4단계: 초기화 흐름

이 문서는 서비스가 시작될 때 어떤 코드가 어떤 순서로 실행되는지 따라갑니다. 코드를 직접 읽으며 이해하는 것을 권장합니다.

전체 초기화 시퀀스

```
Phase 1: 부트스트랩                [bin/rodi-x-svc.js]
  |
  ├── 1.1 시스템 검증
  ├── 1.2 설정 로드
  ├── 1.3 core-framework 초기화
  ├── 1.4 코어 모듈 생성 및 실행
  └── 1.5 모듈 등록 (DI)
        |
Phase 2: 모듈 초기화               [각 모듈의 factory 함수]
  |
  ├── 2.1 로봇 API 초기화          [index.js]
  ├── 2.2 applicationContext 초기화   [modules/applicationContext/index.js]
  |     ├── 플러그인 추출 (ASAR)
  |     ├── 어셈블리 로딩 (JS)
  |     └── Activator 시작
  └── 2.3 SDK 서비스 구독          [modules/interface/sdkService.js]
        |
Phase 3: 서비스 활성화             [MQTT 이벤트: rodix_initialize]
  |
  ├── 3.1 Extension 서비스 실행
  ├── 3.2 Daemon 서비스 실행
  └── 3.3 Widget 서비스 실행
```

Phase 1: 부트스트랩

파일: bin/rodi-x-svc.js

1.1 시스템 검증 (라인 1-3)

```
const systemFileChecker = require('../modules/checker/systemFileChecker')();
const { INtimeLicenseChecker } = require('../modules/checker/licenseChecker');
```

- 시스템 파일 무결성 검사 모듈 로드
- INtime 라이선스 검증 모듈 로드

1.2 설정 로드 (라인 14-26)

```
var configs = {
   framework : require(path.join(configPath, 'framework.json')),
   runtime   : require(path.join(configPath, 'runtime.json'))
```

```
};
convertConfigToTgos(configs);
```

실행 순서:

1. config/framework.json 로드 (MQTT, 로깅, 이벤트 토픽)
2. config/runtime.json 로드 (Python/Java 경로)
3. convertConfigToTgos() 호출:
   - TGOS 루트의 config.json에서 robotEnv, robotSpec, filePath 가져옴
   - configs.main 객체 생성 (robotEnv, robotSpec)
   - configs.filePath 설정 (dataRepository, robotConfig, system, logs)
4. Python/Java 경로 결정 (절대경로 또는 services.infra 기준 상대경로)

1.3 core-framework 초기화 (라인 28-29)

```
var framework = require(path.join(rcPath, 'core-framework'));
framework.initialize(configs.framework);
```

- core-framework 싱글톤 인스턴스 생성
- configs.framework 설정으로 초기화
- 전역 변수 등록: $logger, $lodash 등

1.4 코어 모듈 생성 및 실행 (라인 38-45)

```
var coreModules = framework.createCoreModules();
coreModules.$eventManager.run();                    // MQTT 연결 시작
coreModules.$dbDataSyncObject.run({ script: true }); // 데이터 동기화 시작
coreModules.$topologyManager.run();                 // 서비스 디스커버리 시작
coreModules.$fileSystemManager.addPath('INtime_license', ...);
```

생성되는 코어 모듈:

- $eventManager: MQTT 클라이언트 (127.0.0.1:1883)
- $aggregator: 내부 이벤트 버스
- $dbDataSyncObject: 서비스 간 데이터 동기화
- $topologyManager: 헬스체크/디스커버리 (heartbeat 토픽)
- $fileSystemManager: 파일 시스템 관리
- $errorHandler: 에러 핸들러
- $q: Promise 라이브러리

1.5 모듈 등록 - DI (라인 49-89)

moduleInit() async 함수에서 모든 모듈을 DI 컨테이너에 등록합니다:

등록 순서 (의존성 해결 순서와 다를 수 있음):

```
Constants (즉시 사용 가능):
  mainConfig, frameworkConfig, assemblyPath,
  assemblyTempPath, filePath, pythonPath, javaPath

Factories (의존성 해결 후 실행):
```

```
persistedApi    → (core-framework 모듈들에 의존)
robotApi        → (core-framework 모듈들에 의존)
uiApi           → (core-framework 모듈들에 의존)
index           → (robotApi, mainConfig, persistedApi, filePath)
xPluginExceptionHandler → (...)
applicationContext      → (assemblyPath, utils, robotApi, persistedApi, ...)
pageDispatcher          → (applicationContext, ...)
sdkCommand              → (applicationContext, pageDispatcher, ...)
sdkService              → (sdkCommand)
extensionPageService    → (applicationContext, ...)
programNodePageService  → (applicationContext, ...)
daemonService           → (applicationContext, ...)
widgetNodePageService   → (applicationContext, ...)
serviceInfoProvider     → (...)
utils                   → (...)
virtualCaller           → (...)
ipcLogger               → (...)
```

> 팩토리 등록 순서가 실제 실행 순서를 결정하지 않습니다. DI 컨테이너가 의존성이 모두 준비되면 자동으로 팩토리를 실행합니다.

Phase 2: 모듈 초기화

DI 컨테이너가 의존성을 해결하면서 각 모듈의 팩토리 함수가 실행됩니다.

