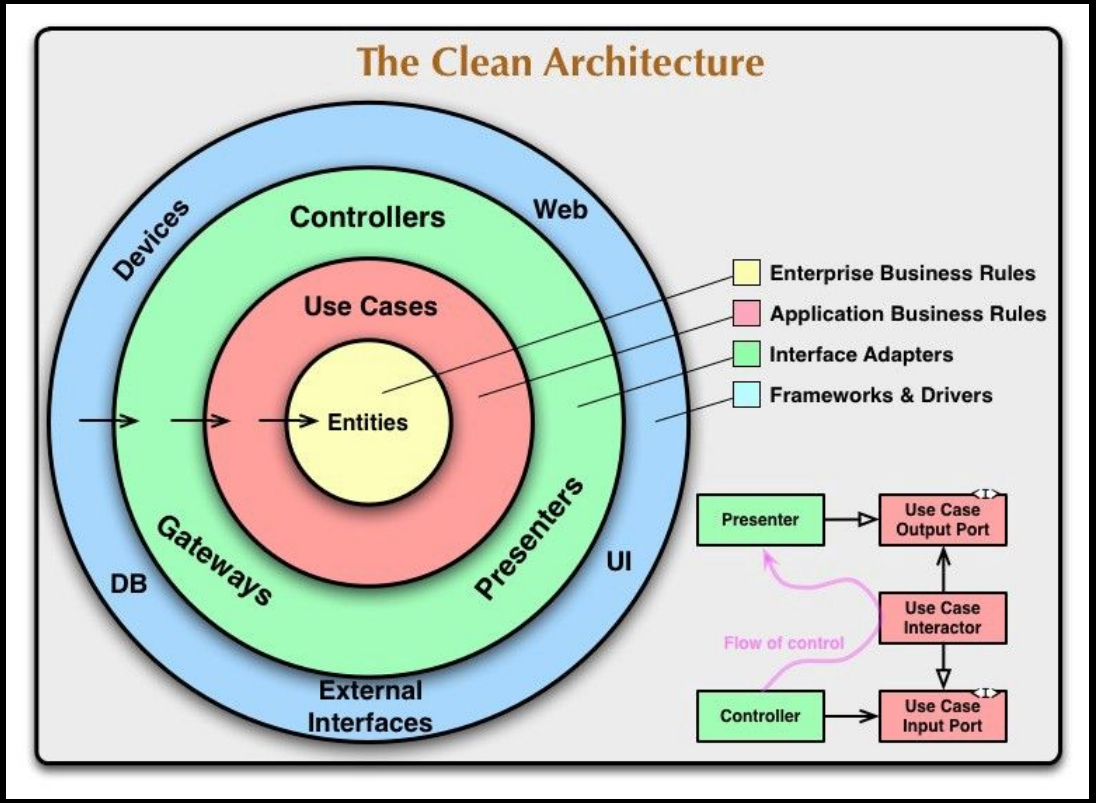
spaceteams

Clean Architecture: Foundation for Better Software

🏗️ Clean Architecture Fundamentals

- **🎯 The Solution to Our Problems**
  - Framework-independent
  - Puts business logic at the center
  - Testable & maintainable by design
- **🔄 The Dependency Rule**
  - Challenge: Use Cases need data persistence but can't depend on outer layers
  - Solution: Use Cases depend on interfaces, not implementations

