### **捲簾估價應用程式 - 專案開發計畫與技術架構 v2.3 (修訂版)**

文件版本：2.3

最後更新日期：23/08/2025

### **第一章：專案開發方法論**

#### **1.1 核心理念：由外而內，混合驅動開發**

本專案採用「視覺驅動」與「架構驅動」相結合的混合開發模式 1。我們先透過視覺探索來定義使用者需求與最終產品樣貌，然後以穩健的技術架構為基礎，進行功能開發 2。這確保了產品既符合使用者預期，又具備良好的擴展性和維護性 3。

#### **1.2 開發流程四階段**

* **第一階段：視覺探索與需求確認 (UI/UX-First Exploration)** 4
  + 使用低保真原型（線框圖）來確定報價單的最終呈現樣式與所需欄位 5。
  + **產出**：核心的「項目清單」與「使用者故事」 6。
* **第二階段：核心架構與數據模型設計 (Data-First Foundation)** 7
  + 根據「項目清單」設計 StateManager 的數據結構 8。
  + 根據「使用者故事」確認各模組的核心職責與事件API 9。
* **第三階段：垂直切片開發與迭代 (Vertical Slice Development)** 10
  + 以單一功能（例如「新增手動捲簾」）為單位，垂直實現 UI、事件、狀態、計算等所有層面的程式碼 11。
* **第四階段：反覆迭代與反饋 (Iterate & Feedback)** 12
  + 對已完成的功能閉環進行測試與反饋，並持續進行下一個功能的開發 13。

#### **[新增] 1.3 架構維護與文件**

* **1.3.1 事件字典 (Event Dictionary)**
  + 本文件第四章的「應用程式事件 API」是整個系統的溝通契約。任何新的開發都必須遵循此契約。若需新增事件，必須先在此文件中進行註冊和說明，確保所有團隊成員對事件的流動有共同的理解。
* **1.3.2 開發者規範 (Developer Guidelines)**
  + 團隊應建立明確的開發規範，包括：
    - **事件命名慣例**：例如，用戶意圖事件以 user... 開頭，數據流轉事件以名詞過去式結尾 (如 quoteCalculated)。
    - **模組職責**：嚴格遵守各模組的單一職責原則，禁止跨模組直接調用方法。
    - **程式碼註釋**：在關鍵的業務邏輯和狀態變更方法處留下清晰的註釋。
* **1.3.3 複雜度管理 (Complexity Management)**
  + 為了在專案擴大時有效管理事件流的複雜度，建議在開發過程中，於瀏覽器控制台建立一個簡易的「事件日誌紀錄器 (Event Logger)」。這個紀錄器可以訂閱所有事件，並將事件名稱和其附帶的數據 (payload) 按時間順序打印出來，這對於追蹤和除錯複雜的交互流程非常有幫助。

### **第二章：核心設計理念與整體架構**

#### **2.1 核心設計理念**

本應用程式的核心架構，基於\*\*「多分支輸入，單一出口合併 (Multi-Branch Input, Single-Exit Merge)」\*\*的設計理念 14。此設計旨在將不同產品（如捲簾、伸縮紗門）的複雜輸入流程，隔離在各自獨立的「視圖 (View)」中，但最終將所有數據匯集到一個統一的狀態管理中心和標準化的報價單輸出中 15。

此架構具備三大優勢：

* **高度可擴展 (Scalable)**: 未來新增任何產品，都只需新增對應的模組分支，無需改動核心架構 16。
* **使用者體驗佳 (User-Focused)**: 使用者在操作時，可以專注於單一產品的細節，避免資訊混淆 17。
* **易於維護 (Maintainable)**: 不同產品的邏輯被完美隔離，程式碼清晰且容易管理 18。