2.1 로봇 API 초기화

파일: index.js

```
1. persisted-domain에서 Configuration과 System 데이터 로드
2. robot-api에서 APIModel, RDataModel 가져오기
3. 로봇 ID 읽기 (RDataModel.RobotData.GetRobotId())
4. 로봇 모델명 가져오기 (persisted-domain → getRobot().getModel())
5. 모델명에서 '_3GEN' 접미사 제거
6. 시뮬레이션 모드에 따라 DLL 선택
   - INTIME 환경: 실제 로봇 DLL
   - 그 외: 시뮬레이션 DLL
7. APIModel.systemMemoryCreate() → 시스템 메모리 초기화
8. APIModel.robotInfoInitialize() → 로봇 정보 초기화
9. 'robot_simulation_mode' 이벤트 구독 (런타임 모드 전환)
10. '$aggregator.emit("system.start")' → 시스템 시작 이벤트 발행
```

2.2 applicationContext 초기화

파일: modules/applicationContext/index.js

이것이 플러그인 로딩의 핵심입니다. initialize() 함수가 모듈 생성 시 자동 호출됩니다.

```
1. initializeContainer()
   └── CompositionContainer에 핵심 타입 등록
     (AssemblyDirectoryInfo, AsarAssemblyExtractor, JsAssemblyLoader,
      ServiceContext, ServiceValidator, rodix_api 등)

2. cleanupAssemblies()
```

```
   └── 이전 실행의 임시 폴더(plugins.temp) 정리

3. extractAsarAssemblies()
   ├── AssemblyDirectoryInfo로 plugins/ 디렉토리 스캔
   ├── .asar 파일 목록 수집
   ├── plugins.temp/ 아래에 임시 폴더 생성
   └── AsarAssemblyExtractor로 각 .asar 파일 추출

4. loadAssemblies()
   ├── 추출된 폴더에서 JS 어셈블리 탐색
   ├── JsAssemblyLoader로 AssemblyCatalog(Map)에 로딩
   │     └── 각 어셈블리: { name, activator, package, errorMessage }
   └── '$aggregator.emit("assembly.load")' 이벤트 발행

5. loadDebugAssemblies()
   └── plugins.debug/ 디렉토리에서도 동일하게 로딩 (개발용)

6. startActivator()
   └── AssemblyCatalog의 각 어셈블리에 대해:
      ├── ServiceContext 생성 (ServiceCollection, 어셈블리명, package.json)
      ├── assembly.activator.start(serviceContext) 호출
      └── 에러 발생 시 assembly.errorMessage에 기록
```

2.3 SDK 서비스 구독

파일: modules/interface/sdkService.js

sdkService 모듈이 생성되면 runServices()가 즉시 실행됩니다:

```
// 22개 MQTT 이벤트에 대해 구독 등록
self.$eventManager.subscribe('rodix_initialize', subscribeEvent);
self.$eventManager.subscribe('rodix_get_assemblies', subscribeEvent);
self.$eventManager.subscribe('rodix_open_page', subscribeEvent);
// ... 나머지 19개

// 테스트용 aggregator 구독도 동시에 등록
runTestService();
```

이 시점부터 서비스가 MQTT 이벤트를 수신할 준비가 완료됩니다.

Phase 3: 서비스 활성화

외부에서 rodix/initialize MQTT 이벤트가 오면 실제 플러그인 서비스들이 활성화됩니다.

파일: modules/interface/sdkCommand.js (라인 13-28)

```
commands.rodix_initialize = function(data, finishCb) {
   if(applicationContext.isServiceExecuted === true) {
      return finishCb();  // 이미 실행됨 → 스킵
   }
   applicationContext.isServiceExecuted = true;

   extensionPageService.runServices()    // 1. Extension 서비스 시작
   .then(function() {
      return daemonService.runServices(); // 2. Daemon 서비스 시작
   }).then(function(){
```

```
        return widgetNodePageService.runServices(); // 3. Widget 서비스 시작
    }).then(function() {
        if(finishCb) finishCb();          // 4. 완료 응답
    }).done();
};
```

실행 순서 (순차적):

1. Extension 서비스: UI 확장 페이지를 가진 서비스들 실행
2. Daemon 서비스: 백그라운드 프로세스(Python/Java/EXE) 시작
3. Widget 서비스: 대시보드 위젯 서비스들 실행
4. 완료 콜백 → MQTT ack 응답

> rodix_initialize는 한 번만 실행됩니다 (isServiceExecuted 플래그 체크).

종료 처리

파일: bin/rodi-x-svc.js (라인 31-36), index.js (라인 43-48)

```
프로세스 종료 시:
  1. process.on('SIGINT') / process.on('uncaughtException')
     → exitHandler() 호출
     → process.emit('exitBefore')

  2. index.js의 'exitBefore' 리스너
     → $aggregator.emit('system.exit')
     → 3초 후 process.exit()
```

초기화 시 코드 따라가기 체크리스트

실제로 코드를 읽어볼 때 이 순서를 따라가세요:

- [ ] bin/rodi-x-svc.js 전체 (97줄) - 부트스트랩
- [ ] externals/core-framework/index.js - DI 컨테이너 원리
- [ ] index.js 전체 (51줄) - 로봇 초기화
- [ ] modules/applicationContext/index.js (306줄) - 플러그인 로딩
- [ ] modules/interface/sdkService.js (102줄) - 이벤트 구독
- [ ] modules/interface/sdkCommand.js (295줄) - 커맨드 핸들러

다음 단계

초기화 흐름을 이해했으면, 5단계: 플러그인 시스템에서 플러그인이 어떻게 만들어지고, 어떤 API를 사용할 수 있는지 상세히 살펴봅니다.