

第四章：Architecture 架構風格與 DDD 的協奏曲

目錄

- 1 開場
- 2 SaaSOvation 演進地圖
- 3 Layered Architecture
- 4 Hexagonal Architecture
- 5 SOA Architecture
- 6 REST architectural style
- 7 CQRS Pattern
- 8 Event-Driven Architecture
- 9 Data Fabric
- 10 結語

開場

*"Architecture should speak of its time and place,
but yearn for timelessness."* - Frank Gehry

- DDD 是一套以 **Bounded Context** 劃界的思維框架，而非單一實作架構。
- 架構 = **品質屬性** + **功能需求** 的平衡藝術。
- 架構應服務於 Domain 模型；勿讓技術層凌駕業務語言。
- 追求可演進，抵禦未知需求，才能保持系統韌性。

SaaSOvation 演進地圖

SaaSovation 架構演進

階段	架構	驅動	收穫	風險
Startup	Monolith (Layered)	MVP 速度	快速迭代	技術債
Scale-up	Hexagonal + CQRS	可測試性	關注點分離	邊界模糊
Enterprise	多 Context + SOA	團隊協作	去耦	協調成本

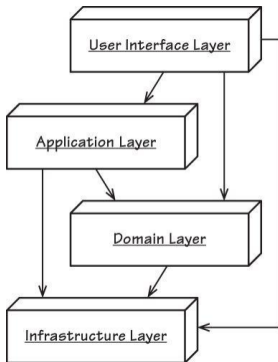
啟示

沒有一步到位的架構：**需求 × 風險 × 演進式重構** 才是真正的長青之道。

Layered Architecture

傳統分層架構 (Traditional Layers)

- 傳統分層架構：UI → Application → Domain → Infrastructure
- 每層只能依賴自己和下層
- 優點：結構清晰、好上手
- 問題：Repository 介面在 Domain，實作在 Infrastructure
- 缺陷：違反層級規則、難以測試、技術層凌駕業務層



依賴反轉原則 (DIP)

- 高階模組不應依賴低階模組，兩者都應依賴抽象
- 抽象不應依賴細節，細節應依賴抽象
- 解決方案：引入 DIP
- 邏輯上將 Infrastructure 層「移至上層」
- 實際上是讓 Infrastructure 實作 Domain 定義的介面

Layered Java (DIP)

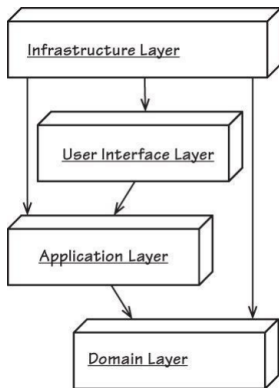
```
1 @Transactional
2 public void commitBacklogItemToSprint(
3     String aTenantId, String aBacklogItemId, String aSprintId) {
4     TenantId tenantId = new TenantId(aTenantId);
5     BacklogItem backlogItem =
6         backlogItemRepository.backlogItemOfId(
7             tenantId, new BacklogItemId(aBacklogItemId));
8     Sprint sprint = sprintRepository.sprintOfId(
9         tenantId, new SprintId(aSprintId));
10    backlogItem.commitTo(sprint);
11 }
```

Layered Java (DIP)

```
1 package
2 com.saasovation.agilepm.infrastructure.persistence;
3 import com.saasovation.agilepm.domain.model.product.*;
4 public class HibernateBacklogItemRepository
5 implements BacklogItemRepository {
6 ...
7 @Override
8 @SuppressWarnings("unchecked")
9 public Collection<BacklogItem>
10 allBacklogItemsComittedTo(
11 Tenant aTenant, SprintId aSprintId) {
12 Query query =
13 this.session().createQuery(
14 "from -BacklogItem as _obj_ "
15 + "where _obj_.tenant = ? and
16 _obj_.sprintId = ?");
17 query.setParameter(0, aTenant);
18 query.setParameter(1, aSprintId);
19 return (Collection<BacklogItem>) query.list();
20 }
21 ...
22 }
```

DIP：分層的變革

- 傳統分層架構是「高階元件依賴低階元件」
- DIP 顛倒依賴方向：Domain 定義介面，Infrastructure 實作
- Infrastructure 被「邏輯上」放在最上層
- 它是「實作者」，不是「控制者」
- 符合 DDD 精神：Domain 是中心，技術在外圍服務它