- **🏛️ The Four Layers**



✅ Clean Architecture: Solutions & Benefits

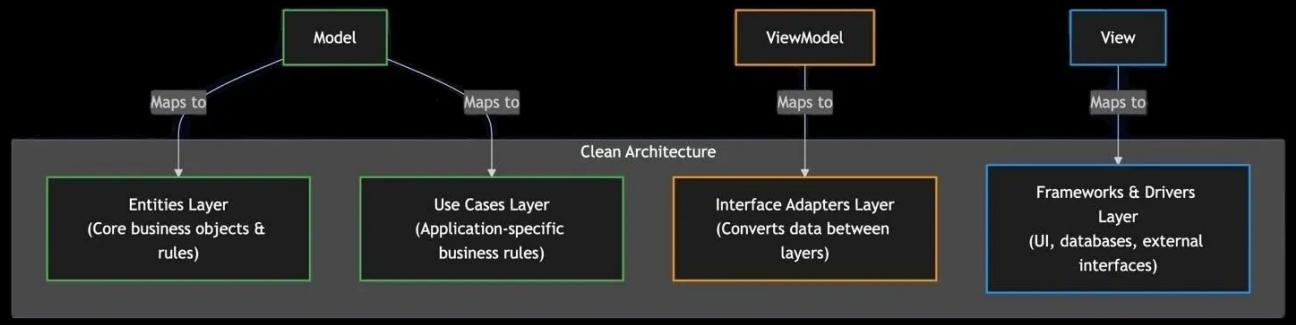
Problem	Clean Architecture Solution
🔄 Code Duplication	Reusable components across layers
🔗 Tight Coupling	Dependency inversion & interfaces
📈 Scalability Challenges	Modular layers for easy expansion
🧪 Testing Complexity	Isolated, testable components
⚠️ Inconsistencies	Consistent architectural patterns
🔧 Reduced Maintainability	Clear separation of concerns
🚚 Difficulty in Migration	Framework independence at core
🤸 Lack of Flexibility	Swappable outer layers

spaceteams

MVVM: The Complementary Pattern

🎯 MVVM & Clean Architecture Mapping

- 📊 **The MV* Pattern Family**
  - **Goal:** Separation of concerns
  - **Popular Members:** MVC, MVP, MVVM
  - **MVVM:** Perfect complement



- 🧩 **MVVM Components in React**
  - 💾 **Model** - Business Logic (Entities & Use Cases)
  - 🎣 **ViewModel** - Custom React Hooks (Interface Adapters)
  - 👀 **View** - React Components (UI Layer)

✨ The Perfect Combination

- **🤝 Clean Architecture + MVVM**
  - Foundation + Presentation Guidance
  - Complete architectural approach from domain to UI
- **💡 Remember Those Problems?**
  - 124% slower development → Faster, predictable delivery
  - 15x more bugs → Robust, tested components
  - 23-42% wasted time → Focused, productive development
- **🎉 From Problems to Solutions**
  - 🎯 Clear Boundaries
  - 🧪 Enhanced Testability
  - 🔄 Framework Independence
  - 📈 Scalable Architecture
  - ⚡ Developer Productivity