#### **2.2 整體模組架構**

應用程式由

**十個** 核心模組構成，它們透過一個中央的 **事件聚合器 (EventAggregator)** 進行通訊，實現了高度的解耦 19。

| 模組名稱 | 核心職責 | 暱稱 |
| --- | --- | --- |
| **StateManager.js** | 應用程式的數據核心，作為單一事實來源，並包含所有狀態變更的業務邏輯。 | 大腦 |
| **UIManager.js** | 負責根據狀態渲染正確的 UI 視圖。 | 臉孔 / 視覺導演 |
| **InputHandler.js** | 負責監聽並翻譯使用者輸入，將其轉換為具語義的「意圖事件」。 | 神經系統 / 事件翻譯官 |
| **[新增] ConfigManager.js** | 負責載入、儲存並提供所有業務相關的設定，例如價格表、計算參數等。 | 規則中心 / 知識庫 |
| **PriceCalculator.js** | 負責所有核心商業計價邏輯，並處理計算過程中的錯誤。 | 引擎室 / 總控制器 |
| **DataExporter.js** | 負責生成最終的報價單。 | 報告產生器 / 報告總監 |
| **EventAggregator.js** | 負責所有模組間的通訊。 | 溝通中樞 / 中央神經系統 |
| **NotificationService.js** | 負責向使用者顯示提示訊息，包含成功、錯誤或警告。 | 訊息廣播員 |
| **APIService.js** | 負責處理所有對外的網路請求，如獲取最新價格表或儲存報價單。 | 對外聯絡官 |
| **PersistenceManager.js** | 負責將應用程式狀態持久化儲存於瀏覽器，防止用戶數據意外遺失。 | 記憶管家 |

#### **2.3 使用者操作流程與模組互動**

此流程展示了加入健壯性狀態管理後，模組間更精確的互動方式 20。

1. **使用者進入頁面** 21
   * PersistenceManager 檢查瀏覽器 localStorage 中是否有暫存的報價單數據。若有，則發布 loadStateFromCache 事件 22。
   * StateManager 接收事件，恢復使用者上次的作業進度 23。
   * UIManager 根據恢復的狀態，顯示對應的視圖 24。
2. **使用者點擊 [捲簾]** 25
   * InputHandler 捕捉點擊 → EventAggregator 發布一個具業務意圖的事件：userSelectedProduct，數據為 { productType: 'rollerBlind' } 26。
   * StateManager 接收事件，執行內部名為 \_changeView 的方法，進行狀態更新 ui.currentView = 'rollerBlind' 27。
   * 狀態更新後，  
     StateManager 自動發布 stateChanged 事件 28。
   * UIManager 接收 stateChanged 事件，根據新狀態，渲染出專為捲簾設計的輸入介面 29。
3. **使用者輸入捲簾細節，點擊 [儲存此捲簾]** 30
   * InputHandler 捕捉輸入數據 → EventAggregator 發布 userSavedLineItem 事件，數據為 { productType: 'rollerBlind', formData: {...} } 31。
   * StateManager 接收事件，呼叫內部的 \_addLineItem 方法。在此方法內，進行數據的 **驗證** 和 **轉換**，確保其符合預期結構，然後才將一個新的物件加入 lineItems 陣列 32。
4. **使用者點擊 [生成報價單]** 33
   * InputHandler 捕捉點擊 → EventAggregator 發布 userRequestedQuoteGeneration 事件 34。
   * PriceCalculator 接收事件，遍歷 lineItems 陣列，計算出最終價格 35。
   * **錯誤處理範例**: 如果在計算某個「捲簾」價格時，發現其布料 fabric-A 的價格未定義，RollerBlindCalculator 會拋出錯誤。PriceCalculator 會捕捉此錯誤，並發布 calculationFailed 事件，數據為 { message: '捲簾 "布料A" 價格配置缺失，請聯繫管理員。' } 36。
   * NotificationService 接收到 calculationFailed 事件，並向用戶顯示錯誤訊息 37。
   * 若計算成功，  
     PriceCalculator 發布 quoteCalculated 事件，附帶計算結果 38。
   * StateManager 接收 quoteCalculated 事件，儲存計算後的價格 39。
5. **使用者點擊 [從伺服器更新價格]** 40
   * InputHandler 發布 userRequestedPriceUpdate 事件 41。
   * APIService 接收事件，向後端發起請求，同時發布 showNotification 事件（訊息："正在更新價格表..."） 42。
   * 請求成功後，  
     APIService 發布 priceListLoaded 事件，並附帶從伺服器獲取的最新價格數據 43。
   * ConfigManager 或 StateManager 訂閱此事件，將新價格更新到應用程式的配置中。