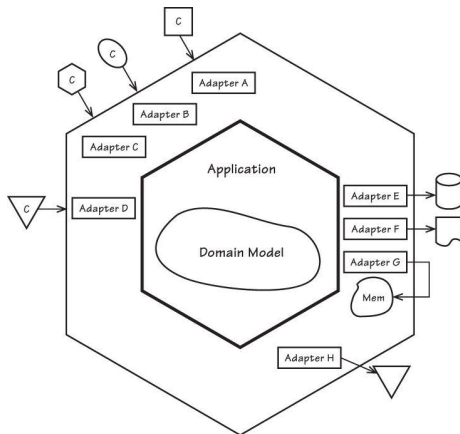


Hexagonal Architecture

Hexagonal 架構三大核心

- 1 所有依賴都「向內」指向應用核心 (Domain + Use Cases)
- 2 對外協定由 Adapter 處理 (如 HTTP、AMQP、JDBC)
- 3 Application Service 對 Port 發出請求，由 Adapter 實作

→ 避免耦合框架，實現技術可替換性、測試友善性、邊界清晰性

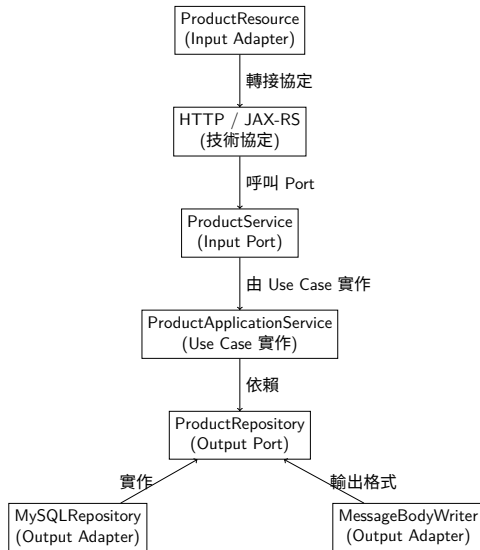


Hexagonal 架構 Input Port Java 範例

```
1 // === Input Adapter ===
2 @Path("/tenants/{tenantId}/products") // 接收 HTTP 請求，是 Input Adapter
3 public class ProductResource extends Resource {
4     // === Port ===
5     private ProductService productService; // 這是 Input Port (介面)
6     @GET
7     @Path("/{productId}")
8     @Produces({"application/vnd.saasovation.projectovation+xml"})
9     public Product getProduct(
10         @PathParam("tenantId") String aTenantId,
11         @PathParam("productId") String aProductId,
12         @Context Request aRequest) {
13         // 呼叫 Port 的方法，將外部資料導入應用層 (Port 實作中會使用 Domain Model)
14         Product product = productService.product(aTenantId, aProductId);
15         if (product == null) {
16             throw new WebApplicationException(Response.Status.NOT_FOUND);
17         }
18         // 回傳的是 Domain 物件，由 Output Adapter (MessageBodyWriter) 序列化為 XML
19         return product;
20     }
21 }
```


Hexagonal 架構的 Port 與 Adapter

依賴方向皆由外往內，所有 Adapter 實作 Port，所有 Port 為內部定義



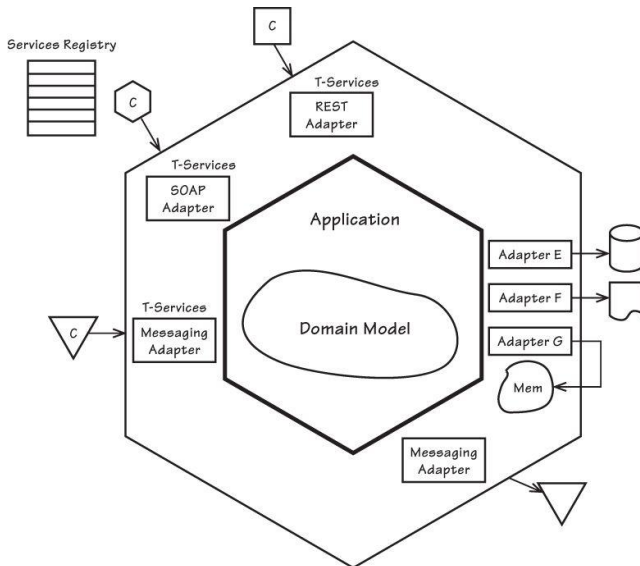
Hexagonal 架構的 DDD 價值

- 隔離技術細節，讓核心 Domain 不依賴外部框架
- 測試容易：可以在無 UI / 無 DB 下進行完整測試
- 高擴充性：新增一種協定只需新增對應 Adapter
- 是實踐 DDD 的最佳部署架構之一

SOA Architecture

- 每個 Bounded Context → 獨立服務 (Service)
- 技術介面可為 REST / SOAP / Messaging
- 不同技術服務可共享同一語意邊界（如一組 REST + Queue 為一 BC 的對外服務）
- 一個業務服務 = 多個技術服務 + 多個 Bounded Context
- 治理與韌性：Registry、Contract-First、Versioning、Circuit Breaker、Retry 等