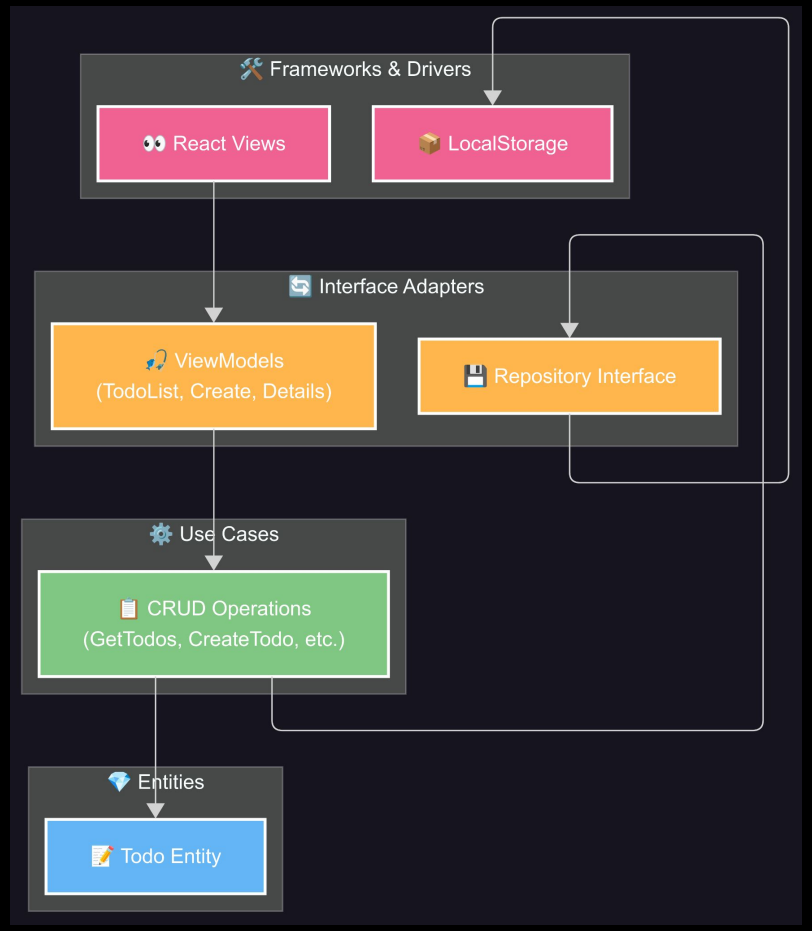
spaceteams

Practical Implementation: Building a TODO App

📝 TODO App Overview

- **🎯 From Theory to Practice**
  - TODO app demonstrating every concept
- **🛠️ What We'll Build**
  - ⚙️ Core CRUD Operations
  - 🖥️ Three-View Structure
  - 🏗️ Full Clean Architecture
- **👀 What You'll See**
  - Theory → TypeScript code with proper layer separation
  - MVVM with React hooks + Dependency injection with Awilix

🗺️ Clean Architecture Mapping & Flow



✦ 🏗️ Layer Mapping

Clean Architecture Layer	TODO App Components
Entities	Todo business object
Use Cases	CRUD operations (GetTodos, CreateTodo, etc.)
Interface Adapters	ViewModels + Repository Interface
Frameworks & Drivers	React Components + LocalStorage

✦ 🔄 Data Flow

- User Action → View → ViewModel → Use Case → Repository → Storage

💉 DI Setup & Project Structure

✦ 📁 Project Structure

```
src/
├── domain/          # Entities + Use Cases
│   ├── model/       # Todo.ts
│   └── useCases/    # createTodoUseCase.ts, etc.
├── adapter/         # Repository implementations
├── presenter/       # ViewModels + React components
└── di/              # Awilix container
```

Layer	Folder(s)	Responsibilities	Examples
Entities	src/domain/model	Pure business objects and interfaces	Todo.ts, ITodoRepository.ts
Use Cases	src/domain/useCases	Application-specific business rules	createTodoUseCase.ts
Interface Adapters	src/adapter, src/presenter	Repository implementations, ViewModels	todoRepository.ts, ViewModels
Frameworks & Drivers	React components inside presenter/pages	Framework-specific implementation	React components, e.g. TodoList.tsx

✦ 🏗️ Awilix DI Container

```ts
// src/di/container.ts
export const DI = createContainer();

DI.register({
  // repository
  todoRepository: asFunction(todoRepository),

  // use cases
  createTodoUseCase: asFunction(createTodoUseCase),
  deleteTodoUseCase: asFunction(deleteTodoUseCase),
  getTodoUseCase:    asFunction(getTodoUseCase),
  getTodosUseCase:   asFunction(getTodosUseCase),
  updateTodoUseCase: asFunction(updateTodoUseCase),

  // view-models
  todoListViewModel:     asFunction(createTodoListViewModel),
  todoDetailsViewModel:  asFunction(createTodoDetailsViewModel),
  createTodoViewModel:   asFunction(createCreateTodoViewModel),
});

// Usage
const { todos, deleteTodo } = DI.resolve('todoListViewModel');
```

