

# 런타임 엔진 서버 안내서

AnyLink 7

**TMAXSOFT**

## 저작권 공지

Copyright 2025. TmaxSoft Co., Ltd. All Rights Reserved.

## 회사 정보

(주)티맥스소프트

주소 : 경기도 성남시 분당구 황새울로258번길 29, 티맥스수내타워 8-9층

기술 서비스 센터: 1544-8629

홈페이지: <https://www.tmaxsoft.com>

## 제한된 권리

이 소프트웨어(Tmax AnyLink®) 사용설명서와 프로그램은 저작권법과 국제 조약에 의해 보호됩니다. 사용설명서와 프로그램은 TmaxSoft Co., Ltd.와의 사용권 계약 하에서만 사용할 수 있으며, 사용설명서는 사용권 계약의 범위 내에서만 배포 또는 복제할 수 있습니다. 이 사용설명서의 전부 또는 일부를 TmaxSoft의 사전 서면 동의 없이 전자, 기계, 녹음 등의 수단으로 전송, 복제, 배포하거나 2차적 저작물을 작성할 수 없습니다.

이 소프트웨어 사용설명서와 프로그램의 사용권 계약은 어떠한 경우에도 사용설명서 및 프로그램과 관련된 지적 재산권(등록 여부를 불문)을 양도하는 것으로 해석되지 않으며, 브랜드나 로고, 상표 등을 사용할 권한을 부여하지 않습니다. 사용설명서는 오로지 정보 제공만을 목적으로 하며, 이로 인한 계약상의 직접적 또는 간접적 책임을 지지 않습니다. 또한 사용설명서 상의 내용이 법적 또는 상업적인 특정 조건을 만족시킬 것을 보장하지 않습니다. 사용설명서는 제품의 업그레이드나 수정에 따라 예고 없이 변경될 수 있으며, 내용상의 오류가 없음을 보장하지 않습니다.

## 상표 공지

Tmax AnyLink®는 TmaxSoft Co., Ltd.의 등록 상표입니다. 본 사용설명서에 기재된 모든 제품과 회사 이름은 각각 해당 소유주의 상표로서 참조용으로만 사용되며 반드시 상표 표시 (™, ®)를 하지는 않습니다.

## 오픈소스 소프트웨어 공지

본 제품의 일부 파일 또는 모듈은 다음의 라이선스를 준수합니다. : APACHE2.0, CDDL1.0, EDL1.0, OPEN SYMPHONY SOFTWARE1.1, TRILEAD-SSH2, Bouncy Castle, BSD, MIT, SIL OPEN FONT1.1

관련 상세 정보는 제품의 다음 디렉터리에 기재된 사항을 참고하시기 바랍니다. : \${AnyLink\_HOME}\AnyLink-licenses

## 유지 보수

구분	지원항목	서비스 내용
제품지원	패치 & 업그레이드	무상 패치 서비스 제공  메이저 버전 업그레이드 시 할인 혜택  웹 지원을 통한 패치 내역 제공
기술 지원 - 기본 서비스	장애 지원	장애 발생 시 원인 분석 및 조치  Service Desk팀 → 기술팀 → R&D의 3단계 장애 분석 및 조치
	일상 지원(온라인 지원)	E-mail, 전화, 원격, 웹 사이트 등 온라인 자원을 통한 질의 응답 서비스
	고객 맞춤 지원(방문 지원)	고객의 요청으로 수행하는 방문 지원 서비스
기술 지원 - 옵션 서비스	예방 지원	정기 점검을 통한 시스템 운영현황 보고 및 장애 예방  <ul style="list-style-type: none"> <li>관리자 또는 운영자의 요구사항 수렴</li> <li>운영 현황(시스템, 엔진 운영) 보고서 제공</li> <li>필요 시 시스템 개선 권장 사항 보고</li> </ul>
유지 보수 비용 및 기간	계약 시 별도 협의	계약 시 EOL/EOS 문서 제공

## 안내서 이력

제품 버전	안내서 버전	발행일	비고
AnyLink 7	3.1.2	2025-01-20	-
AnyLink 7	3.1.1	2023-03-13	-
AnyLink 7	2.1.4	2019-08-23	-
AnyLink 7	2.1.1	2017-03-24	-

# 목차

1. 소개	1
1.1. 개요	1
1.2. 주요 기능	1
1.3. 서버 구성	4
2. 런타임 엔진 서버	6
2.1. 서버 컴포넌트	6
2.1.1. 서비스 플로우 엔진	6
2.1.2. 어댑터	7
2.1.3. 멀티바인딩 라우터	8
2.2. 서버 아키텍처	8
2.2.1. 시스템 아키텍처	8
2.2.2. 딜리버리 채널	10
2.2.3. 런타임 엔진 서비스	12
2.3. 런타임 엔진 에러 처리	12
2.3.1. 딜리버리 채널	12
2.3.2. 서비스 플로우	13
2.3.3. 어댑터	13
3. 런타임 엔진 서버 리소스	15
3.1. 개요	15
3.2. 비즈니스 리소스	15
3.2.1. 거래 및 거래그룹	15
3.2.2. 거래 및 거래그룹 환경설정	19
3.2.3. Java 클래스 로더와 거래 트리 클래스 로딩 방식	19
3.3. 라이브러리	21
3.4. 어댑터 리소스	21
3.5. 업무시스템 정보 설정	22
3.5.1. 스레드 풀	24
3.5.2. 시스템 로깅	25
3.5.3. 클러스터 설정	25
3.5.4. 배포 정책 설정	26
3.5.5. 암호화 알고리즘 설정	27
3.5.6. 디버깅 모드 설정	27
4. 서비스 플로우 엔진	28
4.1. 서비스 플로우 다이어그램	28
4.1.1. 서비스 플로우 다이어그램 기본 요소	28
4.1.2. 서비스 플로우 병렬 처리	29
4.1.3. 다이어그램 그래프와 블록 구조	30
4.2. 기본 패턴	31
4.2.1. 기본 흐름 패턴	31

4.2.2. 확장된 분기 및 병합 패턴	33
4.3. 서비스 구현 방식과 서비스 플로우 형태	39
4.3.1. Proxy 서비스	39
4.3.2. Composite 서비스	40
4.3.3. 서비스 구현	40
4.4. 서비스 플로우 변수와 액티비티 파라미터	40
4.5. 서비스 액티비티와 파라미터	41
4.6. 매핑	42
4.7. 서비스 플로우 표현식	43
4.7.1. 구조적 표현식	44
4.7.2. 변수 표현	44
4.7.3. 매핑 표현식	45
4.8. 사용자 코드와 핸들러	47
4.9. 코릴레이션	49
5. 멀티바인딩 라우터	51
5.1. 개요	51
5.2. 멀티바인딩 룰	51
5.2.1. Value	52
5.2.2. WeightBased	52
5.2.3. TimeRange	53
5.2.4. Handler	53
부록 A: 서버 API	54
A.1. com.tmax.anylink.api 패키지	54
A.2. com.tmax.anylink.api.serviceflow 패키지	54
A.2.1. ActivityContext	55
A.2.2. ActivityErrorHandler	56
A.2.3. ActivityHandler	56
A.2.4. BlockContext	57
A.2.5. ErrorCodeMapper	57
A.2.6. UserMapping	58
A.2.7. ProcessContext	58
A.2.8. ProcessHandler	59
A.2.9. ServiceActivityHandler	60
A.2.10. UserActivity	60
A.2.11. VariableContext	61
A.3. com.tmax.anylink.api.multibinding 패키지	61
A.4. Default 추상 클래스 목록	61

# 1. 소개

본 장에서는 AnyLink 서버에 대한 기본 기능과 구성에 대하여 설명한다.

## 1.1. 개요

AnyLink는 크게 두 개의 서로 다른 역할을 하는 서버들로 구성된다.

- 런타임 엔진 서버(RTE, Runtime Engine Server)

서비스 플로우 엔진, 어댑터, 멀티바인딩 라우터 등 내부 컴포넌트로 구성되어 있으며, 실제 서비스를 구현하는 서버이다.

업무 요청을 처리하는 AnyLink 런타임 엔진 서버는 다음과 같은 컴포넌트들로 구성되어 있다.

- 서비스 플로우 엔진(Service Flow Engine)

요청을 처리하는 일련의 흐름을 정의하는 서비스 플로우를 실행시키는 엔진이다.

- 어댑터(Adapter)

다양한 프로토콜과 애플리케이션 등을 연계하여 내부적으로 표준화된 규격으로 변환해주는 컴포넌트이다. 어댑터에 대해서는 별도의 안내서에서 자세히 다룬다.

- 멀티바인딩 라우터(Multi-binding Router)

서비스 컴포넌트 내부적인 라우팅이 실행 시점에 동적으로 이루어질 수 있는 컴포넌트간 라우팅 기능을 담당하는 컴포넌트이다.

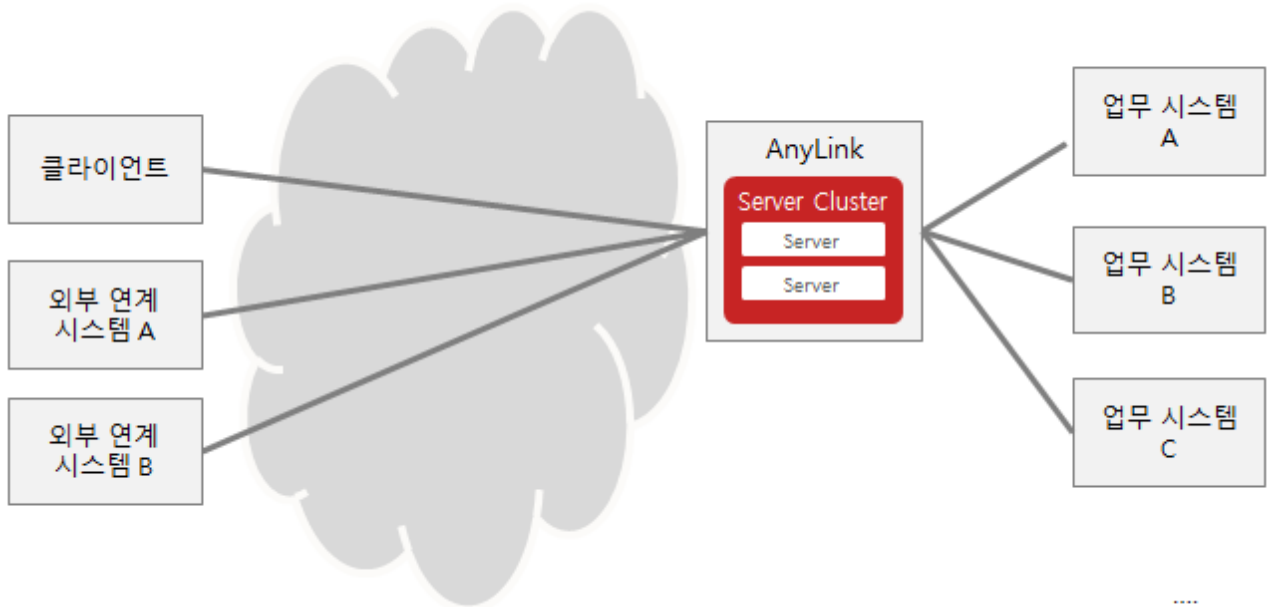
- 데이터 통합 서버(DIS, Data Integration Server)

배포 리소스 및 환경 설정 정보를 관리하는 서버이다. DIS는 런타임 엔진 서버로 전달되는 리소스들을 관리하고 소스 코드를 생성하거나 컴파일하는 역할을 수행한다. WebAdmin과 통합 스튜디오에 정보를 제공하는 서버이다.

## 1.2. 주요 기능

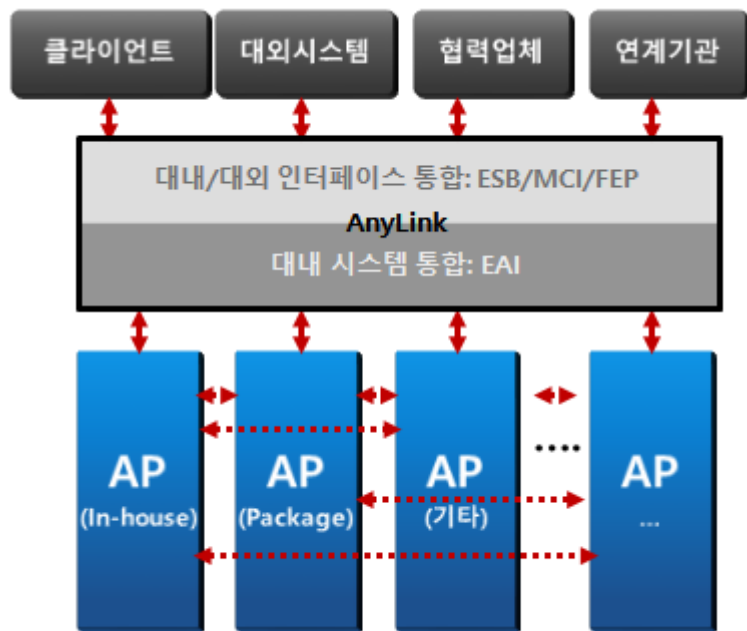
AnyLink는 서로 다른 시스템들 간의 연계 혹은 연동을 담당하거나 다양한 클라이언트에 대한 대외 서비스를 제공하는 역할을 한다.

AnyLink는 연계 오버헤드를 최소화해서 I/O 처리를 빠르게 처리하고 작업 대기 시간을 최소화한다. 또한 여러 서비스들의 흐름을 조율(orchestrate)하는 기능과 다양한 형태의 서버와 클라이언트의 요청과 응답을 처리하기 위해 프로토콜 및 애플리케이션 어댑터 기능이 있다.



AnyLink와 타 시스템 연계 구성도

AnyLink는 대내 연계를 하거나 내부 시스템의 세부 구조를 감추는 대외 통합 인터페이스 계층의 역할을 한다.

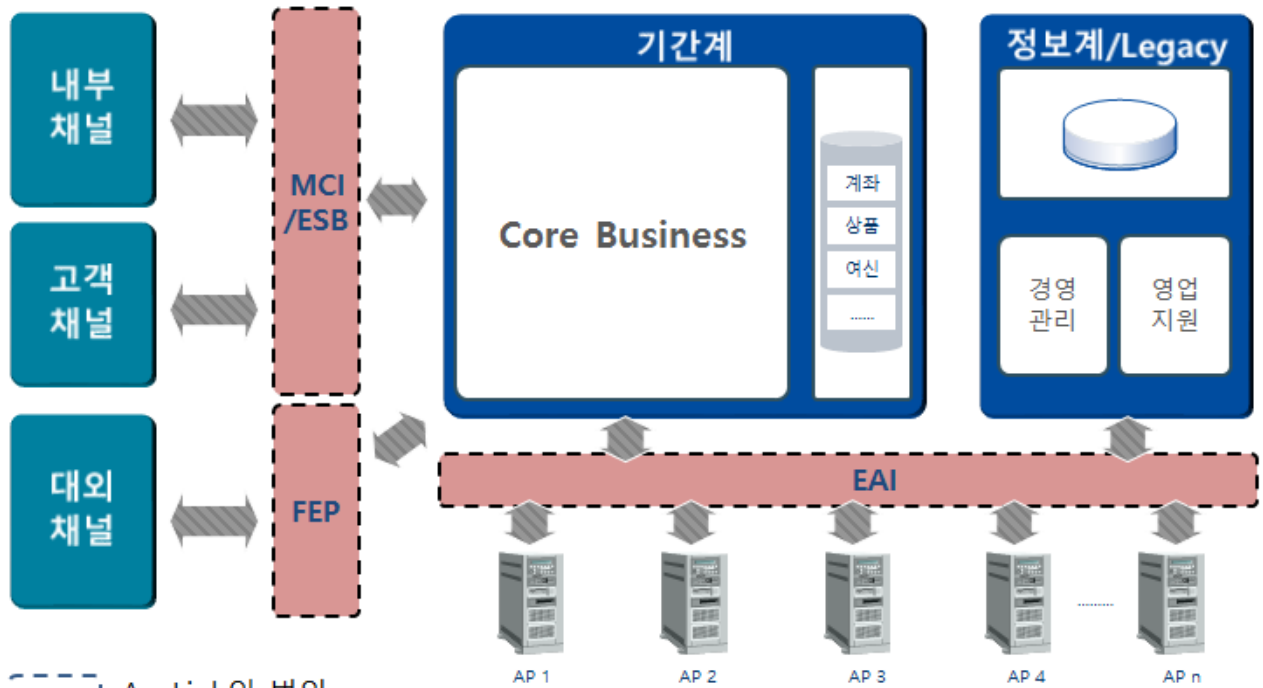


대외 연계 형태

다음은 AnyLink가 하는 계층역할에 대한 자세한 설명이다.

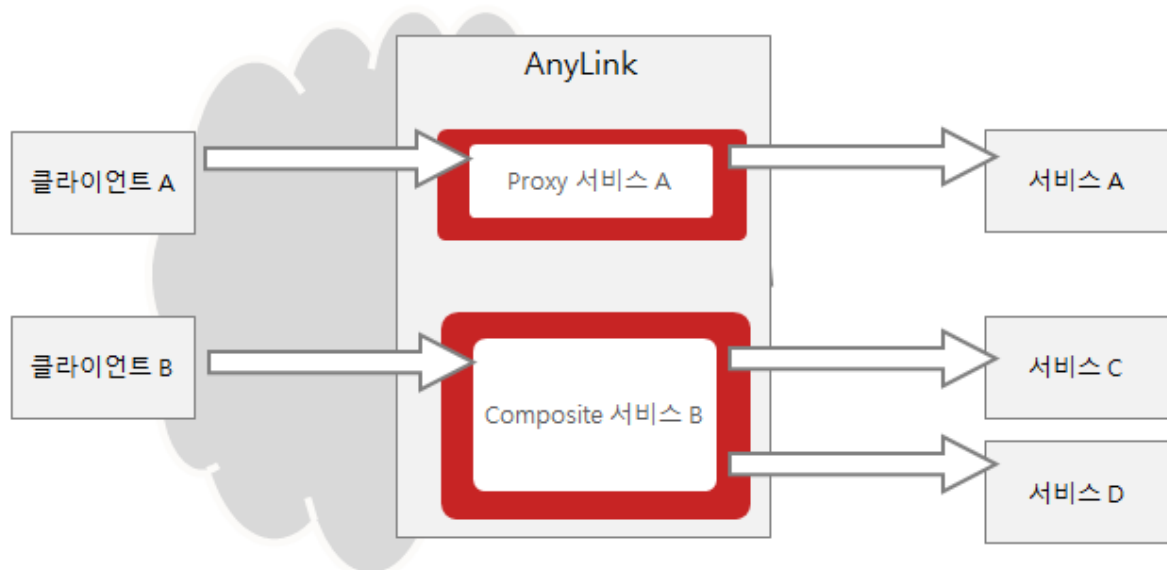
- 서비스 오케스트레이션

AnyLink는 전체 엔터프라이즈 시스템 아키텍처에서 제어를 담당하는 노드의 역할을 한다. AnyLink는 입출력과 이벤트 등을 처리하는 제어 노드로서 각 업무 처리 노드들과의 오케스트레이션을 담당한다.