### **第三章：各模組詳細架構藍圖**

#### **3.1 StateManager.js (狀態管理模組)**

* **核心設計理念**： 作為應用程式的\*\*「單一事實來源」，並進化為包含「狀態變更邏輯」\*\*的中央處理器 44。所有狀態的修改都必須透過訂閱具業務語義的事件來觸發，確保了數據流的單向與可預測性 45。
* **同時，它也負責管理與非同步操作相關的 UI 狀態，例如載入中 (loading) 或閒置 (idle) 狀態。**
* **模組架構藍圖**：
* JavaScript

// StateManager.js (增強版)

class StateManager {

constructor(initialState, eventAggregator) {

this.state = {

...initialState,

ui: {

...initialState.ui,

loading: {} // 用於追蹤不同非同步操作的載入狀態

}

};

this.eventAggregator = eventAggregator;

}

init() {

// 訂閱來自 InputHandler 的業務意圖事件

this.eventAggregator.subscribe('userSelectedProduct', (data) => this.\_changeView(data.productType));

this.eventAggregator.subscribe('userSavedLineItem', (data) => this.\_addLineItem(data));

this.eventAggregator.subscribe('userUpdatedField', (data) => this.\_updateField(data));

// 訂閱來自 PriceCalculator 的數據更新事件

this.eventAggregator.subscribe('quoteCalculated', (summary) => this.\_saveQuoteSummary(summary));

// 訂閱來自 PersistenceManager 的狀態讀取事件

this.eventAggregator.subscribe('loadStateFromCache', (cachedData) => this.\_loadState(cachedData));

// [新增] 訂閱來自 APIService 的通用請求狀態事件

this.eventAggregator.subscribe('apiRequestStarted', (data) => this.\_setLoadingState(data.operation, true));

this.eventAggregator.subscribe('apiRequestFinished', (data) => this.\_setLoadingState(data.operation, false));

}

// --- 私有狀態變更方法 (Mutations) ---

\_changeView(viewName) {

this.state.ui.currentView = viewName;

this.eventAggregator.publish('stateChanged', this.state);

}

\_addLineItem({ productType, formData }) {

// \*\*核心增強點：數據驗證與轉換\*\*

const width = parseFloat(formData.width);

const height = parseFloat(formData.height);

if (isNaN(width) || isNaN(height) || width <= 0 || height <= 0) {

this.eventAggregator.publish('showNotification', { message: '請輸入有效的寬高尺寸。', type: 'error' });

return;

}

const newItem = { productType: productType, ...formData };

this.state.quoteData.lineItems.push(newItem);

this.eventAggregator.publish('stateChanged', this.state);

this.eventAggregator.publish('showNotification', { message: '已成功新增一項產品。', type: 'success' });

}

// [新增] 管理非同步操作的載入狀態

\_setLoadingState(operation, isLoading) {

this.state.ui.loading[operation] = isLoading;

this.eventAggregator.publish('stateChanged', this.state);

}

// ... 其他私有方法 ...