👀 View

```tsx
import { FC } from 'react'
import { useNavigate } from 'react-router-dom'
import { DI } from '../../../di/ioc.ts'

export const TodoList: FC = () => {
  const navigate = useNavigate()

  const { todos, deleteTodo, showDeleteDialog, closeDeleteDialog, todoToDelete } =
DI.resolve('todoListViewModel')

  return (
    <>
      <Page
        headline="TODOs"
        footer={<Button customStyles={styles.button} label="+" onClick={() =>
navigate('/todo/create')} />}>
        <List
          items={todos}
          onItemClick={todo => navigate(`/todo/detail/${todo.id}`)}
          onItemDelete={showDeleteDialog} />
      </Page>

      {todoToDelete !== undefined && (
        <DeleteTodoDialog
          open={true}
          todoName={todoToDelete.title}
          onConfirm={deleteTodo}
          onCancel={closeDeleteDialog} />
      )}
    </>
  )
}
```

ViewModel

```
import { useEffect, useState } from 'react'
import { Todo } from '../../../domain/model/Todo.ts'
import { Id, UseCase, UseCaseWithParams } from '../../../domain/model/types'

type Dependencies = {
  readonly getTodosUseCase: UseCase<Todo[]>
  readonly deleteTodoUseCase: UseCaseWithParams<void, Id>
}

export const todoListViewModel = ({ getTodosUseCase, deleteTodoUseCase }: Dependencies) => {
  const [todoToDelete, setTodoToDelete] = useState<Todo>()
  const [todos, setTodos] = useState<Todo[]>([])

  const showDeleteDialog = (todo: Todo) => setTodoToDelete(todo)

  const closeDeleteDialog = () => setTodoToDelete(undefined)

  const getTodos = async () => {
    const result = await getTodosUseCase.execute()
    setTodos(result)
  }

  const deleteTodo = async () => {
    if (todoToDelete !== undefined) {
      await deleteTodoUseCase.execute(todoToDelete.id)
      setTodos(todos.filter(todo => todo.id !== todoToDelete.id))
      closeDeleteDialog()
    }
  }

  const sortById = (prevTodo: Todo, todo: Todo) => prevTodo.id < todo.id ? -1 : prevTodo.id >
todo.id ? 1 : 0

  useEffect(() => {
    void getTodos()
  }, [])

  return { todos: todos.sort(sortById), deleteTodo, showDeleteDialog, closeDeleteDialog,
todoToDelete }
}
```

🔧 Business Logic Implementation

✦ 🔧 Simple CRUD Use Case

```
export const getTodosUseCase =
  ({ todoRepository }: Dependencies): UseCase<Todo[]> => ({
    execute: () => todoRepository.get(),
  })
```

✦ 💾 Repository Interface Implementation

```
export const todoRepository = (): ITodoRepository => {
  const get = (): Promise<Todo[]> => {
    try {
      const result = localStorage.getItem(COLLECTION_NAME)
      return result !== null ? JSON.parse(result) : []
    } catch (error) {
      return Promise.reject(error)
    }
  }
  ...
  return { get, getById, create, update, delete }
}
```

✦ 🏢 Complex Business Logic

```
const processOrderUseCase = ({ /* Repositories */ }) => ({
  execute: async (orderData: OrderData) => {
    // Multi-step workflow with business rules
    const customer = await customerRepository.getById(orderData.customerId)
    if (customer.status !== 'active')
      throw new Error('Customer account inactive')

    const reservations = await Promise.all(/* inventory checks */)
    const pricing = calculateOrderPricing(orderData.items, customer.tier)
    const payment = await paymentRepository.processPayment(
      customer.paymentMethod,
      pricing.total
    )

    return await orderRepository.create({
      ...orderData,
      pricing,
      paymentId: payment.id
    })
  }
})
```