엔터프라이즈 시스템 아키텍처에서 AnyLink 역할

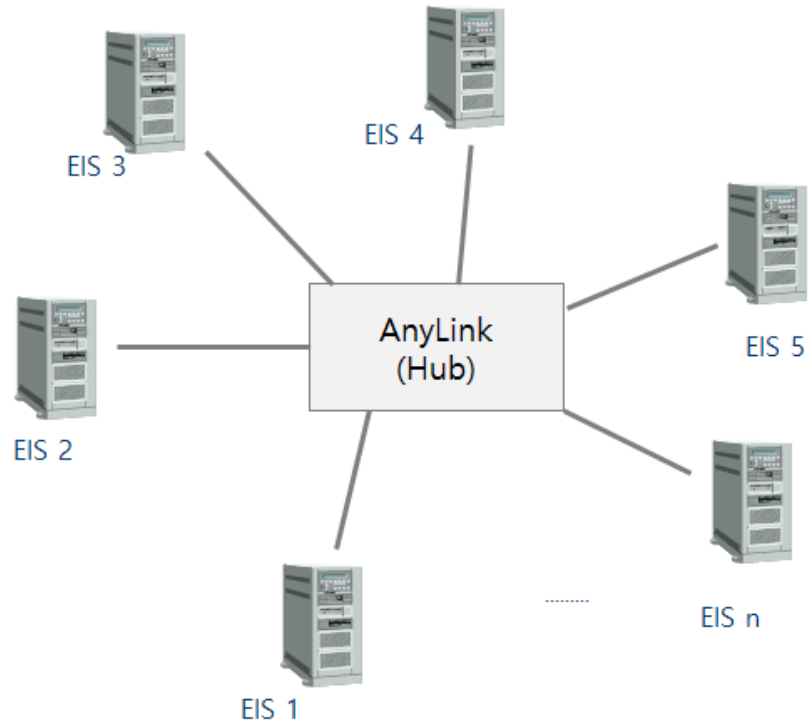
- Proxy 서비스
  - Wrapper 서비스
  - Composite 서비스



Proxy 서비스 형태

- 연계 서비스





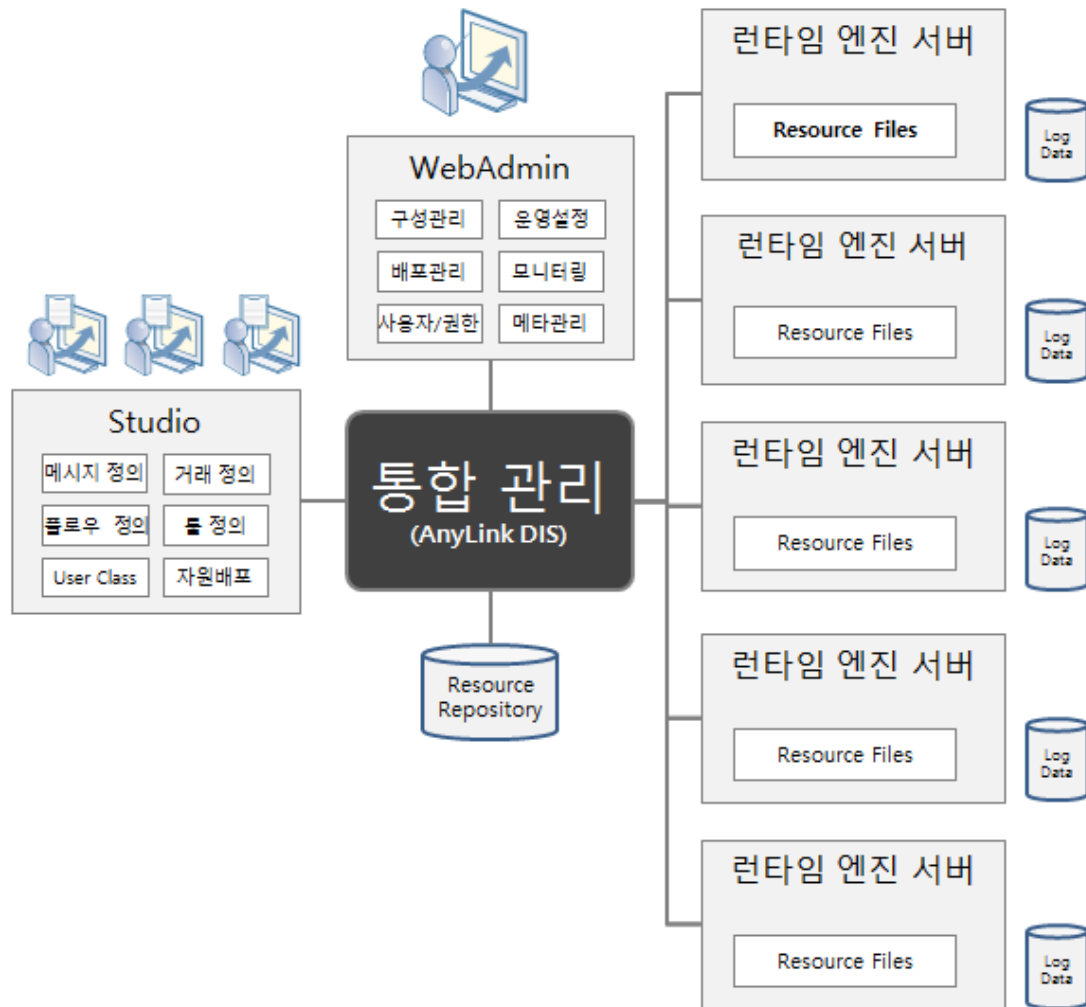
내부 연계 - Hub & Spoke 형태

- 자체 구현 서비스
  - 연계 및 업무 로직이 결합된 형태

## 1.3. 서버 구성

AnyLink는 크게 실행 환경과 개발 및 배포 환경 두 가지 형태로 구분해볼 수 있다. 실행 환경은 AnyLink 런타임 엔진 서버들과 연계 대상 시스템들로 구성되며, 연계 대상 시스템들의 구성과 형태에 따라 몇 가지 형태를 갖는다.

개발 및 배포 환경은 배포를 담당하는 서버인 데이터 통합 서버와 실행 서버인 런타임 엔진 서버 그리고 각종 룰과 환경 정보를 작성하는 클라이언트인 스튜디오와 WebAdmin으로 구성된다.



AnyLink 개발 및 배포 구성 형태

AnyLink는 개발 및 배포 서버인 데이터 통합 서버(DIS)와 운영 서버(RTE)를 분리하여, 실제로 운영을 하는 경우에는 배포 서버인 DIS의 기동 여부와 상관없이 RTE 서버가 서비스를 정상적으로 실행되도록 설계되어 있다.

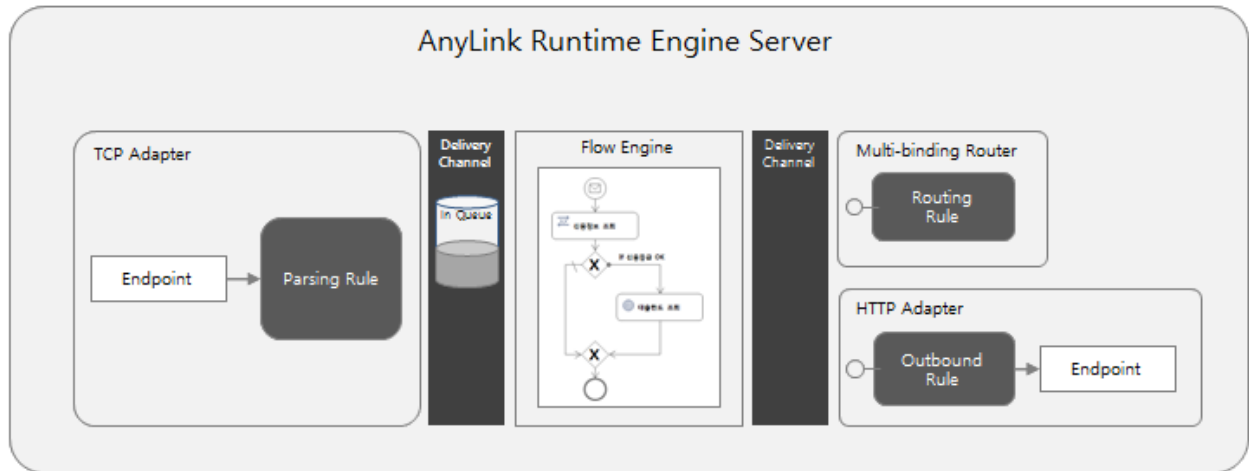
개발 및 배포를 담당하는 DIS는 업무 개발을 위한 통합 스튜디오와 운영 및 모니터링을 위한 WebAdmin에서 요청하는 각 명령들을 처리하는 역할을 한다. DIS는 스튜디오로부터 전달받은 개발 리소스들을 런타임 엔진 서버로 배포하고 이력 관리를 한다. DIS는 개발 리소스를 관리하고 로그 통계 등 모니터링 정보를 제공하기 위하여 RDBMS를 필요로 한다.

## 2. 런타임 엔진 서버

본 장에서는 AnyLink 런타임 엔진 서버의 컴포넌트와 아키텍처에 대해서 설명한다.

### 2.1. 서버 컴포넌트

AnyLink 런타임 엔진 서버는 역할에 따라 프로토콜별 I/O 처리를 담당하는 **어댑터**와 제어의 흐름을 결정하는 **서비스 플로우 엔진**, 여러 개의 서비스를 하나로 묶어서 처리할 수 있는 **멀티바인딩 라우터**로 구성된다.



AnyLink 컴포넌트 구성

#### 2.1.1. 서비스 플로우 엔진

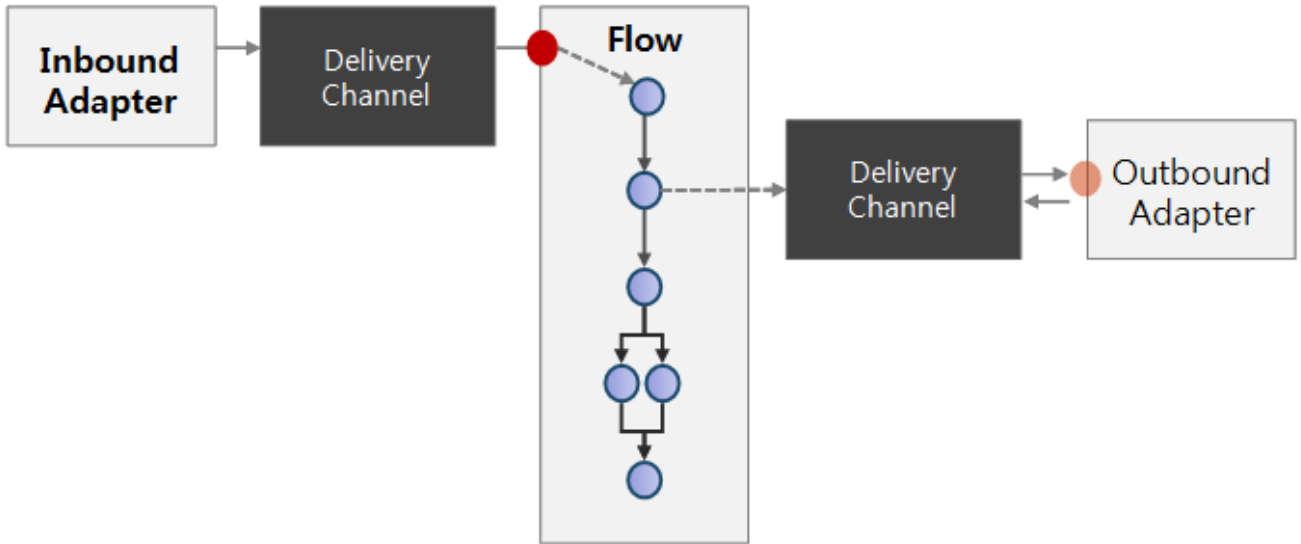
AnyLink 서비스 플로우 엔진(Service Flow Engine)은 이벤트에 의해 트리거되는 프로세스 실행 엔진이다. 서비스 플로우 엔진이 처리하는 이벤트는 어댑터 등의 다른 컴포넌트가 넘겨주는 메시지이다.

서비스 플로우는 비즈니스 프로세스를 모델링하는 데 사용되는 표준인 BPMN(Business Process Modeling Notation)을 차용한 다이어그램을 통해 프로세스의 흐름을 표현한다. BPMN은 OMG(Object Management Group)에서 정의하고 있는 비즈니스 프로세스를 표현하는 모델링과 표기법 표준으로 비동기적이며 동시성이 필요한 프로세스를 잘 표현할 수 있다.

서비스 플로우는 스튜디오를 통하여 액티비티와 이벤트로 구성된 다이어그램 형태로 정의하며, 프로세스 흐름 외에 변수, 표현식, 매핑, 핸들러, 사용자 액티비티 등의 기능을 추가로 제공한다.

서비스 플로우는 어댑터 등의 컴포넌트로부터 전달된 메시지에 의해 만들어지며, 다른 어댑터 혹은 서비스 플로우를 호출하거나 메시지를 주고 받을 수 있다. 또한 조건 분기, 선택적 실행, 동시 실행 등의 기능을 가지며, 흐름을 제어하기 위한 상태를 저장하는 변수 개념을 가진다.

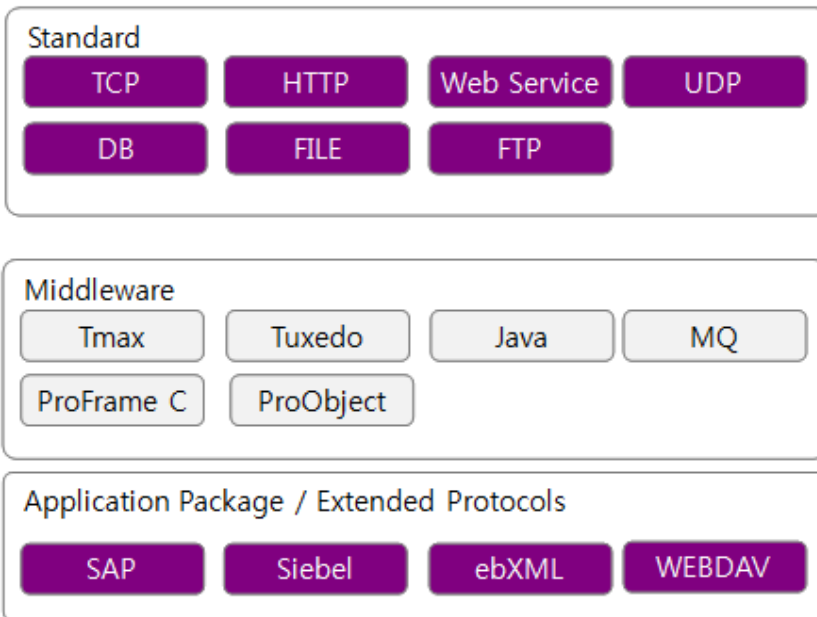
AnyLink 서비스 플로우 엔진은 비동기적이고 동시성이 필요한 프로세스를 잘 실행할 수 있는 특별한 아키텍처를 가지고 있으며, 엔진에 할당된 스레드 풀을 통해 프로세스를 실행한다.



AnyLink 서비스 플로우 엔진 트리거링

## 2.1.2. 어댑터

AnyLink 런타임 엔진 서버에서 어댑터(Adapter)는 AnyLink가 연계해야 하는 각 대상들과의 연계를 쉽게 해주는 역할을 한다. 연계 대상의 프로토콜에 맞추어 입출력 처리를 통한 연계를 하거나, 연계 대상 애플리케이션의 라이브러리를 사용한 연계 등을 실행한다.



프로토콜 어댑터

AnyLink 어댑터는 입출력의 방향에 따라 크게 인바운드 룰과 아웃바운드 룰 두 가지 룰을 정의할 수 있다.

- 인바운드 룰은 외부 시스템으로부터 AnyLink로 요청을 전달해주고, 그 결과 응답을 외부 시스템으로 돌려주는 역할을 한다.
- 아웃바운드 룰은 AnyLink에서 외부 시스템을 호출하고, 그 결과 응답을 받는 역할을 한다.

프로토콜에 따라 인바운드 룰의 요청 메시지와 아웃바운드 룰의 응답 메시지의 구분이 어려운 경우가 있다. 이를 효율적으로 처리하기 위해 AnyLink는 외부 시스템으로부터 들어오는 메시지 파싱 룰(parsing rule)을 통해서 인바운드 룰의 요청 메시지인지 혹은 아웃바운드 룰의 응답 메시지인지 여부를 판단할 수 있게 설계되었다.

### 2.1.3. 멀티바인딩 라우터

멀티바인딩 라우터(Multi-Binding Router)는 서비스 컴포넌트 내부적인 라우팅이 실행 시점에 동적으로 이루어질 수 있는 컴포넌트 간 라우팅 기능을 담당하는 컴포넌트이다.

AnyLink는 내부적으로 컴포넌트 간에 메시지를 주고받는 방식으로 다른 컴포넌트를 트리거링(triggering)한다. 기능의 이름과 특정한 형태의 요청 메시지 자료형과 응답 메시지 자료형, 오류 응답 메시지 자료형 등을 규격화하여 AnyLink에서는 서비스라고 한다.

AnyLink에서는 다음을 내부적으로 서비스로 사용한다.

- Receive 메시지 이벤트

서비스 플로우 정의의 Receive 메시지 이벤트는 어댑터로부터 메시지를 넘겨받는다.

- 아웃바운드 룰

어댑터의 아웃바운드 룰은 서비스 플로우로부터 메시지를 넘겨받는다.

- 멀티바인딩 룰

서로 다른 서비스를 묶어서 하나의 서비스처럼 그룹핑하기 위한 용도로 사용된다. 서로 다른 서비스 플로우의 메시지 이벤트, 어댑터의 서로 다른 아웃바운드 룰 또는 다른 멀티바인딩 룰 등을 그룹으로 묶어서 하나의 서비스처럼 표현할 수 있다. 멀티바인딩 룰은 여러 개의 서비스로 구성된 그룹을 조건에 따라 또는 규칙에 따라 특정 서비스로 분기하거나 여러 개의 서비스로 멀티캐스트하는 기능을 제공한다.

## 2.2. 서버 아키텍처

AnyLink 런타임 엔진 서버는 대외 시스템들과 통신을 담당하는 어댑터와 처리 로직을 담당하는 서비스 플로우 엔진, 큰 두 가지 서비스 컴포넌트와 다양한 서비스간 라우팅을 제공하는 멀티바인딩 라우터 서비스 컴포넌트로 구성되어 있으며, 컴포넌트들간의 통신과 룰 배포 등을 위한 **딜리버리 채널(Delivery Channel)**, **리소스 관리자** 등의 서비스가 정의되어 있다.

### 2.2.1. 시스템 아키텍처

본 절에서는 런타임 엔진 서버의 시스템 아키텍처에 대해서 설명한다.

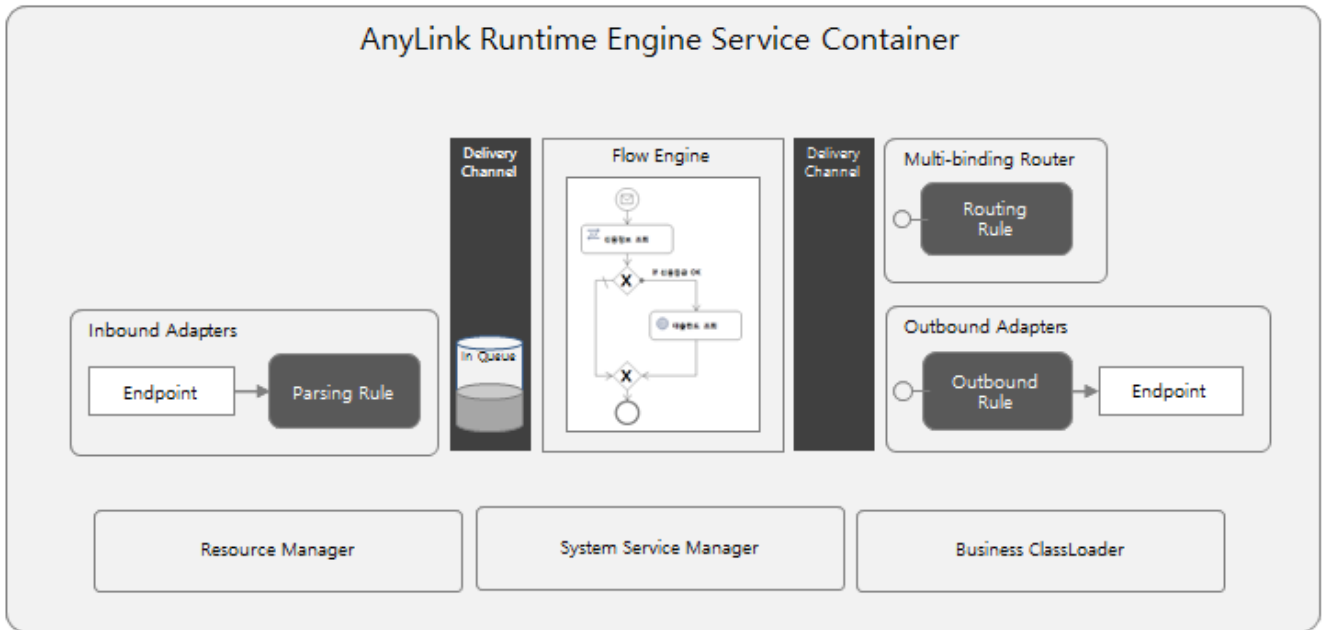
#### 런타임 엔진 서비스 컨테이너

런타임 엔진 서비스 컨테이너(RTE Service Container)는 컴포넌트의 라이프사이클을 관리하고 컴포넌트에 필요한 서비스들을 정의하고 제공하는 역할을 한다. 서비스 플로우 엔진, 멀티바인딩 라우터, 어댑터 관리자 등을

실행시켜주는 역할을 한다.