}

#### **3.2 UIManager.js (UI 管理模組)**

* **核心設計理念**： 作為一個\*\*「視覺導演」\*\*。其職責保持不變：專注於根據 state 渲染 UI，不包含任何業務邏輯 46。

#### **3.3 InputHandler.js (輸入處理模組)**

* **核心設計理念**： 作為一個\*\*「事件翻譯官」\*\*。其核心職責從「發布指令式事件」進化為「發布描述用戶行為的意圖事件」 47。它只說明「用戶做了什麼」，而不關心「應用程式應該如何反應」 48。

#### **[新增] 3.4 ConfigManager.js (設定管理模組)**

* **暱稱**： **規則中心 / 知識庫**
* **核心設計理念**： 作為\*\*「規則與數據的中央儲存庫」\*\*。它負責載入、儲存並提供所有業務相關的設定，例如各產品的價格表、計算公式的參數、可選配件列表等。PriceCalculator 等模組不再直接持有這些設定，而是向 ConfigManager 查詢。這使得未來更新價格或業務規則時，只需修改設定檔和 ConfigManager，而無需改動核心的計算邏輯。
* **模組架構藍圖**：
* JavaScript

// ConfigManager.js 模組架構藍圖

import { rollerBlindCalculator } from './calculators/RollerBlindCalculator.js';

import { flyscreenCalculator } from './calculators/FlyscreenCalculator.js';

class ConfigManager {

constructor(eventAggregator) {

this.eventAggregator = eventAggregator;

this.priceLists = {};

this.calculators = {};

}

init(initialConfig) {

this.priceLists = initialConfig.prices || {};

this.registerCalculator('rollerBlind', rollerBlindCalculator);

this.registerCalculator('flyscreen', flyscreenCalculator);

// 訂閱事件以動態更新價格

this.eventAggregator.subscribe('priceListLoaded', (payload) => this.updatePriceList(payload));

}

updatePriceList({ product, data }) {

this.priceLists[product] = data;

}

registerCalculator(productType, calculator) {

this.calculators[productType] = calculator;

}

getCalculator(productType) {

const calculator = this.calculators[productType];

if (!calculator) {

throw new Error(`未找到 ${productType} 的計算引擎配置。`);

}

// 將當前的價格配置傳遞給計算引擎

calculator.setConfig({ prices: this.priceLists[productType] || {} });

return calculator;

}

}

#### **3.5 PriceCalculator.js (計算模組)**

* **核心設計理念**： 採用\*\*「總控制器 + 專家計算引擎」架構，並增加了健壯的錯誤處理機制\*\* 49。
* **它自身不包含任何具體的價格數據或計算公式，而是作為一個協調者，從 ConfigManager 獲取對應的計算引擎和配置，來完成計算任務。**
* **模組架構藍圖**：
* JavaScript

// PriceCalculator.js (重構版)

class PriceCalculator {

constructor(configManager, eventAggregator) {

this.configManager = configManager;

this.eventAggregator = eventAggregator;

}

init() {

this.eventAggregator.subscribe('userRequestedQuoteGeneration', (quoteData) => this.calculateQuote(quoteData));

}

calculateQuote(quoteData) {

try {

const calculatedLineItems = quoteData.lineItems.map(item => {

// 動態獲取計算引擎

const calculator = this.configManager.getCalculator(item.productType);

const itemPrice = calculator.calculate(item);

return { ...item, price: itemPrice };

});

// ... 匯總總價的邏輯 ...

const summary = { /\* ... 最終價格 ... \*/ };

this.eventAggregator.publish('quoteCalculated', summary);

} catch (error) {

console.error("Price calculation failed:", error.message);

this.eventAggregator.publish('calculationFailed', { message: error.message });

this.eventAggregator.publish('showNotification', { message: `計算失敗：${error.message}`, type: 'error' });