Hexagonal Architecture supporting SOA, with REST, SOAP, and messaging services



REST architectural style

Contract-First Design

- 先編寫明確的介面定義（「合約」），再實作服務程式碼
- 合約成為單一事實來源：
 - **Provider**：依照合約生成或手寫服務端程式碼
 - **Consumer**：生成存根/SDK 或用於驗證與模擬
 - **Governance**：版本控制、相容性檢查、策略執行、文件

協定	合約格式	程式碼生成工具
REST/HTTP	OpenAPI 3 / AsyncAPI	Swagger Codegen, openapi-generator
gRPC	Protocol Buffers (.proto)	protoc, Buf, Grpc-Tools
SOAP	WSDL + XSD	wsimport, wsdl2java
Events	Avro Schema, AsyncAPI	avro-tools, Karate, Springs Cloud Stream

為何選擇 Contract-First ?

Code-First (實作優先)	Contract-First (合約優先)
API 常模仿內部類別結構；後續重構會破壞客戶端	API 專為外部需求設計；與內部模型解耦
客戶端團隊需等待服務端存根或模擬服務	合約早期發布 → 客戶端和服務端可並行工作
更難追蹤破壞性變更	可比較合約差異，執行語義版本檢查
較少保證；文件容易過時	程式碼、測試、文件、模擬都從單一來源生成

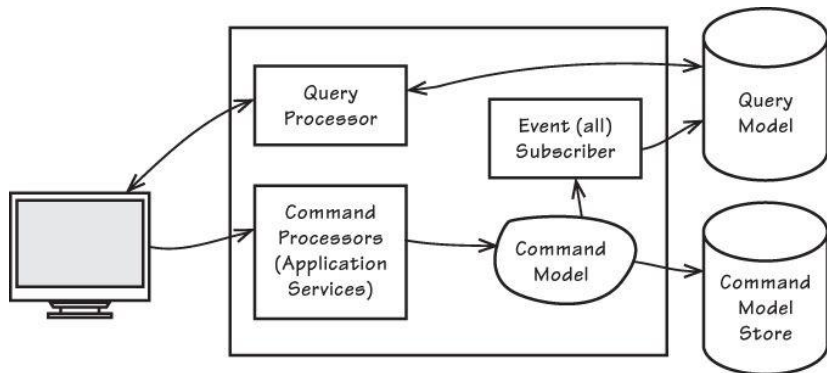
REST Java 實踐範例

```
1 @RestController
2 @RequestMapping("/orders")
3 class OrderController {
4     private final OrderService svc;
5     OrderController(OrderService svc) { this.svc = svc; }
6
7     @GetMapping("/{id}")
8     RepresentationModel<OrderDTO> get(@PathVariable String id) {
9         Order o = svc.findById(id);
10        OrderDTO dto = new OrderDTO(o.status().name());
11        dto.add(linkTo(methodOn(OrderController.class).get(id)).withSelfRel());
12        return dto;
13    }
14 }
```

CQRS Pattern

CQRS 核心概念

- Command Query 完全分離；寫入模型 (Write Model) 不直接用於查詢
- 寫入優先強一致；讀取可為最終一致並為效能/報表優化
- 常配 Event Sourcing；需要處理同步/異步投影更新



CQRS Java —Command Side

```
1 public record CreateUserCommand(String id, String name) {}  
2  
3 public class UserCommandHandler {  
4     private final UserRepository repo;  
5     public UserCommandHandler(UserRepository repo) { this.repo = repo; }  
6     public void handle(CreateUserCommand cmd) {  
7         repo.save(new User(cmd.id(), cmd.name()));  
8     }  
9 }
```

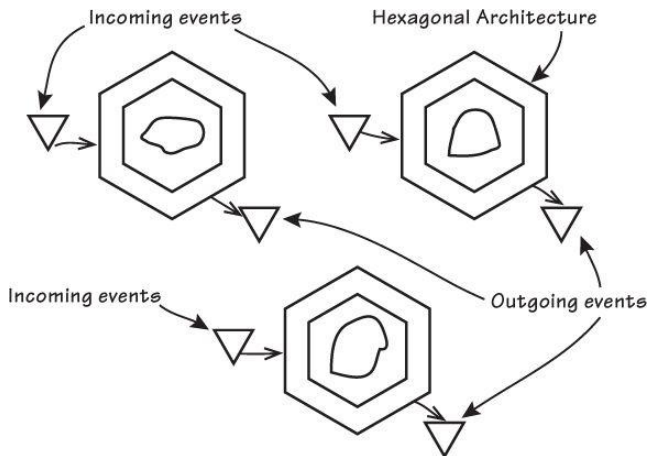
CQRS Java —Query Side

```
1 public class UserProjection {
2     @EventListener
3     public void on(UserCreated e) {
4         // 更新 Read Model (denormalized DB)
5     }
6 }
7
8 public class UserQueryService {
9     private final UserReadRepo repo;
10    public UserQueryService(UserReadRepo r) { this.repo = r; }
11    public UserDTO fetch(String id) { return repo.find(id); }
12 }
```