주요 서비스로 내부 통신 채널인 딜리버리 채널(Delivery Channel), 서비스 플로우 엔진과 어댑터 룰 등의 배포와 적재를 관리해주는 리소스 관리자, 스레드 풀과 로깅 등과 같은 시스템 자원을 관리하는 시스템 리소스 관리자, 각종 룰과 플로우에 사용되는 자바 코드들을 로드하기 위한 비즈니스 클래스 로더(Business ClassLoader) 등을 제공한다.

서비스 플로우 엔진 컴포넌트와 멀티바인딩 라우터 컴포넌트는 컨테이너별로 하나만 존재하며 어댑터 컴포넌트는 어댑터 종류별로 여러 개가 존재할 수 있다. 어댑터 컴포넌트는 어댑터 관리자가 기동 및 적재를 담당한다. 멀티바인딩 라우터 컴포넌트는 딜리버리 채널 내부에 존재하는 컴포넌트로 서비스를 라우팅할 때 서비스 ID 외에 다른 방식으로 라우팅을 하기 위한 컴포넌트이다.

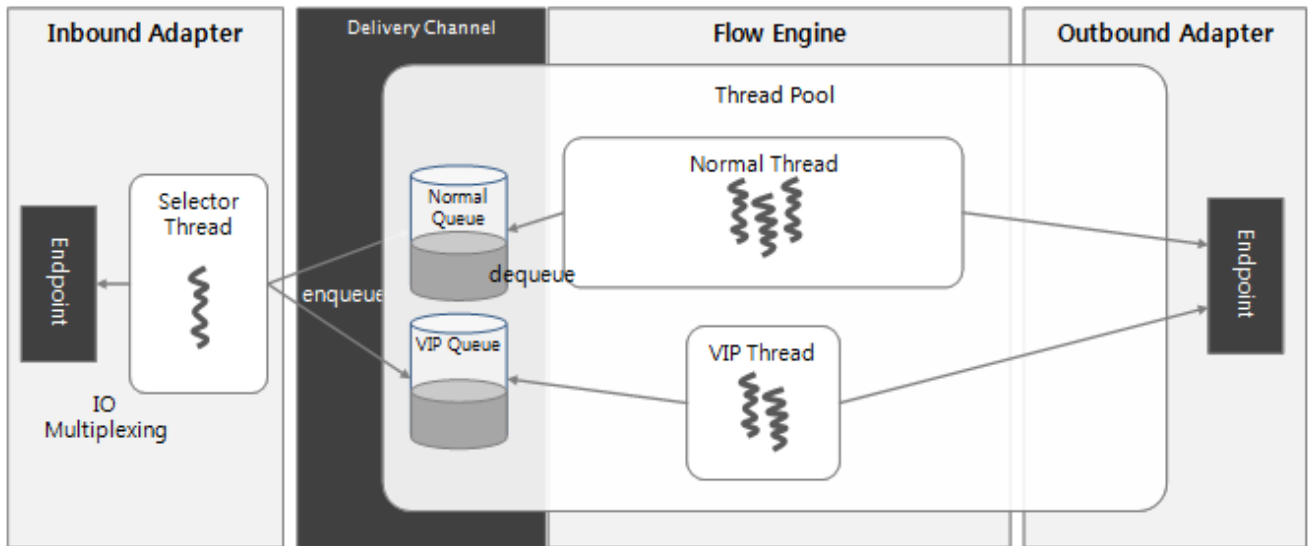


런타임 엔진 서비스 컨테이너 구성

## 엔진 아키텍처

AnyLink 엔진은 효율적으로 대량의 입출력 채널을 관리하는 I/O 구조와 스레드 자원의 낭비를 최소화하는 비동기적 프로세싱 구조를 가지고 있다.

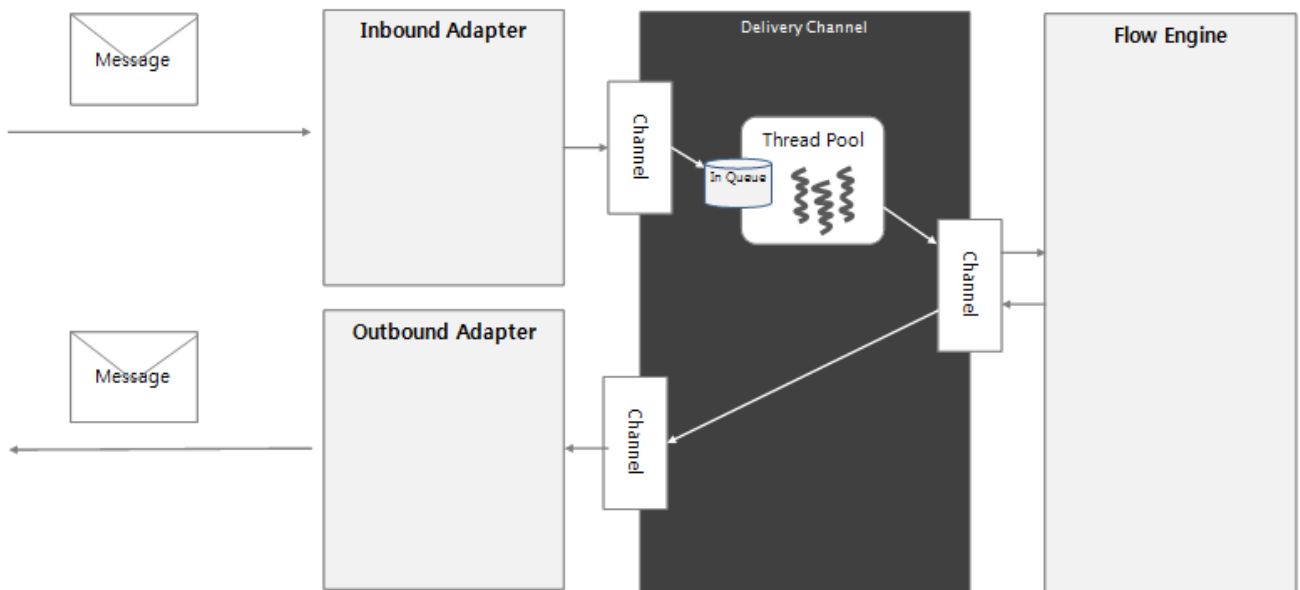
AnyLink 주요 통신 어댑터는 I/O 멀티플렉싱 방식의 입출력 처리를 하며, 특정 I/O 채널의 버퍼링에 다른 채널 처리가 영향받지 않도록 full non-blocking 방식의 처리를 구현하고 있다. 스레드 풀에서 주로 실행되는 서비스 플로우 엔진은 wait 시간을 최소화하도록 최대한 비동기적 특성을 가지도록 구현되어 있어서, 다수의 서비스 플로우가 동시에 실행되더라도 스레드 풀 부족으로 지체되지 않도록 고안되었다.



런타임 엔진 I/O 및 스레드 아키텍처

## 2.2.2. 딜리버리 채널

딜리버리 채널(Delivery Channel)은 미리 정의된 유형의 메시지 패턴을 통해 서비스 컴포넌트들간의 내부 통신을 비동기적인 방식으로 지원하는 서비스이다. 서비스 컴포넌트들 간의 의존성을 줄여주는 핵심 아키텍처이며, 메시징 API에 따라 적절한 스레드 풀을 사용하는 등의 기능을 가지고 있다.

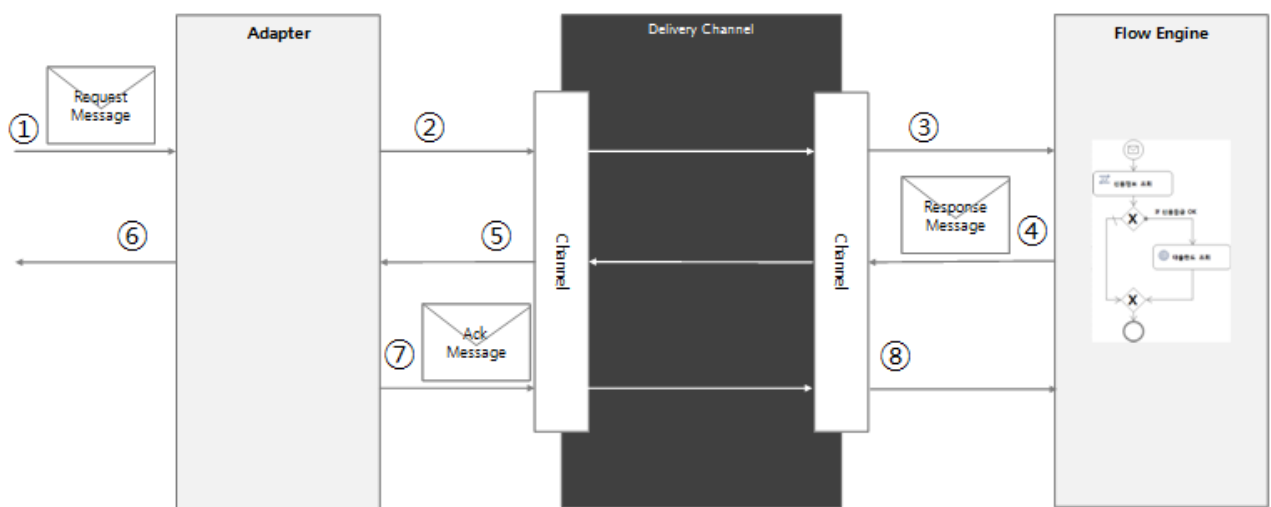


Delivery Channel 시스템 아키텍처

### • 메시지 패턴

딜리버리 채널은 서비스 컴포넌트들 간 내부 통신을 몇 가지 패턴 형태로 제공한다.

구분	설명
Request-Response(2way)	가장 일반적인 메시징 형태로 요청에 대한 응답을 기대하는 형태이다.
Oneway	응답을 기다리지 않고 요청 메시지만 전달하는 형태이다.
Oneway-ACK	요청 메시지가 처리된 후에 ACK 메시지를 기대하는 형태이다. 보통 딜리버리 채널에서 전달 보장을 처리한 후 ACK 메시지 혹은 NAK 메시지를 답변으로 보낸다.
Request-Response-ACK(3way)	요청, 응답 메시지 외에 응답에 대한 ACK 메시지가 추가된 형태이다. 응답 메시지를 잘 처리하였는지를 알려주는 ACK 혹은 NAK 메시지를 전달해줘야 한다.
Test-Message	요청 메시지를 처리할 수 있는지 여부를 질의하는 메시지 형태이다. 멀티바인딩 라우터에서 코릴레이션 매칭(Correlation Matching) 방식의 라우팅을 위해 사용한다. 코릴레이션 매칭에 대해서는 <a href="#">코릴레이션</a> 을 참고한다.



Request-Response-ACK 컴포넌트 통신

## • 신뢰 전송(Reliable Messaging) 및 XA 전송

딜리버리 채널은 OneWay 메시지 패턴을 사용할 경우 선택적으로 신뢰 전송과 XA 전송을 지원한다.

구분	설명
신뢰 전송	영구 저장 장치인 디스크나 RDBMS에 메시지를 저장한 후 ACK을 주는 방식으로 구현되어 있다.
XA 전송	Global Transaction을 지원하기 위해 지원하는데, 주로 Tmax 어댑터에서 Global Transaction이 걸려있는 상태에서 여러 개의 메시지가 전달될 경우에 사용한다. Tmax 어댑터를 통해 prepare, commit, rollbACK 등의 글로벌 트랜잭션 2-phase commit 관련 메시지를 처리하여 하나의 트랜잭션으로 묶여있는 여러 건의 메시지를 일괄 처리할 수 있도록 지원한다.

## • 트랜잭션 전파(Transaction Propagation)

딜리버리 채널은 메시지 속성에 따라 트랜잭션을 시작 혹은 종료하거나 트랜잭션 문맥을 전파하는 역할을 한다. 또한 스스로를 글로벌 트랜잭션의 XA 리소스 관리자 역할을 할 수 있다.



딜리버리 채널은 Tmax나 JEUS 등 트랜잭션 매니저(Transaction Manager)로부터 들어오는 메시지의 트랜잭션 문맥을 받아서 트랜잭션에 참여할 수 있는 기능을 제공하며, 그 반대로 스스로를 트랜잭션 매니저로 동작하여 AnyLink 내부적으로 트랜잭션을 시작하고 또 외부의 RDBMS와 같은 XA 리소스 관리자나 JEUS, Tmax 등으로 글로벌 트랜잭션을 전파할 수 있는 기능을 제공한다. 개별 트랜잭션의 설정은 서비스 플로우나 각 어댑터 룰의 속성으로 설정을 한다.

### 2.2.3. 런타임 엔진 서비스

런타임 엔진 서비스 컨테이너는 딜리버리 채널 외에도 런타임 운영에 필요한 서비스들 제공한다.

- 리소스 관리자

내부적으로 정의하는 업무의 거래 및 거래그룹에 속하는 각종 룰과 리소스, Java 코드들을 관리한다.

- 시스템 리소스 관리자

런타임 엔진이 사용하는 스레드 풀과 시스템 로깅을 관리한다.

- 비즈니스 클래스 로더

Java 코드 및 주요 리소스들의 계층 구조 및 Symbolic Link 등 의존성 처리를 구현한 Java 코드 클래스로더이다.

- 배포 관리자

데이터 통합 서버(DIS)로부터 여러 가지 리소스를 안정적으로 배포하기 위해 2-phase 배포를 구현하는 서비스이다. 데이터 통합 서버로부터 배포할 리소스를 넘겨받아 동적으로 배포를 실행하며, Undeploy(배포 해지)나 Redeploy(재배포)가 발생하면 기존의 리소스 파일은 백업 디렉터리로 이동시키고 삭제 혹은 갱신을 하게 된다. 런타임 엔진의 배포 서비스가 배포의 1단계(prepare)에서 오류를 발생시키면 DIS는 전체 서버로의 배포를 취소(rollback)시킨다.

## 2.3. 런타임 엔진 에러 처리

AnyLink는 여러 경우의 오류를 다루는 방법을 제시하고 있다. 어느 수준에서 발생하는 오류인가에 따라 용어도 다르고 처리하는 방식도 다르다. 본 절에서는 런타임 엔진에서 발생하는 예러의 처리방법에 대해서 간단히 설명한다.

### 2.3.1. 딜리버리 채널

AnyLink 런타임 엔진 내부의 서비스 컴포넌트 간 통신 채널인 딜리버리 채널에서 하나의 서비스 컴포넌트로 다른 서비스 컴포넌트로 메시지를 전달할 때 발생하는 에러 처리가 있다. 이 부분은 시스템 내부적인 오류이므로 AnyLink 사용자가 고려할 필요는 없지만, 내부적인 동작 방식을 이해하는 데 도움이 될 수 있다.

엔진 내부의 서비스 컴포넌트들 간의 통신을 담당하는 딜리버리 채널은 응답과 에러 응답을 구분하여 보내도록 구성되어 있다. 딜리버리 채널의 에러 응답은 서비스 컴포넌트에서 메시지를 처리하다가 내부적으로 오류가 발생하였을 때 이 에러를 응답으로 보내기 위한 것이다.

딜리버리 채널의 에러 응답은 업무적 관점의 비정상 응답과는 다르다. 업무적 관점의 비정상 응답은 딜리버리 채널에서는 정상 응답과 마찬가지로 응답 메시지를 통해 전달되어야 한다. 즉, 각 서비스 컴포넌트들은 비정상 응답

메시지를 보낼 때에는 딜리버리 채널에서 정상 응답 메시지를 보내는 방식과 동일한 API를 사용하고, 에러를 보낼 때에는 에러를 전달하는 별도의 API를 사용한다.

## 2.3.2. 서비스 플로우

AnyLink 서비스 플로우 정의는 내부적으로 에러 이벤트와 에러 핸들러, 에러 코드 매퍼를 정의하고 있다 또, 서비스 요청에 대한 비정상 응답을 처리하는 메커니즘도 정의하고 있다.

- 에러 이벤트

AnyLink 서비스 플로는 흐름 제어를 위해 에러 이벤트를 발생시키거나(끝 이벤트로 에러 이벤트를 사용할 경우), 감지할(바운더리 이벤트로 에러 이벤트를 사용할 경우) 수 있다.

에러 이벤트를 던지거나 받을 때에는 에러 코드를 사용한다. 서비스 플로우 인스턴스를 실행하는 도중 내부에서 예기치 않은 오류가 발생할 경우(Java 언어의 Exception이 발생한 경우)에는 이를 에러 코드로 변환하는 에러 코드 매퍼를 사용할 수가 있다. 즉, 에러 코드 매퍼는 Java 언어의 Exception이 발생했을 때, 이것을 서비스 플로우가 알 수 있는 에러 코드로 변환해 주는 인터페이스이다.

기본적으로는 에러 코드 매퍼가 정의되어 있지 않는데, 에러 코드로 변환되지 않은 Exception은 해당 Exception 객체의 전체 클래스 이름이 에러 코드로 사용된다. 바운더리 이벤트로 에러를 잡을 때 에러 코드를 지정할 수 있는데, ALL을 선택하면 Exception 종류나 에러 코드 값에 관계없이 모든 에러를 잡을 수 있다.

- 에러 핸들러

서비스 플로우별로 지정할 수 있는 프로세스 에러 핸들러와 액티비티별로 지정할 수 있는 액티비티 에러 핸들러가 있다. 서비스 플로우 인스턴스 실행 도중 Exception이 발생했을 때 액티비티 혹은 프로세스 에러 핸들러가 지정되어 있으면 이 에러를 처리하여 서비스 플로우를 정상적으로 진행시킬 수 있다.

- 서비스 액티비티의 비정상 응답

어댑터 아웃바운드 룰이나 멀티바인딩 라우터 룰과 같은 서비스를 호출하였을 때 비정상 응답을 받을 수 있다. 비정상 응답의 정의는 비정상 응답을 보내는 쪽에서 문맥적으로 정의된다. 비정상 응답이 발생할 수 있는 경우에는 비정상 응답을 위한 매핑을 별도로 지정할 수 있다.

서비스가 비정상 응답을 보내 올 경우에 액티비티에서 비정상 응답 매핑이나 비정상 응답 파라미터를 지정하지 않으면, 응답 매핑이나 응답 파라미터를 대신 사용하는 것이 아니라 비정상 응답 전문을 파라미터에 저장하지 않게 되므로 주의해야 한다.

AnyLink 서비스 플로우에서 비정상 응답은 에러 이벤트와는 상관이 없다. 즉, 비정상 응답은 서비스 플로우의 에러를 처리하는 흐름 제어 방식과는 관련이 없다.

## 2.3.3. 어댑터

어댑터는 파싱 룰에서 응답 메시지 외에 비정상 응답 메시지를 정의하고 있으며, 어댑터의 엔드포인트에서 내부 시스템 에러가 발생했을 때 시스템 에러 메시지를 정의할 수 있도록 하고 있다.

어댑터는 파싱 룰과 아웃바운드 룰에서 각각 요청 메시지와 응답 메시지, 비정상 응답 메시지를 정의하게 되어 있다. 또 시스템 에러, 포맷 에러 등 다양한 에러 시나리오에 대한 대응을 지정할 수 있도록 룰이 설계되어 있다. 어댑터의

에러 처리에 대한 상세 내용은 해당 어댑터 안내서를 참고할 수 있다.

## 3. 런타임 엔진 서버 리소스

본 장에서는 AnyLink 런타임 엔진 서버의 실행에 필요한 리소스를 정의하고 관리하는 방법에 대해서 설명한다.

### 3.1. 개요

AnyLink 런타임 엔진 서버는 실행에 필요한 몇 가지 리소스들을 정의하고 관리한다.

- 업무 혹은 거래에 관련된 비즈니스 리소스
- 어댑터와 어댑터 엔드포인트에 관련된 어댑터 리소스
- 스레드 풀(Thread Pool)과 시스템 로깅 등에 관련된 시스템 설정 정보

### 3.2. 비즈니스 리소스

비즈니스 리소스는 업무 혹은 거래에 관련된 리소스로 스튜디오에서 정의하는 거래와 거래그룹에 포함된 리소스들로 구성된다. 업무 개념을 구성하는 요소들이며, 여러 개의 계층으로 구성된 거래그룹과 말단 노드인 거래로 분류된다.

런타임 엔진 서버에서 비즈니스 리소스들은 기본적으로 각 서버 홈 디렉터리 아래에 위치한다.

```
${server.home}/repository/services
```

#### 3.2.1. 거래 및 거래그룹

거래와 거래그룹에는 다음과 같은 비즈니스 리소스들이 존재할 수 있다. 이 리소스들은 런타임 엔진 서버 내부적으로 거래를 나타내는 특정 디렉터리 아래에 하나의 압축 파일(.iar 파일을 확장자로 가지는 zip 형태의 파일)로 만들어져 있다.