}

}

}

#### **3.6 DataExporter.js (數據匯出模組)**

* **核心設計理念**： 採用\*\*「報告總監 + 專家格式化引擎」\*\*架構。職責保持不變 50。

#### **3.7 EventAggregator.js (事件聚合器)**

* **核心設計理念**： 實現\*\*「發布/訂閱 (Publish/Subscribe)」\*\*模式。職責保持不變，是整個架構的神經中樞 51。

#### **3.8 NotificationService.js (通知服務模組)**

* **核心設計理念**： 作為一個\*\*「訊息廣播員」\*\*，負責顯示所有類型的用戶提示，包括成功、資訊、警告與錯誤 52。

#### **3.9 APIService.js (API 服務模組)**

* **核心設計理念**： 作為\*\*「對外聯絡官」\*\*，它是應用程式中唯一負責處理 HTTP 網路請求的模組 53。
* **模組架構藍圖**：
* JavaScript

// APIService.js (增強版)

class APIService {

constructor(apiBaseUrl, eventAggregator) {

this.baseUrl = apiBaseUrl;

this.eventAggregator = eventAggregator;

}

init() {

this.eventAggregator.subscribe('userRequestedPriceUpdate', () => this.fetchLatestPrices());

this.eventAggregator.subscribe('userSavedQuoteToServer', (quoteData) => this.saveQuote(quoteData));

}

async fetchLatestPrices() {

const operation = 'fetchLatestPrices';

// [修訂] 發布更具體的請求開始事件

this.eventAggregator.publish('apiRequestStarted', { operation });

this.eventAggregator.publish('showNotification', { message: '正在從伺服器獲取最新價格...', type: 'info' });

try {

const response = await fetch(`${this.baseUrl}/prices/flyscreen`);

if (!response.ok) {

throw new Error('網路回應不正確');

}

const priceData = await response.json();

this.eventAggregator.publish('priceListLoaded', { product: 'flyscreen', data: priceData });

this.eventAggregator.publish('showNotification', { message: '價格表已更新。', type: 'success' });

} catch (error) {

this.eventAggregator.publish('apiErrorOccurred', { source: operation, error: error });

this.eventAggregator.publish('showNotification', { message: '無法獲取最新價格，請檢查網路連線。', type: 'error' });

} finally {

// [新增] 無論成功或失敗，都發布請求結束事件

this.eventAggregator.publish('apiRequestFinished', { operation });

}

}

async saveQuote(quoteData) {

// ... 儲存報價單到伺服器的邏輯 ...

}

}

#### **3.10 PersistenceManager.js (數據持久化模組)**

* **核心設計理念**： 作為\*\*「記憶管家」\*\*，在背景默默地將重要的應用程式狀態儲存到瀏覽器的 localStorage 54。它透過訂閱
* stateChanged 事件來觸發儲存操作，並在應用程式啟動時檢查是否有可恢復的數據，極大地提升了使用者體驗 55。