Event-Driven Architecture

事件驅動架構

- 服務間透過事件進行非同步通訊
- 鬆耦合、高擴展性、最終一致性
- 可結合事件溯源 (Event Sourcing) 模式

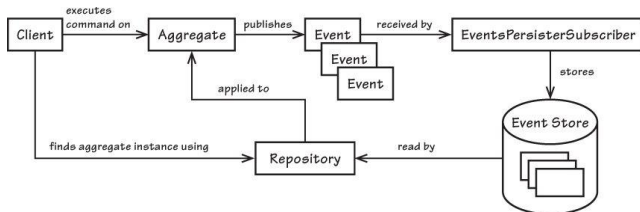


Saga 編排範例

```
1 class ShippingSaga {
2     @SagaEventHandler
3     void on(OrderCreated e) {
4         send(new ReserveInventory(e.id()));
5     }
6     @SagaEventHandler
7     void on(InventoryReserved e) {
8         send(new ArrangeShipment(e.orderId()));
9     }
10    @SagaEventHandler
11    void on(ShipmentArranged e) {
12        send(new MarkOrderShipped(e.orderId()));
13        end();
14    }
15 }
```


事件溯源模式

- 將 Aggregate 的每次狀態變更以事件持久化 (Append-Only)
- 可重播事件重建任何時間點狀態，支援審計與時間旅行
- 常與 CQRS / Saga 搭配使用



Event Sourcing Java 範例

```
1 public interface DomainEvent { Instant occurredAt(); }
2
3 public record OrderCreated(String id, Instant occurredAt) implements DomainEvent
4     {}
5
6 public class OrderAggregate {
7     private String id;
8     private OrderStatus status;
9
10    public static OrderAggregate reconstitute(List<DomainEvent> history) {
11        OrderAggregate agg = new OrderAggregate();
12        history.forEach(agg::apply);
13        return agg;
14    }
15    private void apply(DomainEvent e) {
16        if (e instanceof OrderCreated oc) {
17            this.id = oc.id();
18            this.status = OrderStatus.CREATED;
19        }
20    }
21 }
```

Data Fabric

- **統一數據平面**——同時支援 OLTP、OLAP、串流、快取等多種工作負載。
- 內建：智慧快取、聯邦查詢、一致性政策、可觀察性等能力。
- 分片設計建議與 DDD Aggregate 對齊，以提升資料本地性與處理效能。

Hazelcast——記憶體快取範例

```
1 Config cfg = new Config();
2 HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
3
4 IMap<String, OrderSummary> orders = hz.getMap("orders");
5
6 // 帶 TTL 寫入快取
7 OrderSummary summary =
8     new OrderSummary("ID-123", 1023, Instant.now());
9 orders.set(summary.id(), summary,
10            30, TimeUnit.MINUTES);
11
12 // 條件式快取查詢
13 Collection<OrderSummary> highValue =
14     orders.values(Predicates.greaterThan("total", 1_000));
```

Apache Ignite——分散式運算範例

```
1 Ignite ignite = Ignition.start();
2 IgniteCache<Integer, Tick> cache =
3     ignite.getOrCreateCache("ticks");
4
5 IgniteCallable<Double> valueAtRisk = () -> {
6     List<List<?>> rows = cache.query(
7         new SqlFieldsQuery(
8             "SELECT price FROM Tick WHERE pid=?")
9         .setArgs(portfolioId)
10    ).getAll();
11
12    List<Double> prices = rows.stream()
13        .map(r -> (Double) r.get(0))
14        .sorted()
15        .toList();
16
17    // 計算 5% Value-at-Risk
18    return prices.get((int) (prices.size() * 0.05));
19 };
20
21 Double var =
22     ignite.compute().call(valueAtRisk);
```

- **記憶體壓力**——高估熱資料集，恐導致 OOM；需加強 TTL 或 LRU 機制。
- **結構演進**——應採用 Schema Registry 並驗證相容性。
- **腦裂問題**——多區集群建議啟用 CP 模式或跨站複寫。
- **安全性**——敏感數據需加密（靜態、傳輸、運算中）並啟用稽核功能。

結語

- DDD 提供語意邊界，Hexagonal 讓依賴指向核心
- SOA / EDA / Saga 以事件串聯跨 Context 協作
- Event Sourcing + CQRS 帶來可追溯與高效查詢，需留意最終一致
- Data Fabric 為超大流量系統提供彈性快取與分散運算
- 選擇模式 \neq 潮流跟風：**從業務痛點出發，按需增量演進**

"Architecture is about the important stuff..." -Ralph Johnson

謝謝收看！

Slides by **Zhima Mochi**