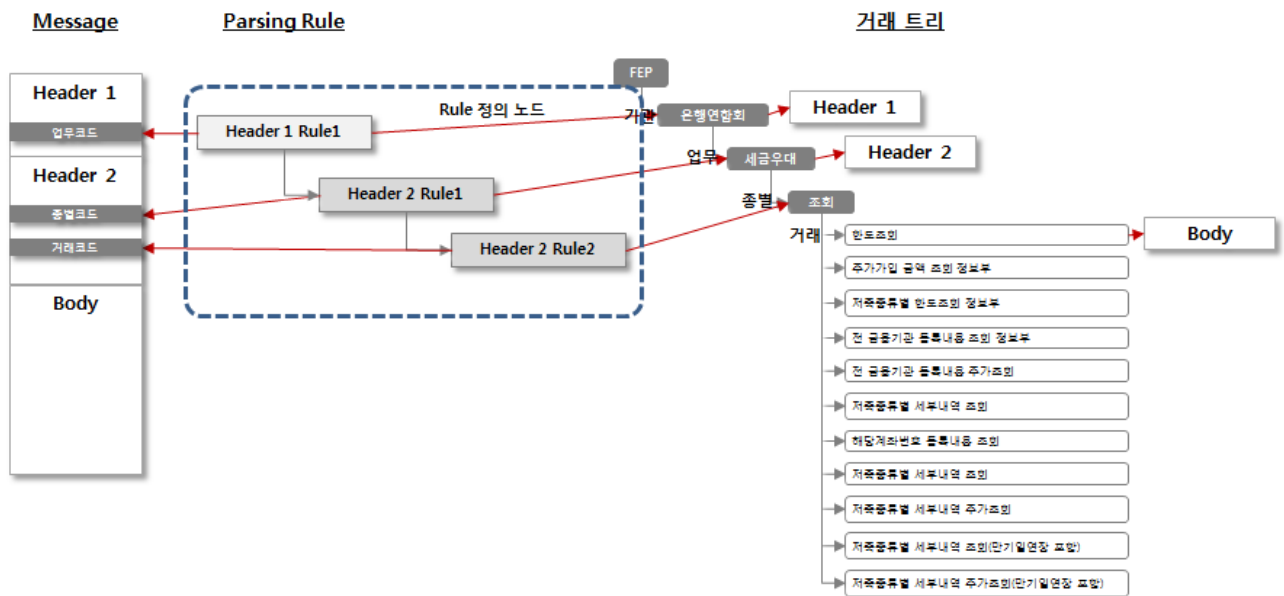
- **거래 및 거래그룹에 대한 정의 파일(.biztx 확장자를 가지는 XML 문서 파일)**

거래 및 거래그룹은 계층 구조(트리 구조)를 가지며 요청 메시지를 파싱하여 거래를 찾을 수 있는 구조로 구성되어 있다.

거래그룹은 하위 거래그룹과 거래들을 포함하는 부모 디렉터리로 표현되며 거래그룹에 관련된 비즈니스 리소스를 거래와 마찬가지로 .iar 아카이브 파일로 가지고 있다. 거래 및 거래그룹 정보는 .iar 아카이브 파일 안에 .biztx 확장자를 가진 XML 문서로 존재한다.

거래는 요청 메시지, 응답 메시지, 비정상 응답 메시지 등으로 구성되며, 계층 구조 상 부모 노드에 있는 거래 그룹의 정보를 상속받는 방식으로 구성된다.

거래 트리의 부모 노드 혹은 중간 노드에 위치하는 거래그룹은 요청 메시지를 파싱하여 하위 거래그룹이나 단말인 거래 노드를 찾기 위해 하위 거래 식별자를 가질 수 있다.



## 거래 트리와 메시지 파싱

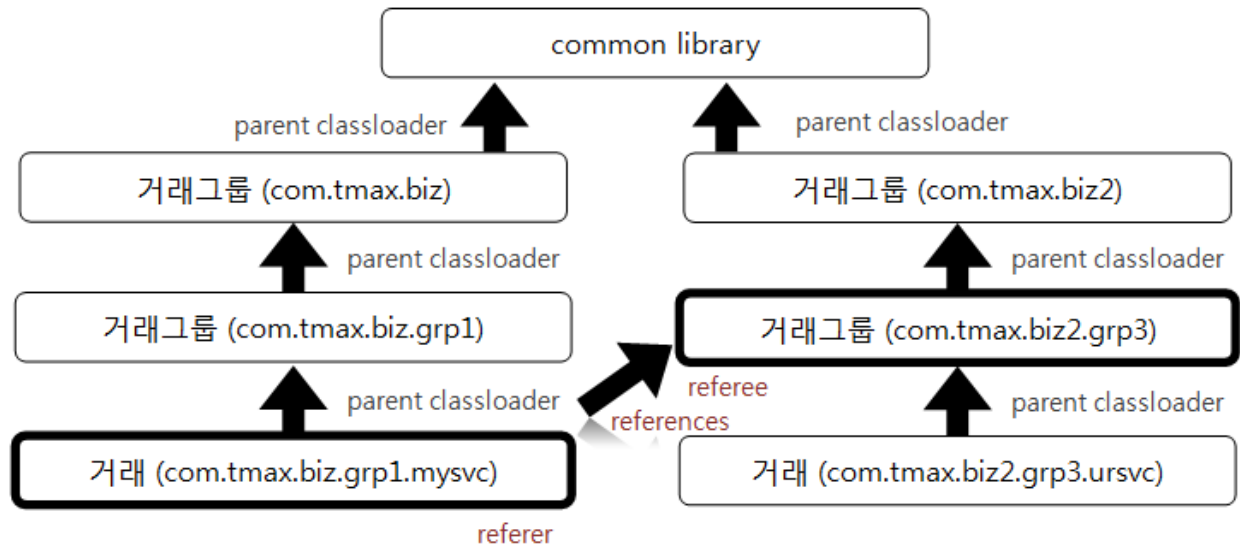
거래 트리의 기본 목적은 요청 메시지를 파싱하여 메시지 내용에 맞는 거래를 찾고 결과적으로 AnyLink 엔진 내부 서비스(서비스 플로우, 아웃바운드 룰 혹은 멀티바인딩 룰)를 호출하는 것이다.

거래, 거래그룹 내용 중엔 해당 거래 혹은 거래그룹 노드가 참고하는 심볼릭 링크(Symbolic Link) 정보가 있다. 심볼릭 링크는 해당 거래 노드가 참고하는 리소스들이 있는 다른 거래 노드들을 링크하는 역할을 한다. 기본적으로 거래 트리 상의 부모 노드의 리소스들은 참조할 수 있지만, 부모가 아닌 다른 거래 혹은 거래그룹 노드의 리소스를 공유해야 할 때 심볼릭 링크를 유용하게 사용할 수 있다.

AnyLink의 거래 혹은 거래그룹 노드에서 함께 관리하는 리소스에는 Java 코드로 변환되는 여러 가지 리소스가 있다. 예를 들면 메시지, 매핑, 표현식, 다양한 사용자 클래스 등이 Java 코드로 사용되는 리소스들이다.

Java 코드들은 내부적으로 클래스 로더 개념을 통해 코드를 읽어들이게 되는데, AnyLink의 각 거래 트리는 클래스 로더 상의 부모, 자식 관계를 가지도록 설계되어 있다.

심볼릭 링크 노드는 클래스 로더 상의 부모가 아니지만, 마치 부모인 것처럼 사용할 수 있도록 AnyLink의 클래스 로더 구현체가 지원한다.



심볼릭 링크와 Java 클래스 참조

#### • 서비스 플로우 정의 파일(.sfdl 확장자를 가지는 XML 문서 파일)

서비스 플로우 정의 파일은 AnyLink 스튜디오의 서비스 플로우 에디터에서 BPMN(Business Process Modeling Notation) 다이어그램을 사용하여 그린 서비스 플로우의 정의 문서이다. 서비스 플로는 서비스 플로우 엔진에서 관리하며 서비스 플로우의 수신(Receive) 메시지 이벤트들이 AnyLink의 내부 서비스로 등록된다.

가장 일반적인 AnyLink 거래 형태는 주로 인바운드 어댑터를 통해 메시지가 들어오며, 들어온 메시지는 거래 트리를 통해 파싱되어 내부 서비스를 식별하게 된다. 대부분 AnyLink 내부 서비스는 서비스 플로우의 메시지 이벤트이며 플로우에 정의된 흐름을 따라 진행 후 플로우의 서비스 액티비티를 호출하게 된다. 플로우에 그려진 서비스 액티비티는 대부분 어댑터의 아웃바운드 룰 서비스를 의미한다. 아웃바운드 룰 서비스를 통해 외부 시스템을 호출한 후 응답을 받으면 호출한 플로우로 넘겨줘서 최초 메시지를 보낸 어댑터로 응답 메시지를 보내는 것이 가장 기본적인 형태의 AnyLink 거래이다.

#### • 파싱 룰

AnyLink에서 외부로 부터 들어오는 메시지를 파싱하는 규칙인 파싱 룰은 별도의 리소스로 관리되지 않고, 거래 및 거래그룹에서 정의한다.

#### • 어댑터 아웃바운드 룰(.orule 확장자를 가지는 XML 문서 파일)

AnyLink에서 다른 시스템으로 요청을 내보내는 아웃바운드 어댑터 룰을 정의하는 문서. 각 어댑터의 프로토콜과 구현방식에 따라 설정 정보들이 달라진다. 기본 설정 정보로는 요청 메시지, 응답 메시지, 비정상 응답 메시지와 어댑터 및 엔드포인트 ID 정보가 있다.

아웃바운드 룰은 AnyLink 내부 서비스의 하나이며 내부적으로 다른 컴포넌트에서 아웃바운드 룰 서비스를 호출하면 해당 룰에 설정된 어댑터와 엔드포인트를 식별하여 외부 시스템으로 메시지를 전송하는 방식으로 동작한다.

#### • 멀티바인딩 룰(.mbind 확장자를 가지는 XML 문서 파일)

AnyLink 내부 서비스를 경우에 따라 다르게 라우팅을 해야 할 경우 여러 서비스를 묶어서 하나의 그룹으로 보게 해주는 서비스가 멀티바인딩 라우터 룰이다. 멀티바인딩 라우터 룰은 설정에 따라 메시지의 특정 값에 따른

서비스 분기, 라운드 로빈(Round Robin)이나 가중치 기반 라운드 로빈, 시간대별, 서비스 플로우의 코릴레이션, 핸들러 등을 사용한 서비스 분기를 지시할 수 있다.

요청, 응답, 비정상 응답 메시지와 필요 시 각 메시지에 대한 매핑 즉, 메시지 변환을 지정할 수 있다.

#### • 메시지(.umsg 확장자를 가지는 XML 문서 파일)

통합 메시지 문서인 .umsg 파일 내부에는 여러 개의 메시지가 존재할 수 있다. AnyLink는 내부적으로 입출력에 사용되는 데이터의 내부 구조를 표현하는 형태인 DTO(Data Transfer Object)와 특정 서식에 따른 메시지를 구분하는데 .umsg 파일에는 같은 데이터 구조를 가진 하나의 DTO와 여러 서식의 메시지가 함께 존재할 수 있다.

지원하는 메시지 서식으로는 XML, 고정 길이 형태(Fixed Length), 구분 문자 형태(Delimiter), NameValue, JSON 등이 있다.

AnyLink는 내부적으로 성능 및 효율성을 고려하여 DTO와 메시지를 Java 코드로 변환하여 관리한다. 즉, .umsg 문서에 해당하는 .java 소스 파일들을 생성한 후 컴파일한 .class 파일들이 거래 리소스에 포함되어 있다.

#### • 메시지 매핑(.map 확장자를 가지는 XML 문서 파일)

메시지 매핑 문서는 입력 메시지들을 출력 메시지들로 변환하는 규칙을 정의하는 문서이다. AnyLink는 각 입력 메시지와 출력 메시지의 필드들을 연결하는 형태로 변환 규칙을 지정하며, 입력 메시지 외에도 몇 가지 표현식을 매핑의 소스로 사용할 수 있다.

입출력 필드간의 조합 혹은 표현식으로 구성된 매핑의 실행 속도를 개선하기 위하여 AnyLink는 내부적으로 매핑을 Java 코드로 변환하여 관리한다. 즉 메시지 파일과 마찬가지로 .java 소스 파일을 생성한 후 컴파일한 .class 파일이 거래 리소스에 포함된다.

#### • 표현식(.expr 확장자를 가지는 XML 문서 파일)

메시지의 일부를 나타내거나 함수식을 표현하기 위하여 AnyLink는 표현식(Expression)을 사용한다.

AnyLink 표현식은 메시지의 구조를 표현할 수 있는 형태이다. 예를 들어 msg1에 field1이라는 필드가 존재한다고 하면 "msg1.field1"이라는 표현식으로 표현한다. 서비스 플로우에서는 플로우에서 선언된 변수인 DataField에 접근할 수 있는데 예를 들어 dataField1이라는 변수의 name이라는 필드에 접근하려면 "\$dataField1.name"이라는 표현식으로 표현한다.

AnyLink 표현식에서는 그룹을 표현하는 괄호를 사용할 수 있으며 Java 메소드들을 호출할 수 있다. 예를 들어 위의 변수 예에서 name이라는 필드가 Java의 String 클래스에 해당한다면 다음과 같이 앞의 두 글자만 추출할 수 있다.

```
"$dataField1.name.substring(0, 2)"
```

필요에 따라 변수의 Java 자료형을 명시적으로 지정할 수도 있다. 예를 들어 dataField1이라는 변수가 com.example.NamedPerson이라는 자료형을 가진다면 다음과 같이 위의 식을 고쳐 쓸 수 있다.

```
"$dataField1<com.example.NamedPerson>.name.substring(0,2)"
```

AnyLink는 표현식의 실행 속도를 고려하여 Java 코드로 변환하여 관리한다. 잘못된 표현식이라면 스튜디오에서 배포할 때 Java 코드 변환 및 컴파일할 때에 오류가 발생하게 된다. .expr 파일도 메시지, 매핑 파일과 마찬가지로 컴파일된 .class 파일이 거래 리소스에 포함되어 관리된다.

AnyLink에서 표현식은 서비스 플로우의 분기 조건문을 나타내거나 코릴레이션 계산식, 멀티바인딩 라우터의 룰 등에서 사용된다.

메시지 매핑의 소스 정보를 표현할 때 사용할 수 있는 표현식은 매핑의 특성을 지원하기 위해 즉, 비어있는 소스의 객체 레퍼런스가 있으면 빈 객체를 만들어준다거나, 배열 형태의 매핑을 지원하는 등의 특성을 지원하기 위해 조금 다른 형태를 가진다.

### 3.2.2. 거래 및 거래그룹 환경설정

다음은 거래 및 거래그룹에 대한 환경설정 파일에 대한 설명이다.

- **거래 및 거래그룹에 대한 환경설정(.bizcfg 확장자를 가지는 XML 문서 파일)**

거래 및 거래그룹의 환경설정 문서는 직접적으로 거래 정보와는 상관이 없으나 거래의 환경에 해당하는 정보들을 저장하는 문서이다. 거래 및 거래그룹 리소스들은 각 노드별로 하나의 .iar 파일에 관리되지만 거래 노드의 환경설정 문서 파일은 해당 거래 혹은 거래그룹 노드를 나타내는 디렉터리 아래에 .bizcfg 확장자를 가지는 문서 형태로 별도로 존재한다.

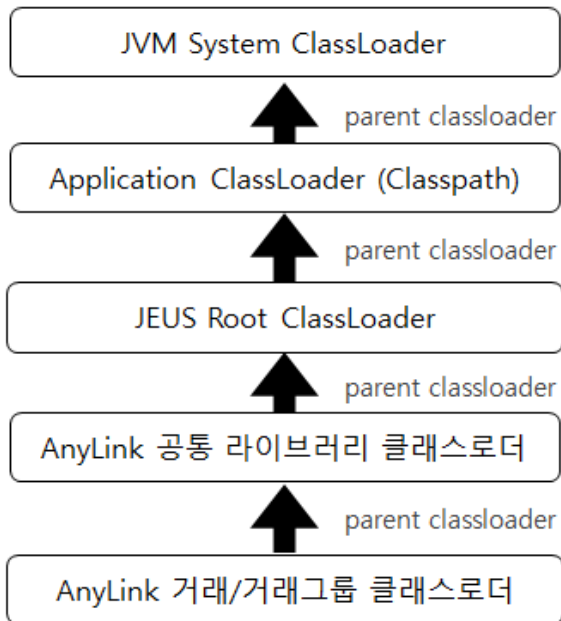
환경설정 문서에는 거래 트레이스 로그 레벨과 아웃바운드 룰별 트레이스 로그 레벨을 설정할 수 있다. 그외 해당 거래 노드의 서비스 플로우 등이 사용할 스레드 풀을 지정할 수 있다.

### 3.2.3. Java 클래스 로더와 거래 트리 클래스 로딩 방식

거래 및 거래그룹을 나타내는 비즈니스 리소스에는 사용자 클래스나 핸들러 등의 Java 코드 외에도 자동으로 생성되는 메시지, 매핑, 표현식 등의 Java 코드들이 포함되어 있다. 앞에서 심볼릭 링크를 설명하면서 언급하였듯이 각 Java 코드들은 Java의 특성에 따라 클래스 로더 구성에 의한 코드 참조 구조를 따르게 된다.

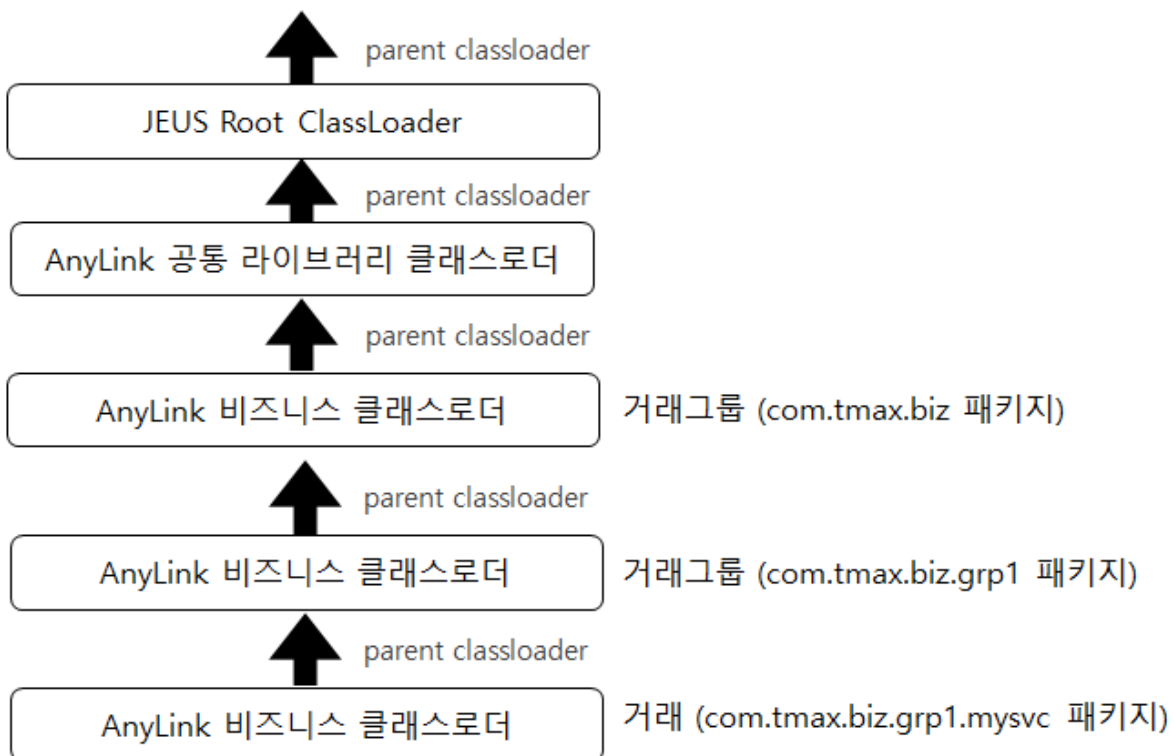
AnyLink는 Java EE 환경에서 실행되므로 기본적인 클래스 로더는 다음과 같은 구조를 가진다.





