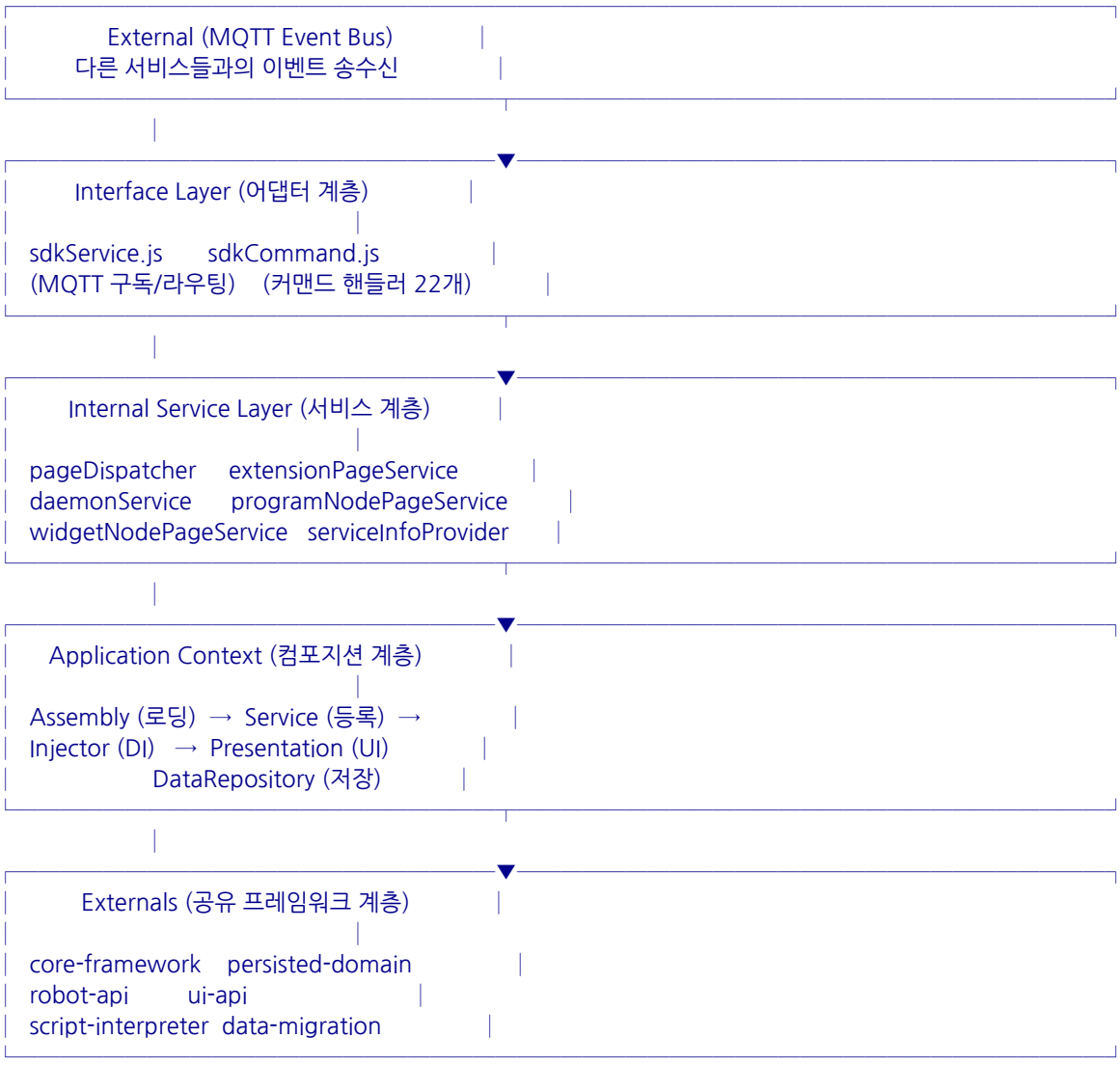
3단계: 아키텍처

설계 철학

rodi-x-svc는 다음 3가지 핵심 원칙으로 설계되어 있습니다:

1. 이벤트 기반 (Event-Driven): 모든 외부 통신은 MQTT Pub/Sub
2. 플러그인 아키텍처 (Plugin Architecture): ASAR 아카이브 기반 확장
3. 의존성 주입 (Dependency Injection): Reflection 기반 팩토리 DI

레이어 구조



핵심 디자인 패턴

1. Reflection 기반 의존성 주입 (DI)

이 프로젝트의 DI는 함수 파라미터 이름을 파싱하여 의존성을 해결합니다.

```
// bin/rodi-x-svc.js 에서 등록
moduleProvider.constant('assemblyPath', assemblyPath);
moduleProvider.factory('applicationContext', require('./modules/applicationContext/index'));

// modules/applicationContext/index.js 에서 사용
module.exports = function(assemblyPath, assemblyTempPath, utils, ...) {
    // ↑ 파라미터 이름 'assemblyPath'가 위에서 등록한 constant 이름과 일치
    // → 자동으로 주입됨
};
```

동작 원리:

1. moduleProvider.factory(이름, 팩토리함수) 호출

2. 팩토리함수의 파라미터 목록을 Reflection으로 파싱 (parseParam.js)

3. 각 파라미터 이름으로 DI 컨테이너(Dictionary)를 조회

4. 아직 등록 안 된 의존성이면 setInterval로 대기

5. 모든 의존성 해결 → 팩토리함수 실행 → 결과를 컨테이너에 저장

> 주의사항: 파라미터 이름을 변경하면 DI가 깨집니다. 이름 리팩토링 시 bin/rodi-x-svc.js의 등록명과 반드시 일치시켜야 합니다.