✅ Complete Implementation

- 🏗️ **What We've Built**
  - All four architectural layers working together
  - MVVM pattern with ViewModels as React hooks
  - Dependency injection coordinating everything
  - Real data flow from UI to localStorage
- 🔗 **Source Code:** https://github.com/spaceteams/clean-architecture-inspired-react-template
- 🏛️ **Solid Foundation**
  - These patterns create maintainable, testable, flexible code that scales with complexity
- 💭 **Reality Check**
  - TODO example demonstrates the patterns
  - Real-world applications involve multiple user roles, external integrations & complex business workflows

spaceteams

Results: From Problems to Solutions

✨ From Problems to Solutions

🤔 The Problems We Identified

- **Tight coupling** making changes risky
- **Testing complexity** slowing development
- **Scalability challenges** creating bottlenecks
- **Reduced maintainability** increasing costs

🎯 Clean Architecture + MVVM

- ✅ **Separation of concerns** → Independent, testable components
- ✅ **Clear boundaries** → Safe, predictable changes
- ✅ **Dependency inversion** → Flexible, swappable implementations
- ✅ **Structured growth** → Scales with complexity

⚖️ When to Use This Architecture

✅ Perfect For

- **Complex business logic** with multiple workflows
- **Multiple teams** working on different parts
- **Long-lived applications** that evolve over years
- **External integrations** requiring flexible data sources
- **Frequent requirement changes** across different domains

❌ Overkill For

- **Simple CRUD applications** with basic operations
- **Prototypes & MVPs** focusing on speed
- **Small teams** with tight communication
- **Short-term projects** with fixed requirements

🎯 Conclusion & Takeaways

📚 What We've Learned

- **Clean Architecture + MVVM** creates maintainable, testable React applications
- **Separation of concerns** makes code predictable & flexible
- **Dependency inversion** enables technology changes without business logic impact
- **Structured patterns** scale with complexity instead of fighting it

💡 Key Takeaways

- ✅ **Start simple, evolve thoughtfully** - Let business needs drive architectural decisions
- ✅ **Focus on problems** you're actually facing
- ✅ **Invest in long-term maintainability** when complexity justifies the trade-off

spaceteams

❓ Technical Questions

- **Q: Does Clean Architecture impact performance?**
  - A: Minimal impact. Dependency injection & layer abstractions add negligible overhead.
  - Benefits of maintainable, optimizable code far outweigh micro-performance costs.
- **Q: How do you test this architecture?**
  - A: Each layer tests independently:
    - Use Cases: Mock repository interfaces
    - ViewModels: Mock use cases, test state management
    - Views: Mock ViewModels, test UI rendering
    - Repositories: Test against real storage or mocks

❓ Technical Questions

- **Q: What about Redux/Zustand/React Query?**
  - A: They complement this architecture:
    - Redux/Zustand: Can replace ViewModel state management
    - React Query: Perfect for repository implementations
    - Clean Architecture: Provides structure, these handle specific concerns
- **Q: Can I migrate existing React apps gradually?**
  - A: Absolutely! Start with:
    - Extract business logic from components
    - Introduce repository pattern for API calls
    - Add ViewModels for complex components
    - Full Clean Architecture when justified

❓ Implementation & Alternatives

- **Q: Why Awilix over other DI solutions?**
  - A: Awilix is lightweight and React-friendly. Alternatives:
    - Manual injection through React Context
    - Custom DI for specific needs
    - Choose what fits your team's complexity.
- **Q: How does this scale with large teams?**
  - A: Excellent scaling properties:
    - Clear ownership: Teams own specific layers
    - Parallel development: UI, business logic, data teams work independently
    - Merge conflicts: Reduced due to clear boundaries
    - Onboarding: New devs understand structure immediately

❓ Implementation & Alternatives

- **Q: What about Next.js/server-side rendering?**
  - A: Works seamlessly:
    - Use Cases: Pure functions, SSR-friendly
    - Repositories: Can handle server/client data sources
    - ViewModels: Adapt to SSR hydration patterns
    - Clean boundaries: Make SSR implementation cleaner
- **Q: Learning curve for the team?**
  - A: 2-3 weeks for comfort with patterns. Investment pays off:
    - Week 1: Understanding layer boundaries
    - Week 2: Writing new features following patterns
    - Week 3: Confident refactoring and testing