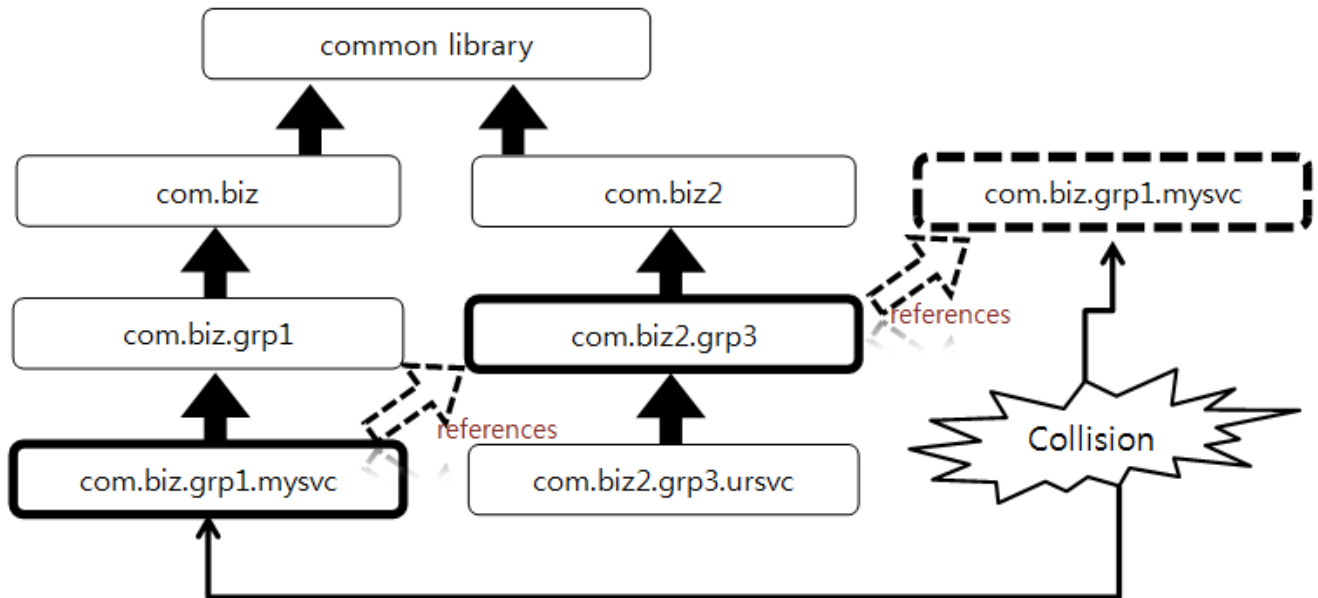
Java 클래스 로더 기본 구성 계층도

AnyLink 비즈니스 클래스 로더는 하나의 클래스 로더가 아니며 루트 클래스 로더 아래에 거래 트리 형태의 클래스 로더를 자손으로 가지는 형태가 된다.



AnyLink 비즈니스 클래스 로더 예를 포함한 클래스 로더 계층도

클래스 로더 계층 구조 특성상 클래스 로더 여러 계층에 걸친 순환 참조를 허용하지 않는다. 이것은 심볼릭 링크를 사용할 때에도 마찬가지이다. 심볼릭 링크는 일종의 부모 클래스 로더와 같은 관계를 만들게 된다.



심볼릭 링크 순환 참조에 의한 오류

특정 거래 혹은 거래그룹 노드의 내용을 변경하여 재배포하게 되면 해당 거래 노드와 모든 자손 노드들이 재적재된다. 이때 재적재되는 자손 노드들은 심볼릭 링크를 통해 참조하는 논리적 자손 노드들을 포함한다.

### 3.3. 라이브러리

AnyLink 거래 트리에서 공통으로 사용하는 Java 라이브러리가 있을 경우 이 라이브러리는 거래 트리과 별도로 라이브러리 디렉터리에서 따로 관리한다.

AnyLink 비즈니스 클래스 로더 계층 구조 상 라이브러리는 최상위 비즈니스 클래스 로더에서 읽어들이게 된다. 운영할 때에 라이브러리를 배포할 경우 클래스 로더 최상위 노드에서 라이브러리 코드를 적재하는 특성 상, 자손 노드들인 전체 거래 트리의 클래스 로더를 다시 적재하게 된다.

런타임 엔진 서버에서 라이브러리 파일들은 기본값으로 각 서버 홈 디렉터리의 아래에 위치한다.

```
${server.home}/repository/lib
```

### 3.4. 어댑터 리소스

AnyLink 런타임 엔진에서는 다양한 프로토콜 및 애플리케이션과 통신을 담당하는 어댑터에 필요한 설정 정보와 어댑터 통신 엔드포인트 등의 리소스를 관리한다. 어댑터 관련 리소스는 크게 어댑터 설정 정보와 엔드포인트 및 엔드포인트 그룹으로 구분할 수 있다.

런타임 엔진 서버에서 어댑터 리소스들은 기본값으로 각 서버 홈 디렉터리의 아래에 위치한다.

```
${server.home}/repository/adapters
```

다음은 어댑터 리소스의 종류이다.

#### • 어댑터

어댑터 설정 정보 파일은 .adt 확장자를 가지는 XML 문서로 프로토콜 정보, 사용하는 스레드 풀 ID, 트레이스 로그 레벨, 클러스터링 정보 등을 설정한다.



AnyLink에서는 각 업무별로 트레이스 로그와 트랜잭션 로그를 설정하고 남길 수 있다. 이 로그들은 내부적으로는 로그 어댑터에서 처리한다. 이 기능에 대한 자세한 설명은 어댑터 안내서를 참고한다.

#### • 엔드포인트 및 엔드포인트 그룹

엔드포인트 정보 파일은 .ep 확장자를 가지는 XML 문서로 통신 방향, 동기 모드, 거래 노드 ID, 연결 타임아웃, 스레드 풀 ID, 트레이스 로그 레벨, 아웃바운드 엔드포인트 로그 레벨, 클러스터링 정보, 시스템 오류 메시지 등의 정보를 설정한다.

엔드포인트 그룹은 여러 개의 엔드포인트를 그룹화하여 처리하는 개념을 가지고 있으며 .epg 확장자를 가지는 XML 문서에 설정된다. 하위에 속한 엔드포인트와 엔드포인트 그룹으로 라우팅을 분기하기 위한 방법을 설정한다. 예를 들어 라운드 로빈, 최소 요청, 핸들러, 연결 전담형 라운드 로빈 등의 라우팅 방법을 지원한다. 엔드포인트 그룹은 하위 엔드포인트 그룹 혹은 엔드포인트를 가지는 부모 디렉터리로 표현된다.

## 3.5. 업무시스템 정보 설정

업무시스템은 개별 런타임 엔진 서버 또는 런타임 엔진 서버의 클러스터 단위로 구성된다.

AnyLink의 기반이 되는 Java EE 서버의 클러스터에 속한 서버들은 같은 업무시스템에 속하게 된다. AnyLink에서는 업무시스템별로 거래 트리 정보 등을 공유하므로 AnyLink에서 실행되는 애플리케이션 개념의 기본 배포 단위는 업무시스템 단위라고 생각할 수 있다. 즉, 업무시스템이 다르면 배포되는 거래 등 업무들도 다르게 구성된다.

AnyLink 런타임 엔진의 시스템 관련 설정은 업무시스템 공통과 개별 런타임 엔진 서버별 설정이다. 주로 스레드 풀과 로깅, Java EE 서버 도메인 사용자 ID와 암호 등의 정보를 설정하고 그외에 특별한 설정이 없을 경우 시스템에서 기본적으로 사용할 스레드 풀 ID와 기본 서비스 타임아웃 값 등을 설정한다.

해당 파일에 스레드 풀과 신뢰 전송 큐 정보, 시스템의 기본 트레이스 로그 수준을 설정한다. 이외에도 설정 파일에는 내부적으로 런타임 엔진 서버에서 필요한 몇 가지 추가 설정들을 가지고 있다. 본 절에서는 주요 설정 내역에 대해서 설명한다. 각 세부 내역에 대한 설명은 해당 절의 설명을 참고한다.

업무시스템 설정 정보는 각 서버별로 다음의 경로에 저장된다.

```
${server.home}/repository/bizsystem/bizsystem.config
```

<bizsystem.config>

```
<?xml version="1.0" encoding="UTF-8"?>
<ns0:bizSystemInfo xmlns:ns2="http://www.tmaxsoft.com/schemas/AnyLink/reliableQueue/"
```

```

xmlns:ns1="http://www.tmaxsoft.com/schemas/AnyLink/common/"
xmlns:ns3="http://www.tmaxsoft.com/schemas/AnyLink/serverPlugin/"
xmlns:ns0="http://www.tmaxsoft.com/schemas/AnyLink/">
<ns0:sysId>IFL_SYS</ns0:sysId>
  <ns0:id>IFL_SYS</ns0:id>
  <ns0:name>IFL_SYS</ns0:name>
  <ns0:version>4</ns0:version>
  <ns0:registererDate>2014-10-13 12:54:03.514</ns0:registererDate>
  <ns0:registeringUser>admin</ns0:registeringUser>
  <ns0:modificationDate>2014-11-10 20:20:09.829</ns0:modificationDate>
  <ns0:user>administrator</ns0:user>
  <ns0:passwd>ba05778dea933f47bc150c743b122336</ns0:passwd>
  <ns0:bizSystemDefaultThreadPoolId>bizSystemDefaultThreadPool
</ns0:bizSystemDefaultThreadPoolId>
  <ns0:threadPool>
    <ns0:id>bizSystemDefaultThreadPool</ns0:id>
    <ns0:name>bizSystemDefaultThreadPool</ns0:name>
    <ns0:queueFullPolicy>SystemError</ns0:queueFullPolicy>
    <ns0:useVipThread>false</ns0:useVipThread>
    <ns0:threadPriority>-1</ns0:threadPriority>
    <ns0:vipThreadPriority>-1</ns0:vipThreadPriority>
    <ns0:commonSetting>
      <ns0:queueSize>-1</ns0:queueSize>
      <ns0:min>60</ns0:min>
      <ns0:max>100</ns0:max>
      <ns0:keepAliveTime>300</ns0:keepAliveTime>
      <ns0:vipThreadCount>10</ns0:vipThreadCount>
    </ns0:commonSetting>
  </ns0:threadPool>
  <ns0:systemLogging>
    <ns0:fileLocation>${server.home}/logs/AnyLink_rte_%d{1}.log</ns0:fileLocation>
    <ns0:logLevel>FINEST</ns0:logLevel>
    <ns0:logger>
      <ns0:name>com.tmax.anylink.dis.level</ns0:name>
      <ns0:logLevel>INFO</ns0:logLevel>
    </ns0:logger>
  </ns0:systemLogging>
  <ns0:nodeList>
    <ns0:node>
      <ns0:name>server1</ns0:name>
    </ns0:node>
  </ns0:nodeList>
  <ns0:clusterGroup>
    <ns0:adapterClusterGroup>
      <ns0:adapterCluster>
        <ns0:clusterId>TCP_ADT</ns0:clusterId>
        <ns0:activeCount>2</ns0:activeCount>
        <ns0:construction>
          <ns0:serverId>server4</ns0:serverId>
          <ns0:priority>0</ns0:priority>
          <ns0:clusterMember>true</ns0:clusterMember>
        </ns0:construction>
        <ns0:construction>
          <ns0:serverId>server5</ns0:serverId>
          <ns0:priority>1</ns0:priority>
          <ns0:clusterMember>true</ns0:clusterMember>
        </ns0:construction>
        <ns0:construction>
          <ns0:serverId>server6</ns0:serverId>

```

```

        <ns0:priority>2</ns0:priority>
        <ns0:clusterMember>false</ns0:clusterMember>
    </ns0:construction>
</ns0:adapterCluster>
</ns0:adapterClusterGroup>
</ns0:clusterGroup>
<ns0:deploymentPolicy>allowPartialDeploy</ns0:deploymentPolicy>
<ns0:encryptAlgorithm>AES128</ns0:encryptAlgorithm>
<ns0:debuggingMode>false</ns0:debuggingMode>
</ns0:bizSystemInfo>

```

### 3.5.1. 스레드 풀

런타임 엔진에서 사용할 수 있는 스레드 풀들을 지정한다. 스레드 풀 세부 설정 정보들은 업무시스템별로 공통이거나 개별 런타임 엔진 서버별로 일부 다르게 설정할 수 있다.

다음은 설정 예와 각 항목에 대한 설명이다.

```

<ns0:threadPool>
  <ns0:id>bizSystemDefaultThreadPool</ns0:id>
  <ns0:name>bizSystemDefaultThreadPool</ns0:name>
  <ns0:queueFullPolicy>SystemError</ns0:queueFullPolicy>
  <ns0:useVipThread>false</ns0:useVipThread>
  <ns0:threadPriority>-1</ns0:threadPriority>
  <ns0:vipThreadPriority>-1</ns0:vipThreadPriority>
  <ns0:commonSetting>
    <ns0:queueSize>-1</ns0:queueSize>
    <ns0:min>60</ns0:min>
    <ns0:max>100</ns0:max>
    <ns0:keepAliveTime>300</ns0:keepAliveTime>
    <ns0:vipThreadCount>10</ns0:vipThreadCount>
  </ns0:commonSetting>
</ns0:threadPool>

```

속성	설명
id	스레드 풀 아이디이다.
name	스레드 풀 이름이다.
queueFullPolicy	Queue가 Full이 되는 경우 정책을 설정한다. <ul style="list-style-type: none"> <li>◦ SystemError : 시스템 오류 처리</li> <li>◦ CallerThread : 호출자 스레드</li> </ul>
useVipThread	일정 개수의 VIP 스레드를 둘 것인지 여부를 설정한다.
threadPriority	일반 스레드의 우선순위를 설정한다. (기본값: 6)
vipThreadPriority	VIP 스레드의 우선순위를 설정한다. (기본값: 3)
queueSize	스레드 풀에서 대기하는 작업 큐의 최대 크기를 설정한다. (서버별 설정 가능)
max	일반 스레드 풀의 최대 크기이다. (서버별 설정 가능)

속성	설명
min	일반 스레드 풀의 최소 크기이다. (서버별 설정 가능)
keepAliveTime	작업이 할당되지 않았을 때 할당된 스레드를 취소시키지 않는 기본 유지 시간이다. (서버별 설정 가능)
vipThreadCount	VIP 스레드 개수(고정)이다. (서버별 설정 가능)

### 3.5.2. 시스템 로깅

업무시스템 설정 정보에는 시스템 로깅 정보를 지정할 수 있다. 시스템 로깅은 AnyLink 런타임 엔진 서버의 내부 로그를 나타내는 것으로 설정을 통해 출력 및 기본 로그 수준과 로거별 로그 수준을 지정할 수 있다. 로거별 로그 수준 지정은 서버별로 다르게 지정할 수도 있다.

로그 파일이 저장될 위치를 나타내는 fileLocation에는 다음과 같은 미리 정의된 변수 값을 사용할 수 있다.

```
<ns0:systemLogging>
  <ns0:fileLocation>${server.home}/logs/AnyLink_rte_%d{1}%h{1}%m{30}.log</ns0:fileLocation>
  <ns0:logLevel>FINEST</ns0:logLevel>
  <ns0:logger>
    <ns0:name>com.tmax.anylink.serviceflow.level</ns0:name>
    <ns0:logLevel>INFO</ns0:logLevel>
  </ns0:logger>
  <ns0:logger>
    <ns0:name>com.tmax.anylink.runtime.level</ns0:name>
    <ns0:logLevel>FINE</ns0:logLevel>
  </ns0:logger>
</ns0:systemLogging>
```

속성	설명
server.home	AnyLink 런타임 엔진 서버별 홈 디렉터리 위치를 나타낸다.
domain.home	AnyLink 도메인별 홈 디렉터리 위치를 나타낸다.
install.root	AnyLink 바이너리가 설치된 루트 디렉터리 위치를 나타낸다.
server.name	현재 AnyLink 서버명을 나타낸다.
adminServer.name	AnyLink의 데이터 통합 서버명을 나타낸다.

### 3.5.3. 클러스터 설정

클러스터를 사용하는 업무시스템의 어댑터별 클러스터 설정을 나타낸다. 아무런 설정을 하지 않을 경우 모든 어댑터가 활성화되어 동작한다.

다음은 설정 예와 각 항목에 대한 설명이다.

```
<ns0:clusterGroup>
  <ns0:adapterClusterGroup>
    <ns0:adapterCluster>
```

```

<ns0:clusterId>TCP_ADT</ns0:clusterId>
<ns0:activeCount>2</ns0:activeCount>
<ns0:construction>
  <ns0:serverId>server4</ns0:serverId>
  <ns0:priority>0</ns0:priority>
  <ns0:clusterMember>true</ns0:clusterMember>
</ns0:construction>
<ns0:construction>
  <ns0:serverId>server5</ns0:serverId>
  <ns0:priority>1</ns0:priority>
  <ns0:clusterMember>true</ns0:clusterMember>
</ns0:construction>
<ns0:construction>
  <ns0:serverId>server6</ns0:serverId>
  <ns0:priority>2</ns0:priority>
  <ns0:clusterMember>false</ns0:clusterMember>
</ns0:construction>
</ns0:adapterCluster>
</ns0:adapterClusterGroup>
</ns0:clusterGroup>

```

속성	설명
clusterId	클러스터 아이디이다.
activeCount	최대 활성 노드 수이다.
construction	<p>서버 노드별 클러스터 우선순위 및 멤버를 설정한다.</p> <ul style="list-style-type: none"> <li>◦ serverId : 서버 아이디를 설정한다.</li> <li>◦ priority : 서버 우선순위를 설정한다.</li> <li>◦ clusterMember : 클러스터에 포함될지 여부를 설정한다.</li> </ul>

### 3.5.4. 배포 정책 설정

기본적으로 리소스 배포는 모든 서버에 배포가 성공하지 못할 경우 실패 처리를 하게 된다. 하지만 클러스터 환경에서 일부 서버의 동작에 이상이 있을 경우, 부분적인 서버에 대한 배포에 대한 허용을 옵션으로 선택하여 동작한다.

다음은 설정 예와 각 항목에 대한 설명이다.

```

<ns0:deploymentPolicy>allowPartialDeploy</ns0:deploymentPolicy>

```

속성	설명
deploymentPolicy	<p>배포 정책을 설정한다.</p> <ul style="list-style-type: none"> <li>◦ allOrNothing : 전부 성공하지 못하면 실패 처리한다.</li> <li>◦ allowPartialDeploy : 살아있는 서버에 대해서만 배포를 허용하고, 배포가 되지 않은 서버가 기동될 때에 자동 동기화한다.</li> </ul>

### 3.5.5. 암호화 알고리즘 설정

인피니링크 내부에서 사용하는 공용 암호화 알고리즘을 설정한다.

다음은 설정 예와 각 항목에 대한 설명이다.

```
<ns0:encryptAlgorithm>AES128</ns0:encryptAlgorithm>
```

속성	설명
encryptAlgorithm	암호화 알고리즘을 설정한다. <ul style="list-style-type: none"><li>◦ AES128 : AES128을 사용하여 암호화한다. 별도 설정이 필요하지 않다.</li><li>◦ AES256 : AES256을 사용하여 암호화한다. JRE에서 해당 암호화 알고리즘을 사용할 수 있도록 설정해야 한다.</li></ul>

### 3.5.6. 디버깅 모드 설정

해당 업무시스템에서 플로우 디버깅 사용을 허용할 것인지에 대한 설정이다.

다음은 설정 예와 각 항목에 대한 설명이다.

```
<ns0:debuggingMode>false</ns0:debuggingMode>
```

속성	설명
debuggingMode	디버깅 모드를 설정한다. <ul style="list-style-type: none"><li>◦ true : 디버깅 모드를 허용한다. 스튜디오에서 기 배포된 리소스에 대한 플로우 디버깅을 사용할 수 있다.</li><li>◦ false : 디버깅 모드를 허용하지 않는다. 플로우 디버깅을 사용할 수 없다.</li></ul>



## 4. 서비스 플로우 엔진

본 장에서는 서비스 플로우 엔진으로 작성하는 다이어그램의 기본 요소와 패턴에 대해서 설명한다.

### 4.1. 서비스 플로우 다이어그램

AnyLink는 다양한 어댑터와 서비스를 연계하고 오케스트레이션할 수 있는 기능을 서비스 플로우 엔진을 통해 지원한다. AnyLink는 수신 서비스들과 송신 서비스들을 서비스 플로우를 통해 연결하여 추가적인 처리를 하거나 새로운 가치를 부여한다.