### **第四章：應用程式事件 API**

這份清單是我們整個應用程式的\*\*「溝通契約」\*\*，定義了所有將在模組之間傳遞的事件 56。

| 事件名稱 | 發布者 | 訂閱者 | 數據範例 | 說明 |
| --- | --- | --- | --- | --- |
| **--- 核心狀態事件 ---** |  |  |  |  |
| stateChanged | StateManager | UIManager, PersistenceManager | {...} (完整的 state 物件) | 狀態已更新，通知相關模組進行響應 (重繪或儲存) 57。 |
| showNotification | 任何模組 | NotificationService | { message: '已儲存', type: 'success' } | 請求向用戶顯示一條提示訊息 58。 |
| **--- 用戶意圖事件 ---** |  |  |  |  |
| userSelectedProduct | InputHandler | StateManager | { productType: 'rollerBlind' } | 用戶選擇了一種類型的產品以開始輸入 59。 |
| userSavedLineItem | InputHandler | StateManager | { productType: 'rollerBlind', formData: {...} } | 用戶儲存了一個新的產品項目 60。 |
| userRequestedQuoteGeneration | InputHandler | PriceCalculator | {...} (當前的 quoteData) | 用戶請求觸發最終的價格計算 61。 |
| userRequestedPriceUpdate | InputHandler | APIService | (無) | 用戶請求從伺服器獲取最新價格表 62。 |
| **--- 數據流轉事件 ---** |  |  |  |  |
| quoteCalculated | PriceCalculator | StateManager | { subtotal: 1200, ... } | 價格已計算完畢，通知 StateManager 儲存結果 63。 |
| priceListLoaded | APIService | ConfigManager | { product: 'flyscreen', data: [...] } | 最新的價格數據已從伺服器載入 64。 |
| loadStateFromCache | PersistenceManager | StateManager | {...} (緩存的 quoteData) | 從瀏覽器緩存中載入的報價單數據 65。 |
| **--- 錯誤/例外事件 ---** |  |  |  |  |
| calculationFailed | PriceCalculator | NotificationService | { message: '價格配置缺失' } | 在價格計算過程中發生錯誤 66。 |
| apiErrorOccurred | APIService | NotificationService | { source: 'fetchPrices', error: {...} } | 在與伺服器通訊時發生錯誤 67。 |
| **--- [新增] 非同步狀態事件 ---** |  |  |  |  |
| apiRequestStarted | APIService | StateManager | { operation: 'fetchLatestPrices' } | 一個非同步網路請求已開始，通知 StateManager 更新 UI 載入狀態。 |
| apiRequestFinished | APIService | StateManager | { operation: 'fetchLatestPrices' } | 一個非同步網路請求已結束（不論成功或失敗），通知 StateManager 解除 UI 載入狀態。 |

### **[新增] 第五章：測試策略**

#### **5.1 測試理念**

本專案的高度解耦架構為實施全面的自動化測試提供了絕佳的基礎。我們的測試策略旨在確保每個模組的正確性、模組間交互的穩定性以及用戶端到端流程的順暢性。

#### **5.2 單元測試 (Unit Testing)**

單元測試是我們品質保證的基石，專注於測試單一模組的內部邏輯。

* **測試目標**：
  + **PriceCalculator.js**：模擬不同的 ConfigManager 回傳值和 quoteData 輸入，驗證其是否能正確調用對應的計算引擎，並處理計算過程中拋出的錯誤。
  + **StateManager.js**：針對每個私有的狀態變更方法 (mutation)，創建一個初始狀態，模擬 EventAggregator 發布一個事件，然後斷言 (assert) StateManager 的最終狀態是否符合預期。
  + **計算引擎 (Calculators)**：例如 RollerBlindCalculator.js，這是測試的核心。提供各種尺寸、布料、配件的組合，驗證計算結果的準確性，特別是邊界條件和異常輸入。
  + **數據驗證邏輯**：StateManager 中的數據驗證邏輯（如檢查寬高是否有效）應被抽離成獨立的純函數，並對其進行單獨的單元測試。

#### **5.3 整合測試 (Integration Testing)**

整合測試專注於驗證模組之間的溝通是否正確。

* **測試場景範例**：
  1. **用戶儲存項目流程**：觸發 InputHandler 的一個儲存按鈕點擊事件 -> 驗證 EventAggregator 是否正確發布了 userSavedLineItem 事件 -> 驗證 StateManager 接收到事件後，其 state.quoteData.lineItems 陣列是否已正確更新。
  2. **計算失敗流程**：觸發 userRequestedQuoteGeneration 事件 -> 模擬 PriceCalculator 在計算中拋出錯誤 -> 驗證 NotificationService 是否正確接收到 calculationFailed 事件，並準備顯示對應的錯誤訊息。

#### **5.4 端對端測試 (End-to-End Testing)**

在開發週期的後期，我們將採用如 Cypress 或 Playwright 等工具進行端對端測試。這些測試會模擬真實用戶的操作流程（例如，從打開頁面、輸入客戶資料、新增三個捲簾、生成報價單、到最終匯出 PDF），從用戶的視角確保整個應用程式的功能是完整且正確的。