2. 이벤트 Pub/Sub (MQTT + Aggregator)

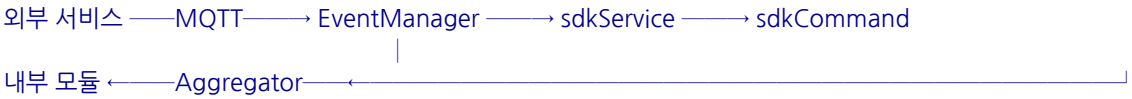
두 가지 이벤트 시스템이 공존합니다:

| 시스템 | 범위 | 용도 | 접근 방법 |

|--------|------|------|----------|

| EventManager (MQTT) | 서비스 간 | 외부 서비스와의 통신 | self.$eventManager.subscribe/publish |

| EventAggregator | 서비스 내부 | 모듈 간 내부 이벤트 | self.$aggregator.on/emit |



3. 플러그인 컴포지션 패턴

플러그인 하나가 서비스로 등록되면, 여러 객체가 조합되어 PresentationComposite를 형성합니다:

```
PresentationComposite
 ├── contribution      ← 플러그인이 제공한 서비스 인스턴스
 ├── pageHandler       ← JSDOM 기반 HTML 페이지 관리
 ├── dataModel         ← 이벤트 기반 Key-Value 데이터
```

```
├── rodiAPI           ← 14개 도메인 모델 접근 객체
├── scriptVariableMap  ← 스크립트 변수 저장소
├── scriptFunctionMap  ← 스크립트 함수 저장소
└── programNodeMap     ← 프로그램 노드 저장소
```

4. Factory-Register-Selector 패턴

PresentationCore는 3단계로 동작합니다:

```
Factory (생성)
  └──→ PresentationComposite 생성
       └──→ PageHandler + DataModel + Contribution 조합

Register (등록)
  └──→ PresentationCollection (Map) 에 key로 저장

Selector (조회)
  └──→ key 또는 서비스 타입으로 검색
```

5. 커맨드 패턴

모든 외부 요청은 커맨드 객체 형태로 처리됩니다:

```
// sdkCommand.js 내부
commands.rodix_open_page = function(data, finishCb) {
    let pageInfo = pageDispatcher.openPage(data.key);
    if (finishCb) finishCb(pageInfo);
};
```

- data: 요청 데이터
- finishCb: 완료 콜백 (MQTT ack 응답)

핵심 전역 객체

core-framework가 제공하는 전역 객체들이 모든 모듈에서 self.로 접근됩니다:

| 객체 | 타입 | 역할 |

|------|------|------|

| self.$eventManager | EventManager | MQTT Pub/Sub 관리 |

| self.$aggregator | EventAggregator | 내부 이벤트 Pub/Sub |

| self.$logger / $logger | Logger | Winston 기반 로깅 |

| self.$loggerData | Logger | 사용자 데이터 전용 로깅 |

| self.$dbDataSyncObject | DataSync | 서비스 간 데이터 동기화 |

| self.$topologyManager | TopologyManager | 서비스 디스커버리 |

| self.$fileSystemManager | FileSystemManager | 파일 시스템 작업 |

| self.$q | Q (Promise) | 비동기 처리 (레거시) |

| self.$lodash / $lodash | Lodash | 유틸리티 라이브러리 |

| self.$utils | Utils | 커스텀 유틸리티 |

> $logger와 $lodash는 전역 변수로도 접근 가능합니다 (core-framework에서 global 등록).

데이터 흐름 요약

일반적인 요청 처리 흐름

1. 외부 서비스가 MQTT 토픽에 메시지 발행
   예: "rodix/open/page" 토픽으로 { key: "myPlugin-myExtension" }

2. EventManager가 수신 → "rodix_open_page" 이벤트로 변환

3. sdkService의 subscribeEvent()가 호출됨
   → sdkCommand["rodix_open_page"](data.data, callback) 실행

4. sdkCommand 핸들러가 비즈니스 로직 실행
   → pageDispatcher.openPage(key)

5. pageDispatcher가 PresentationComposite를 찾아서
   → contribution.openView() 호출
   → pageHandler.active = true 설정
   → updateList, lastUpdate, iconText 수집

6. 결과를 callback(finishCb)으로 응답
   → MQTT ack로 요청자에게 전달

플러그인 내부 데이터 변경 흐름

1. 플러그인 코드에서 dataModel.set(key, value)

2. dataModel이 변경 감지
   → notifyPropertyChangedCallback 호출

3. EventManager.publish('rodix_update_page', data)
   → MQTT로 UI(web-svc)에 변경 알림

4. 동시에 aggregator.emit('assembly.datamodel.changed')
   → 내부 모듈에도 변경 전파

다음 단계

아키텍처 패턴을 이해했으면, 4단계: 초기화 흐름에서 서비스가 시작될 때 실제로 무엇이 어떤 순서로 실행되는지 따라가봅니다.