AnyLink 서비스 플로는 비즈니스 프로세스를 모델링하는 데 사용되는 표준인 BPMN(Business Process Modeling Notation)을 사용하여 그려지는 프로세스 정의를 뜻한다. BPMN 형태로 그려진 다이어그램은 내부적으로 XML 문서 형태인 SFDL(Service Flow Description Language) 서식으로 저장된다.

#### 4.1.1. 서비스 플로우 다이어그램 기본 요소

다음은 서비스 플로우 다이어그램 기본 요소에 대한 설명이다.

- 서비스 플로우(Service Flow)

프로세스(Process)라고도 부르며, 액티비티, 이벤트와 트랜지션으로 구성된다.

- 액티비티(Activity)

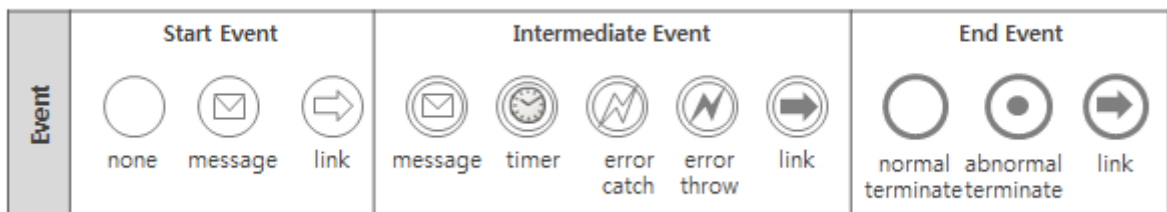
작업(Task)이라고도 부르며, 서비스 플로우에서 해당 위치로 흐름이 도달하게 되면 실행해야 할 어떤 일들을 표현한다.

- 트랜지션(Transition)

액티비티 또는 이벤트들 간의 흐름의 순서를 나타내는 화살표이다.

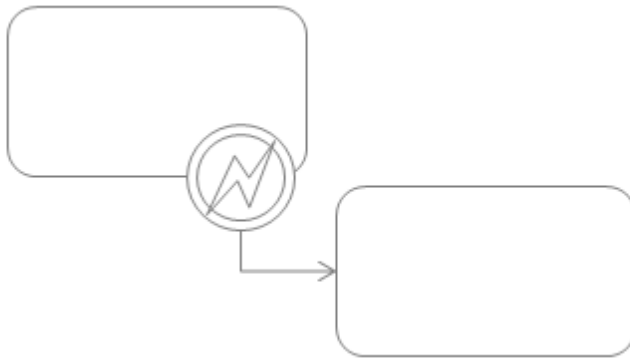
- 이벤트(Event)

사건을 나타내는 특별한 액티비티로 메시지, 에러, 타임아웃 등의 사건을 나타낸다. 이벤트는 전후 트랜지션 유무에 따라 들어오는 트랜지션이 없는 시작 이벤트(Start Event), 들어오는 트랜지션과 나가는 트랜지션이 모두 있는 중간 이벤트(Intermediate Event), 나가는 트랜지션이 없는 끝 이벤트(End Event)로 구분할 수 있다.



Start Event

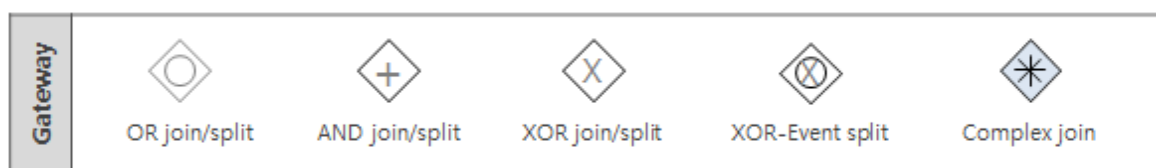
중간 이벤트의 특수한 형태인 바운더리 이벤트(Boundary Event)는 액티비티나 블록의 경계에 위치하여 표현하는 이벤트이며 해당 액티비티 혹은 블록이 활성화된 시점에 발생하는 이벤트가 있으면 바운더리 이벤트로 분기가 일어난다.



Boundary Event

- 게이트웨이(Gateway)

흐름을 제어하는 데 사용되는 특별한 액티비티 형태라고 생각할 수 있다. 크게 하나의 트랜지션 흐름을 여러 개의 트랜지션 흐름으로 분리시켜주는 스플릿 게이트웨이(Split Gateway)와 그 반대로 여러 개의 트랜지션 흐름을 하나의 트랜지션 흐름으로 합쳐주는 조인 게이트웨이(Join Gateway)가 있다.



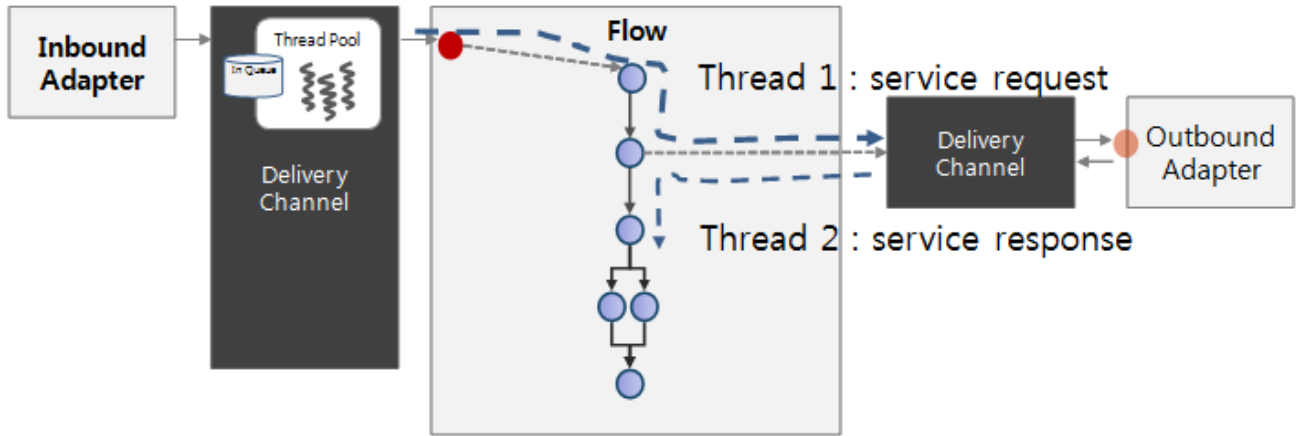
Split Gateway

## 4.1.2. 서비스 플로우 병렬 처리

AnyLink 서비스 플로는 런타임 엔진 서버의 서비스 플로우 엔진에서 실행이 된다. 보통 인바운드 어댑터를 통해서 전달된 메시지에 의해서 정의된 서비스 플로우의 인스턴스가 만들어지게 되는데 이렇게 서비스 플로우로 전달되는 메시지나 타이머 정보를 서비스 플로우에서는 트리거(trigger)라고 한다.

일반적으로 하나의 트리거에 의해서 발생하는 서비스 플로우의 흐름은 하나의 스레드에 의해 실행된다. 하지만, 서비스 플로우의 흐름을 정의하는 SFDL은 흐름의 병렬 분기나 병합이 가능하다.

AnyLink 서비스 플로우 엔진은 서비스 플로우 다이어그램의 형태에 따라 하나의 서비스 플로우 인스턴스를 동시에 다수의 스레드에서 실행하게 되는 경우도 있다. 또한 AnyLink 서비스 플로우 엔진은 서비스 플로우를 생성하고 흐름을 처리할 때 시스템 자원인 스레드의 활용을 극대화하고 다량의 동시 처리를 가능하게 설계되어 있다.



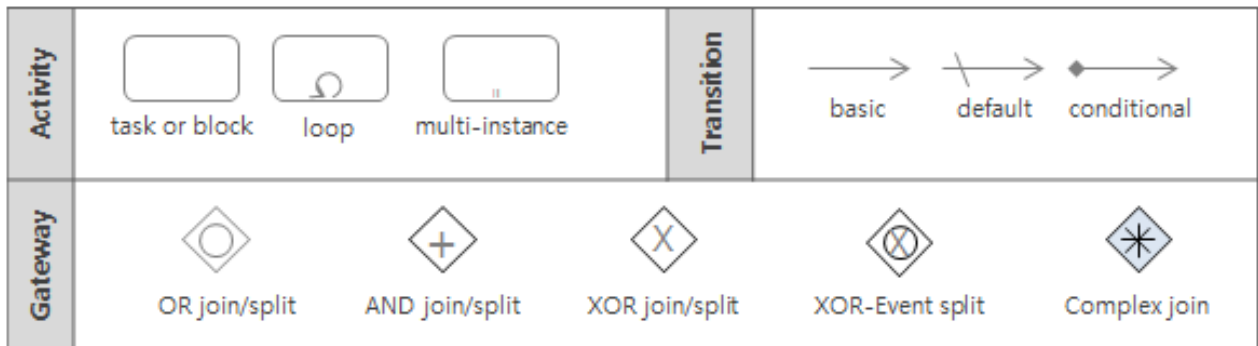
서비스 플로우 엔진의 응답 대기 없는 효율적인 스레드 처리 구조

서비스 호출 스레드(thread 1)는 요청을 보낸 후 응답을 기다리지 않는다. 서비스 응답 처리 스레드(thread 2)는 응답이 올 때 어댑터에서 만들어진다.

### 4.1.3. 다이어그램 그래프와 블록 구조

AnyLink 서비스 플로우의 흐름은 기본적으로는 액티비티들을 방향이 있는 화살표로 이어서 표현하는 그래프 구조 기반이다. 그래프 구조는 사람이 사고하는 대로 흐름을 표현할 수 있어 직관적이고 유연한 장점이 있다.

AnyLink 플로우 엔진의 각 액티비티(Activity)들과 트랜지션(Transition, 화살표로 표현됨)들 그리고 각종 흐름 분기와 흐름 병합을 표현하는 게이트웨이(Gateway)들은 이러한 그래프 구조의 서비스 플로우를 잘 표현해준다.



서비스 플로우의 그래프 구조 요소들 - Activity

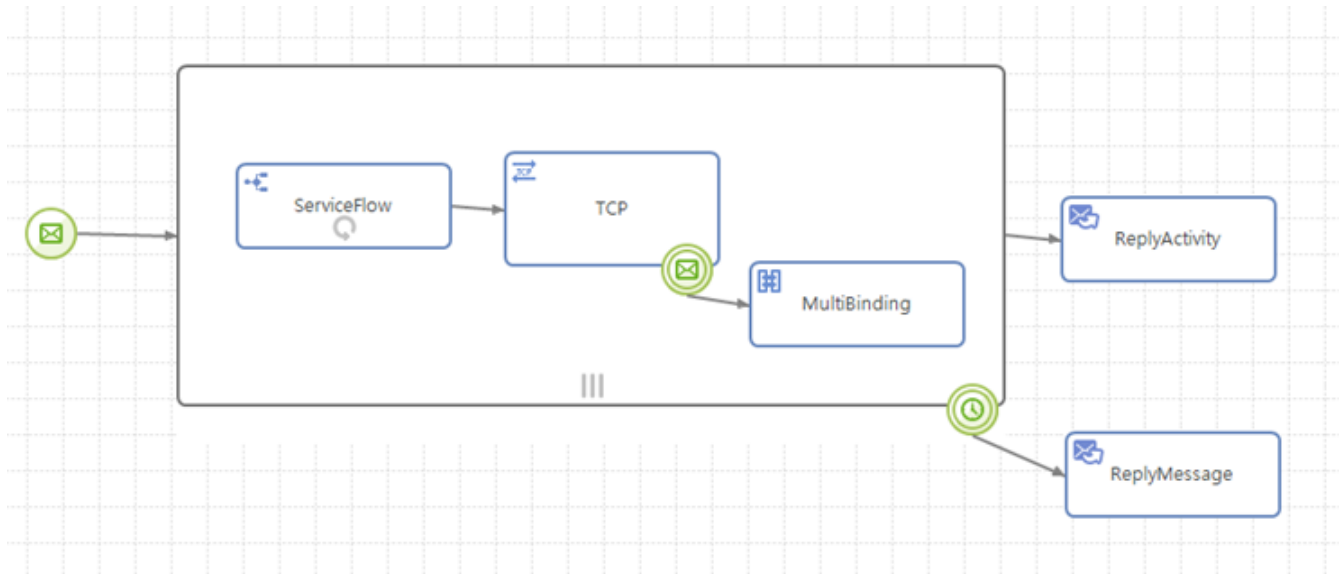
반면에 특정 구역을 진행하는 도중에 발생하는 사건들이나 예외에 대한 처리를 위해서는 특정 구역을 묶어서 표현하는 것이 효율적인데, 보통 이렇게 특정한 흐름의 단위를 묶는 것을 블록(Block)이라고 한다.

Java와 같은 프로그래밍 언어들은 기본 실행 단위가 여러 문장을 묶어놓은 블록이나 함수가 된다. 특정 블록이나 함수 내에서 발생한 에러들에 대해서는 에러 처리 로직에 의해 공통적으로 처리가 가능하다. AnyLink 플로우는 이러한 구역에 해당하는 액티비티 묶음을 서브 프로세스(Sub-Process) 또는 블록(Block)이라고 한다.

블록 안의 액티비티 혹은 트랜지션 수행 도중에 발생하는 에러나 타임아웃과 같은 이벤트는 블록 경계선에 붙여진 에러 이벤트나 타이머 이벤트에 의해 흐름이 이동하게 된다. 일반적으로 좋은 프로세스는 그래프 구조와 블록 구조의

복합적인 형태로 구성된다. AnyLink 서비스 플로우에는 그래프 구조를 기본으로 블록 구조를 부분적으로 지원하는 형태라고 볼 수 있다.

AnyLink 서비스 플로우 다이어그램에서 블록 액티비티와 블록 액티비티의 경계에 붙여지는 바운더리 이벤트들은 블록 구조의 흐름을 구현하는 예라고 볼 수 있다.



서비스 플로우의 블록 구조 요소들 - Block Activity

블록 액티비티 외에도 서비스 플로우에도 에러 핸들러 등을 지정할 수 있어 서비스 플로우 수행 중에 발생하는 에러를 처리할 수 있으므로 이 역시 블록 구조의 한 형태라고 볼 수 있다.

## 4.2. 기본 패턴

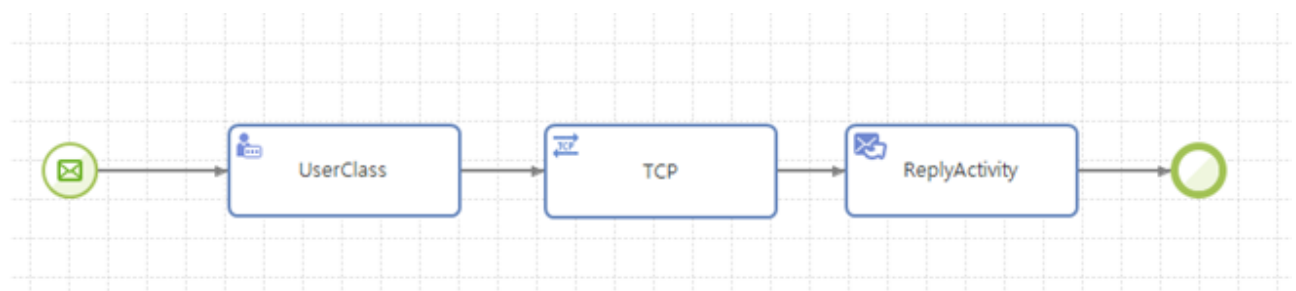
AnyLink 서비스 플로우 다이어그램은 앞에서 설명한 대로 BPMN 표준을 기반으로 그래프 구조와 블록 구조를 혼합하여 제어의 흐름을 표현한다. 본 절에서는 많이 사용되는 서비스 플로우 패턴에 대해서 설명한다.

### 4.2.1. 기본 흐름 패턴

다음은 기본 흐름 패턴에 대한 설명이다.

- 순차 실행(Sequence)

순차적인 흐름은 가장 기본적인 흐름이며, 각 액티비티들은 그 순서에 따라 앞의 액티비티가 끝나야 다음 액티비티가 수행된다.



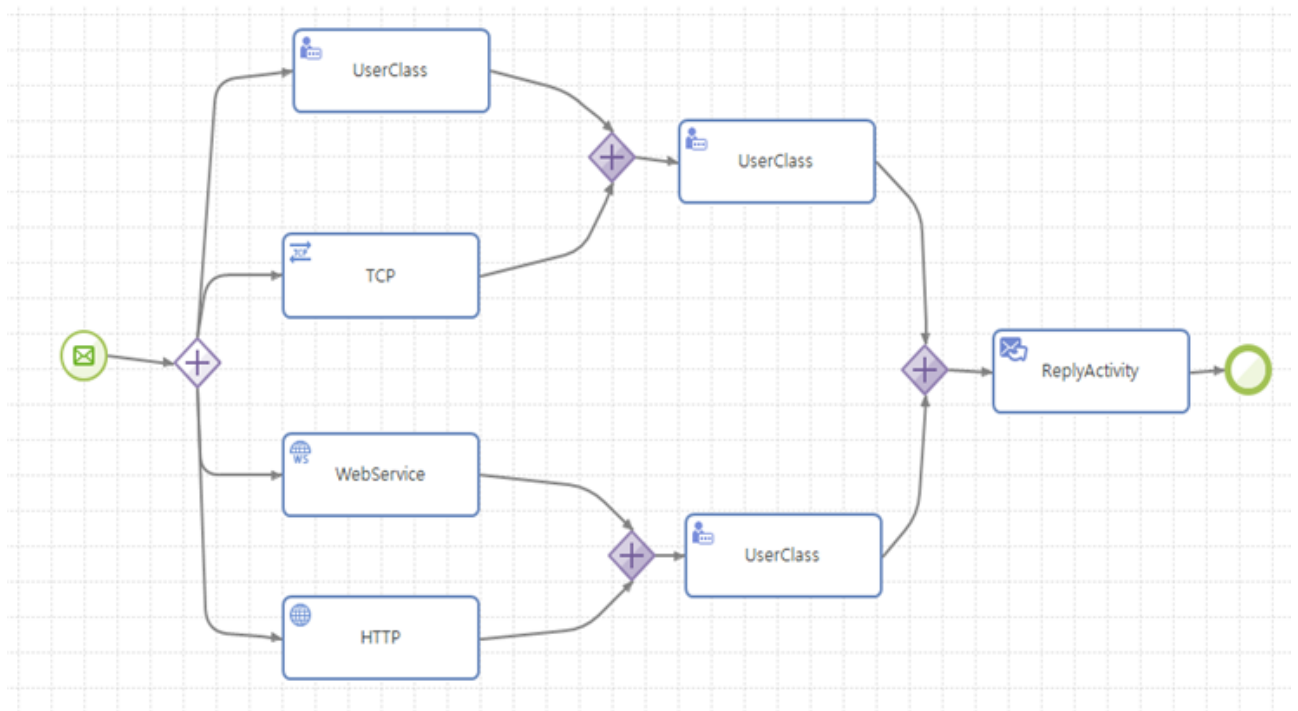
순차 실행

- 병렬 수행을 위한 분기(Parallel Split)

하나의 액티비티가 여러 개의 액티비티로 나누어져서 진행되며, 각각의 액티비티는 독립적으로 실행된다. 보통 And Split 게이트웨이를 사용하여 표현한다.

- 동기화(Synchronization)

병렬적으로 수행적인 스레드를 동기화하여 하나의 흐름으로 합친다. 보통 And Join 게이트웨이를 사용하여 표현한다.



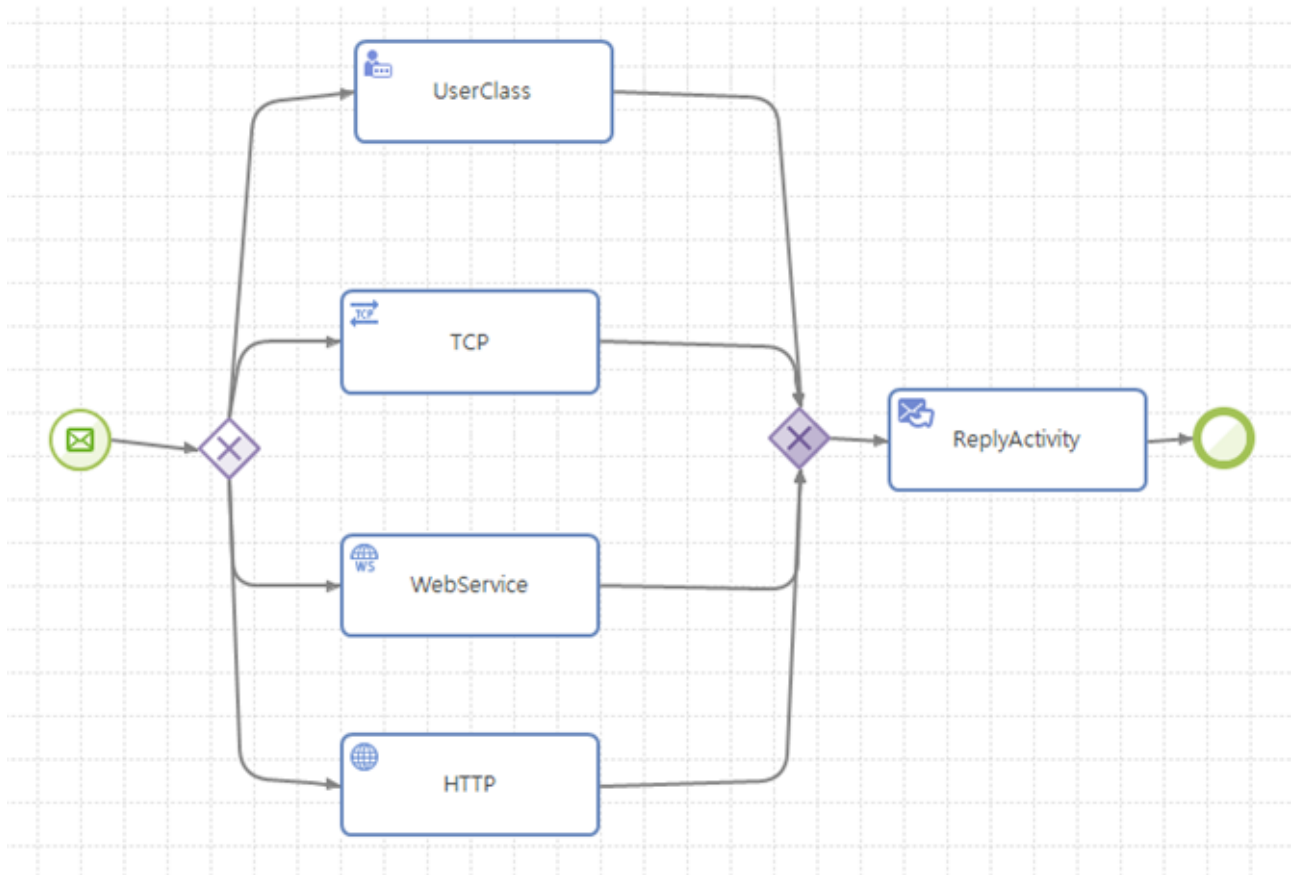
병렬 수행 분기 및 동기화

- 배타적인 분기(Exclusive Choice)

여러 실행 경로들 중 하나의 실행 경로를 선택하는 형태이다. 보통 XOR Split 게이트웨이를 사용하여 표현한다.

- 단순 병합(Simple Merge)

여러 개의 실행 경로 중 하나의 실행 경로만 허용하여 단순 병합하는 형태이다. 보통 XOR Join 게이트웨이를 사용하여 표현한다.



배타적인 분기와 단순 병합

## 4.2.2. 확장된 분기 및 병합 패턴

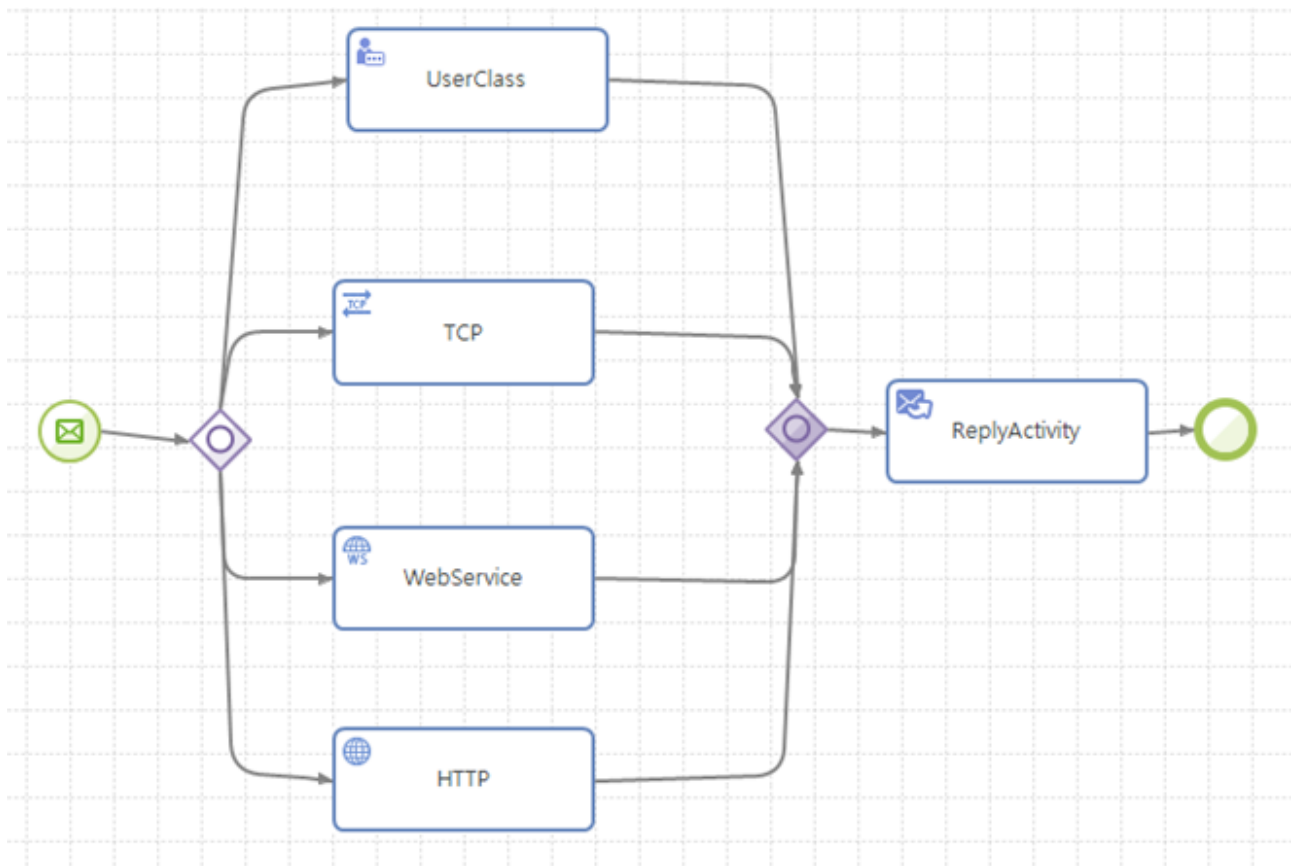
다음은 확장된 분기 및 병합 패턴에 대한 설명이다.

- 다중 선택(Multiple Choice)

여러 개의 실행 경로 중 몇 개의 실행 경로를 선택하여 병렬적으로 실행하는 형태이다. 보통 OR Split 게이트웨이를 사용하여 표현한다.

- 동기 방식 병합(Synchronizing Merge)

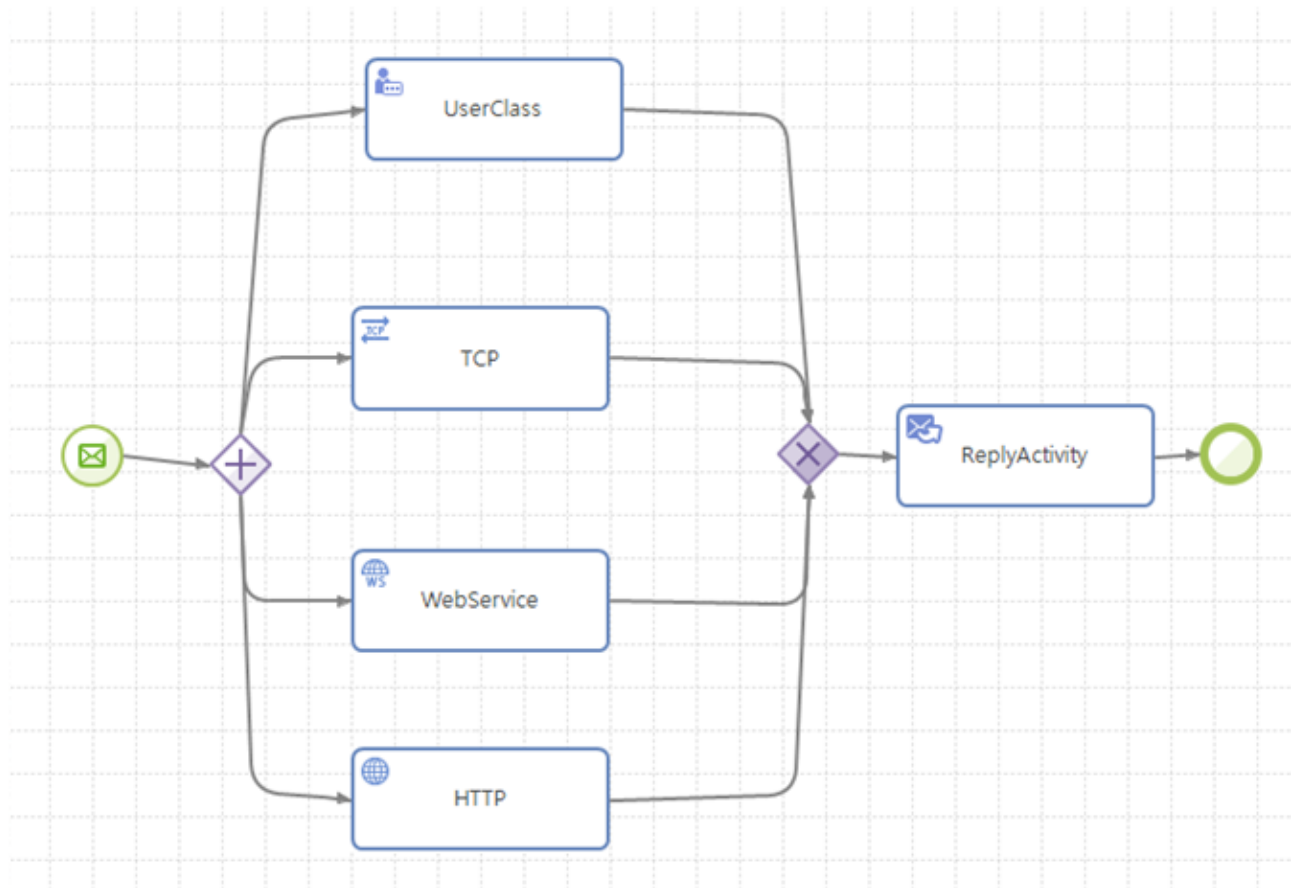
여러 개의 실행 경로를 동기화시키면서 병합하는 형태이다. 보통 OR Join 게이트웨이를 사용하여 표현한다.



다중 선택과 동기 방식 병합

#### • 동기화 없는 병합(Multiple Merge)

여러 개의 실행 경로를 동기화 없이 경로만 하나로 합치는 형태이다. 여러 개의 경로가 실행되고 있는 상태라면 이후에는 하나의 경로로 여러 번 지나가게 된다. And Split 게이트웨이 다음에 XOR Join 게이트웨이로 병합하면 다음과 같이 동작한다.



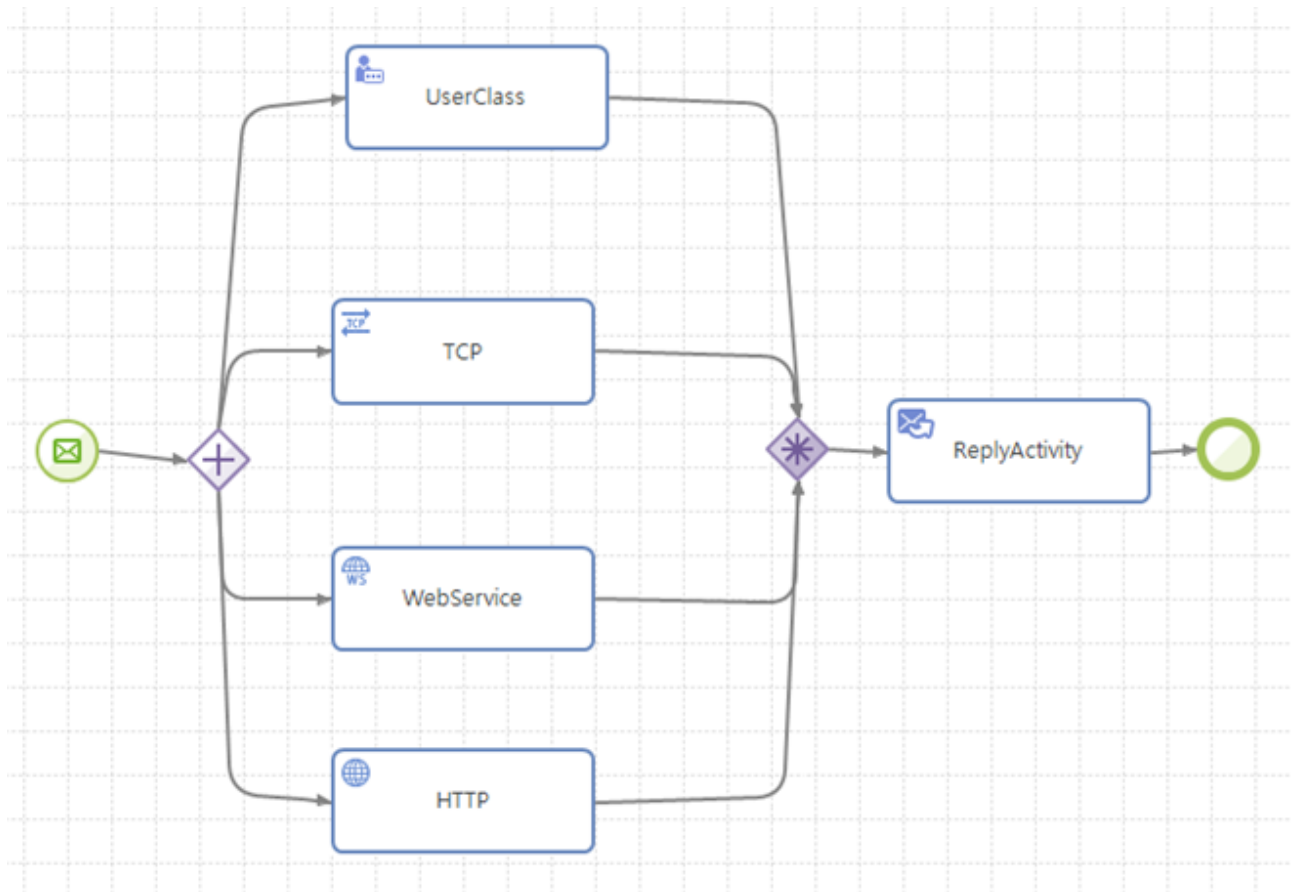
동기화 없는 병합

#### • 동기화 없이 하나만 흘림 병합(Discriminator)

여러 개의 실행 경로를 동기화 없이 첫 번째 도달하는 흐름만 계속 흘러보내는 형태이다. 여러 개의 경로가 실행되고 있는 상태라면 이후에는 하나의 경로로 가장 먼저 도달한 스레드만 지나가게 된다.

And Split 게이트웨이 다음에 Complex Join 게이트웨이로 병합할 때 Complex Join 게이트웨이의 흐름 조건(flow condition)이 1이면 다음과 같이 동작한다.





동기화 없이 하나만 흐름 병합

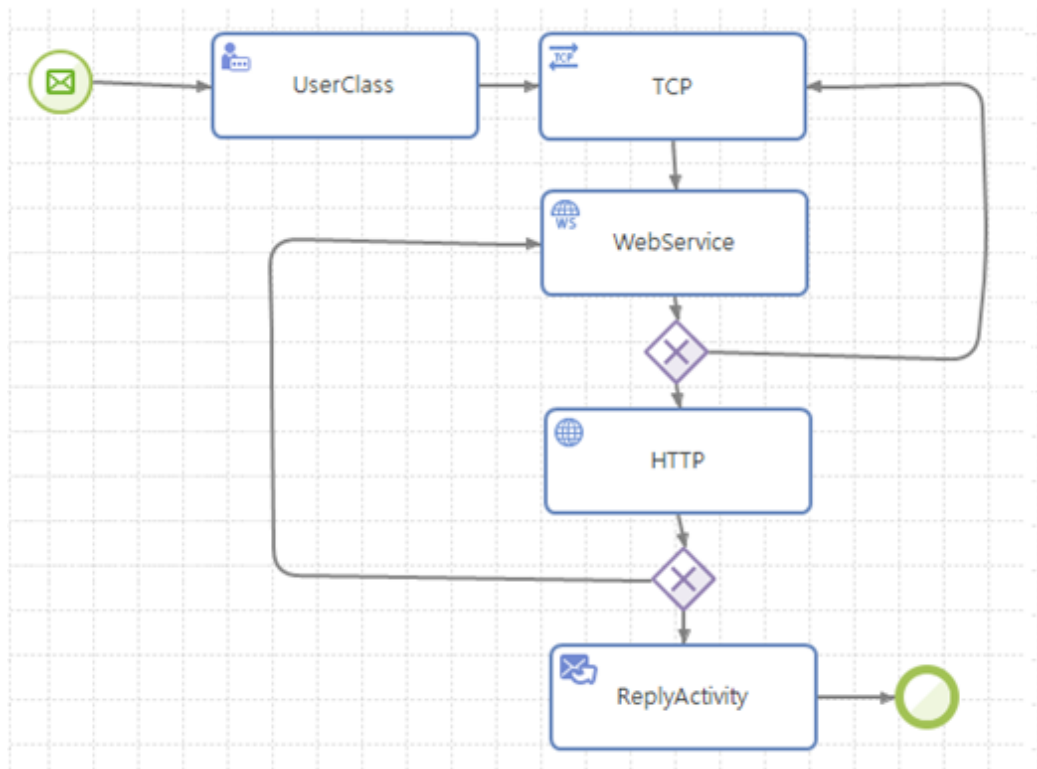
#### • M 개 중 N 개 병합(N-out-of-M Join)

여러 개의 실행 경로를 지정한 갯수만큼만 도달할 때까지 동기화한 후에 하나로 흘러보내는 형태이다. 이 게이트웨이 이후의 경로는 하나의 흐름만 지나가게 된다.

And Split 게이트웨이 다음에 Complex Join 게이트웨이로 병합할 때 Complex Join 게이트웨이에 들어오는 흐름 갯수 설정이 N에 해당한다.

#### • 임의의 사이클 형태 반복(Arbitrary Cycle)

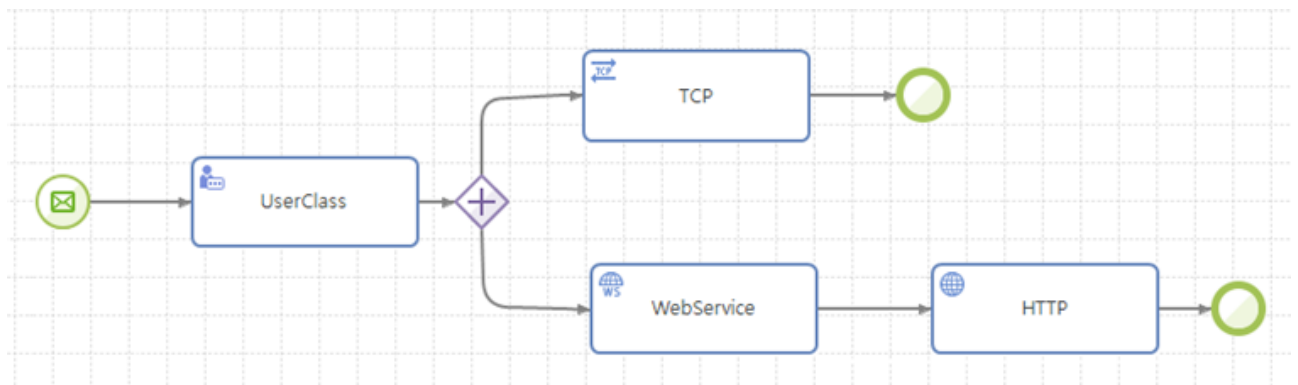
AnyLink에서 반복을 표현하기 위해 블록의 반복 설정을 사용할 수도 있지만 좀더 자유로운 반복을 위해 트랜지션을 통해 사이클을 형성하게 하여 반복시킬 수도 있다.



임의의 사이클 형태 반복

#### • 암묵적 종료(Implicit Termination)

서비스 플로우를 명시적으로 종료시키기 위해 Terminate 이벤트를 사용할 수 있지만 실행 중이던 각 흐름이 모두 끝나게 되면 암묵적으로 종료하게 된다.



암묵적 서비스 플로우 종료

#### • 다중 인스턴스(Multiple Instance)

다중 인스턴스는 하나가 아니라 여러 건을 순차적으로 혹은 병렬적으로 처리하는 인스턴스를 표현하는 액티비티 실행 형태이다. 액티비티는 하나이지만 실제 인스턴스는 여러 개가 생성된다.

다중 인스턴스는 반복 완료 조건과 반복 수행 형태, 다음 액티비티로의 트랜지션 방식 속성을 설정한다.

속성	설명
반복 완료 조건	인스턴스 갯수를 뜻하는 숫자값이나 완료 조건을 의미하는 bool 조건식을 지정한다.

속성	설명
반복 수행 형태	<p>순차(Sequential)와 병렬(Parallel) 두 가지가 있다.</p> <p>순차 수행 형태는 일반적인 단순 반복문과 같은 방식으로 수행하며, 병렬 수행 형태는 동시에 여러 건이 실행된다. 병렬 수행 형태는 마치 And Split으로 여러 경로가 실행되는 것과 비슷한 방식으로 여러 인스턴스가 실행된다.</p>
다음 액티비티로의 트랜지션 방식	<p>다음 액티비티로의 트랜지션 방식은 네 가지 형태로 구분된다.</p> <ul style="list-style-type: none"> <li>• 첫 번째는 모든 인스턴스들이 완료 후에 트랜지션이 일어나는 경우로 마치 And Join과 비슷한 방식으로 동작한다.</li> <li>• 두 번째는 여러 인스턴스들 중 하나의 인스턴스가 완료되면 바로 트랜지션이 일어나는 경우이다.</li> <li>• 세 번째는 여러 인스턴스들 중 각 인스턴스가 완료될 때마다 트랜지션이 하나씩 일어나는 경우이다. 이 경우에는 트랜지션부터는 인스턴스 갯수만큼 여러 번 흐르게 된다.</li> <li>• 네 번째는 지정한 조건이 만족되면 트랜지션이 일어나는 경우이다. bool 조건식 값이 만족되면 트랜지션이 일어난다.</li> </ul>

다음은 다중 인스턴스에 관련된 몇 가지 주요 사용 패턴이다.

◦ 동기화 없는 다중 인스턴스 패턴(MI Without Synchronization)

하나의 액티비티의 여러 인스턴스들을 동기화하지 않고 생성하는 경우로 MI 트랜지션 방식 중 인스턴스 완료때마다 트랜지션이 일어나는 경우가 여기에 해당한다.

◦ 설계 시점에 인스턴스 갯수를 알고 있는 다중 인스턴스 패턴(MI With A Priori Known Design Time Knowledge)

설계 시점에 MI 방식으로 실행할 액티비티의 생성할 인스턴스 갯수를 알고 있는 형태로 반복 완료 조건에서 MI 카운트를 상수로 지정하는 경우이다.

◦ 런타임 시점에 인스턴스 갯수를 미리 알 수 있는 다중 인스턴스 패턴(MI With A Priori Known Runtime Knowledge)

MI 방식으로 실행할 액티비티의 생성할 인스턴스 갯수를 실행 중 어떤 시점에서 미리 알 수 있는 형태로 반복 완료 조건을 변수값으로 지정하는 경우가 여기에 해당한다.

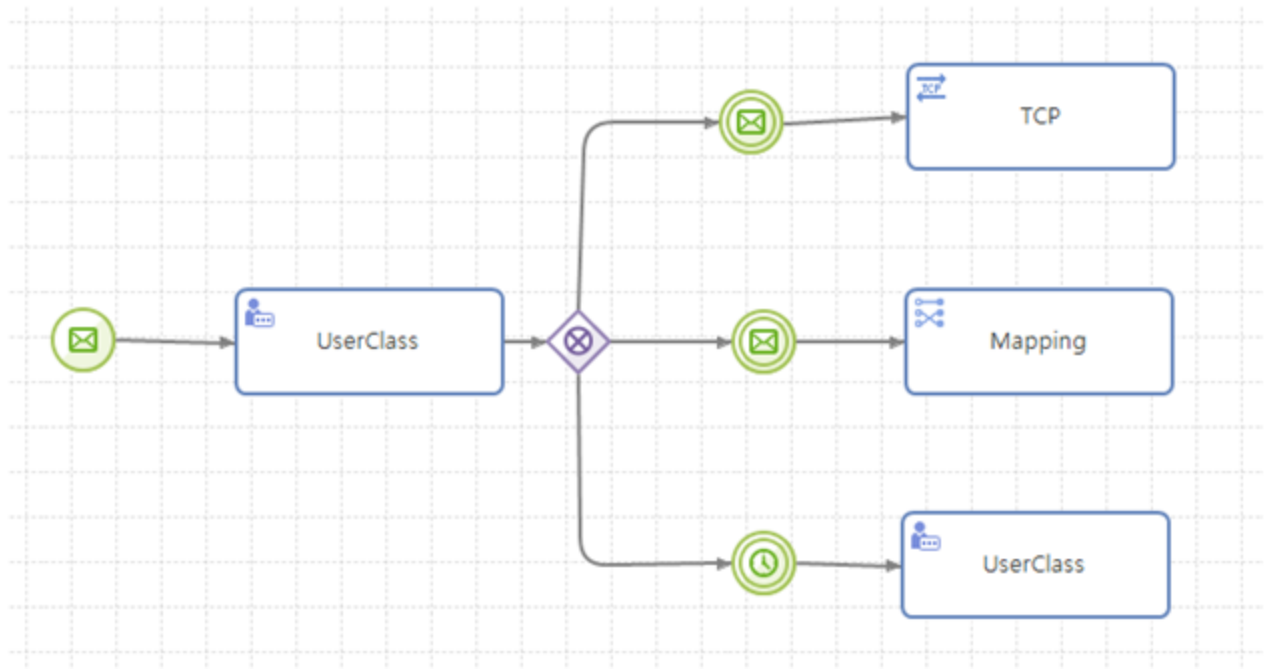
◦ 런타임에 미리 알 수 없는 다중 인스턴스 패턴(MI With No A Priori Runtime Knowledge)

MI 방식으로 실행할 액티비티의 인스턴스 갯수를 실행 중에도 미리 알 수 없는 형태로 반복 완료 조건이 bool 조건문일 경우가 대부분 여기에 해당한다.

• 지연 선택(Deferred Choice)

여러 개의 경로 중 하나의 경로를 선택한다는 점에서 XOR Gateway와 경로면에서는 유사하나 여러 경로의 대기중인 이벤트들 중 가장 먼저 일어나는 이벤트에 의해 실행될 하나의 경로가 결정된다는 점에서 큰 차이가 있다.

AnyLink 서비스 플로우 다이어그램에서는 주로 XOREvent Gateway가 이러한 지연 선택을 표현한다.



지연 선택

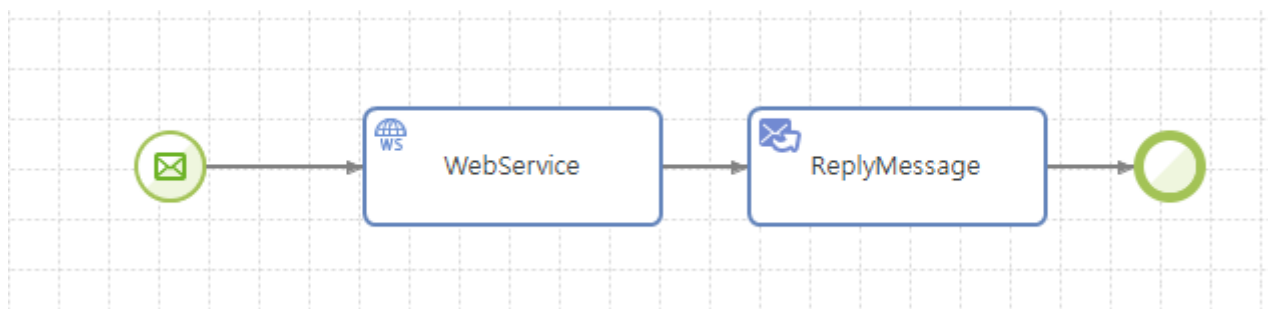
## 4.3. 서비스 구현 방식과 서비스 플로우 형태

본 절에서는 다양한 형태의 서비스 구현 방식과 서비스 플로우 형태에 대해서 설명한다.

### 4.3.1. Proxy 서비스

기존에 존재하거나 별도로 구축되는 서비스를 AnyLink를 통해서 서비스를 하도록 서비스 엔드포인트를 AnyLink 한 곳으로 통합하고 로깅이나 보안 등 추가적인 기능들을 기존 서비스에 추가하는 형태의 서비스 모델이다. 예를 들어 기존 서비스가 웹서비스 형태인 경우 해당 서비스의 WSDL(웹 서비스 기술 언어로 작성된 서비스 기술 문서)을 import하여 wrapper 서비스를 만드는 형태가 된다.

서비스 플로는 Incoming 메시지 이벤트와 Outgoing 서비스 액티비티, 그리고 Reply 액티비티를 기본으로 하는 형태이며 그 가운데 필요한 로깅, 보안 등을 처리하기 위한 액티비티들이 삽입되는 형태가 기본이 된다.

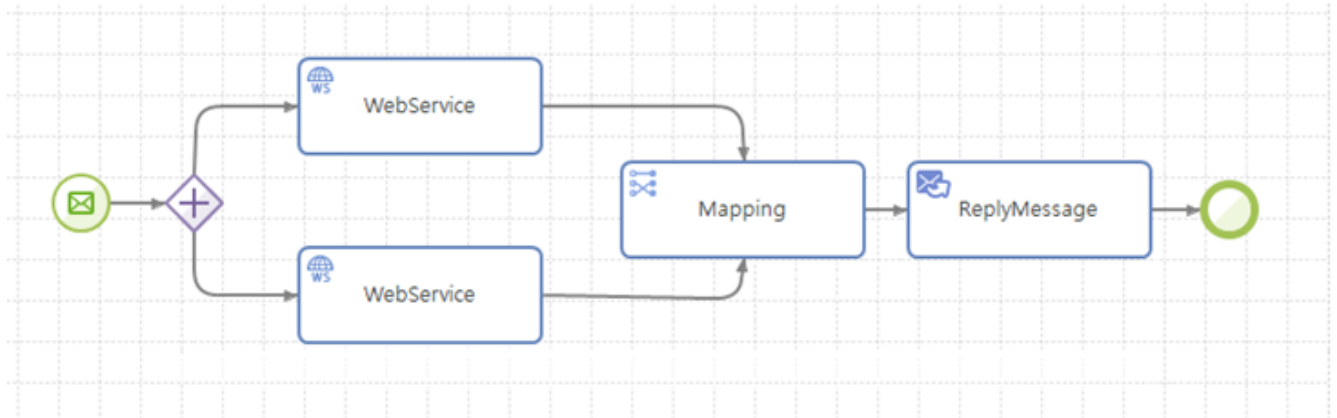


Proxy 서비스를 구현하는 서비스 플로우 형태

### 4.3.2. Composite 서비스

기존에 존재하거나 별도로 구축되는 여러 가지 서비스들을 AnyLink가 조합하여 새로운 서비스를 만드는 형태의 서비스 모델이다.

서비스 플로우는 하나의 Incoming 메시지 이벤트와 여러 개의 Outgoing 서비스 액티비티, 그리고 Reply 액티비티를 기본으로 하는 형태이며 그 가운데 필요한 로깅, 보안, 메시지 변환 등의 액티비티들이 삽입되는 형태가 기본이 된다.

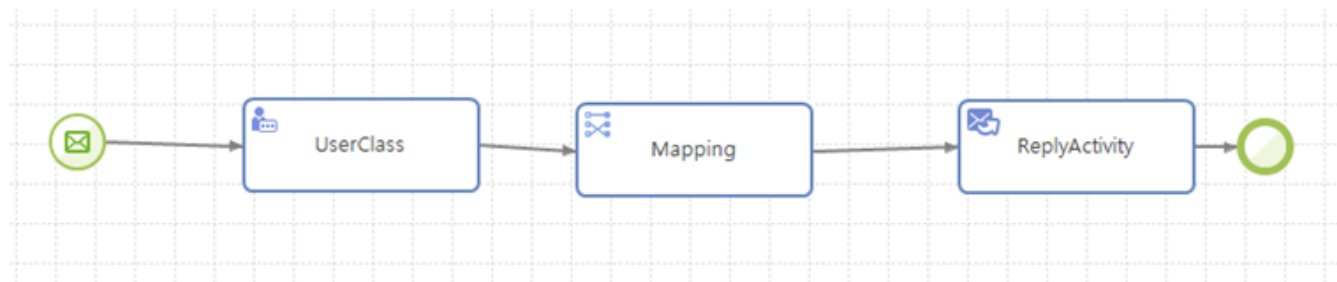


Composite Service를 구현하는 서비스 플로우 형태

### 4.3.3. 서비스 구현

AnyLink가 새로운 서비스를 구현(Service Implementation)하는 형태로 새로운 서비스의 로직은 대부분 서비스 플로우의 사용자 클래스 액티비티와 다양한 어댑터 액티비티, 분기 처리 등을 통하여 직접 구현하게 된다.

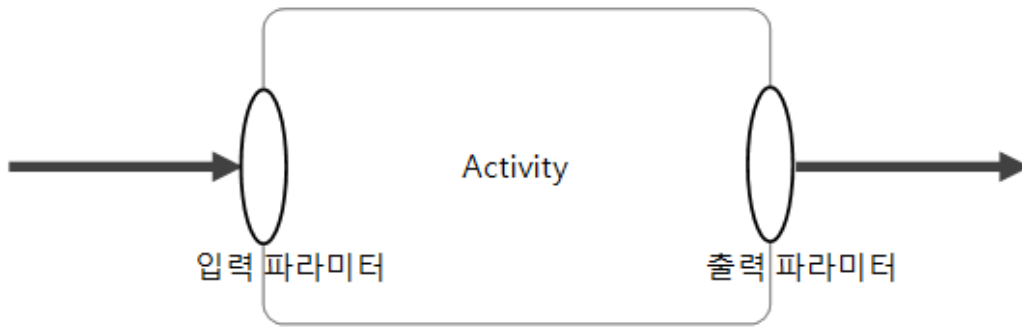
서비스 플로우는 하나의 Incoming 메시지 이벤트와 Reply 액티비티, 그리고 그 가운데 필요한 다양한 어댑터 서비스 액티비티와 사용자 클래스 액티비티들이 삽입되는 형태가 기본이 된다.



서비스 구현 형태의 서비스 플로우

## 4.4. 서비스 플로우 변수와 액티비티 파라미터

AnyLink 서비스 플로우의 가장 기본적인 형태는 액티비티와 이벤트를 트랜지션으로 연결하여 흐름을 표현하는 것이다. 이때 액티비티는 입력, 출력에 해당하는 파라미터를 가진다.



#### 액티비티의 입력과 출력

- 입력 파라미터는 해당 액티비티가 읽기 위해 사용하는 변수로 해당 변수는 프로세스나 액티비티를 둘러싼 블록 액티비티에 선언된다.
- 출력 파라미터는 해당 액티비티가 쓰기 위해 사용하는 변수로 해당 변수는 역시 프로세스나 액티비티를 둘러싼 블록 액티비티에 선언된다.

서비스 플로우의 변수는 서비스 플로우 즉, 프로세스에 선언되거나 액티비티들을 묶어둔 블록 액티비티에 선언할 수 있는데 각 변수를 볼 수 있는 범위(scope)는 프로세스 변수의 경우 프로세스에 속한 모든 액티비티와 이벤트, 트랜지션 등이며 블록 변수의 경우는 해당 블록 액티비티 안에 속한 액티비티, 이벤트, 트랜지션 등이다.

## 4.5. 서비스 액티비티와 파라미터

AnyLink 서비스 플로우에서 다른 서비스를 호출하는 것은 주로 서비스를 나타내는 액티비티를 통한 호출로 표현된다.

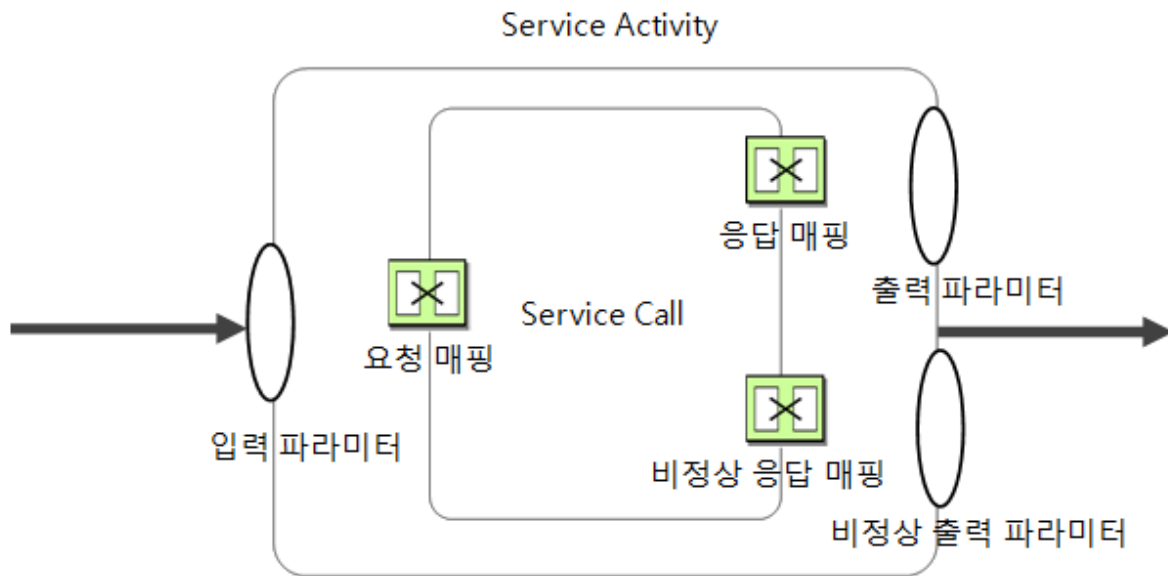
서비스는 크게 서비스 플로우 엔진에서 관리하는 각 서비스 플로우들의 **메시지 이벤트**들과 **어댑터의 아웃바운드 룰**, **멀티바인딩 라우터 룰**로 나눌 수 있다.

서비스 중 하나의 서비스 플로우에서 다른 서비스 플로우를 호출하는 경우는 **서비스 플로우 액티비티**라고 한다. 어댑터의 아웃바운드 룰 호출은 각 어댑터별 액티비티(TCP, HTTP, Tmax, 웹서비스 등의 액티비티)로 각각 표현된다. 마찬가지로 멀티바인딩 라우터 룰을 호출할 때에는 멀티바인딩 액티비티를 사용한다.

서비스 호출 액티비티들은 호출하는 각 서비스의 특성은 다를 수 있지만 서비스 플로우 관점에서는 딜리버리 채널을 통해 서비스를 호출한다는 점에서 동작 방식이 모두 같다. 다만 이 중 다른 서비스 플로우를 호출하는 경우는 내부적으로 서비스 플로우 엔진 자신을 호출하기 때문에 딜리버리 채널을 통하지 않지만 논리적으로는 동일하다.

AnyLink에서 서비스는 요청과 응답, 그리고 비정상 응답 메시지를 입출력 메시지로 나뉜다. **서비스 액티비티**는 액티비티들 중 서비스를 호출하는 액티비티이므로 입력, 출력 파라미터 외에 비정상 출력 파라미터를 추가적으로 선언할 수 있다.

일반적인 서비스 액티비티에서는 입력 파라미터는 서비스의 요청 메시지로 전달되며, 서비스 호출 결과인 응답 혹은 비정상 응답 메시지는 각각 출력 파라미터, 비정상 출력 파라미터에 저장된다. 입력 파라미터와 요청 메시지 서식이 다른 경우에는 서비스 액티비티의 요청 매핑을 사용하여 데이터 변환을 지정할 수 있다. 마찬가지로 응답 메시지와 출력 메시지 서식이 다른 경우에는 응답 매핑, 비정상 응답 메시지와 비정상 출력 메시지 서식이 다른 경우에는 비정상 응답 매핑을 사용하여 데이터 변환을 할 수 있다.



서비스 액티비티의 입력

## 4.6. 매핑

AnyLink 서비스 플로우에서 데이터 간의 변환은 매핑(Mapping) 액티비티나 서비스 액티비티의 요청 매핑, 응답 매핑 및 비정상 응답 매핑을 통해 이루어진다.

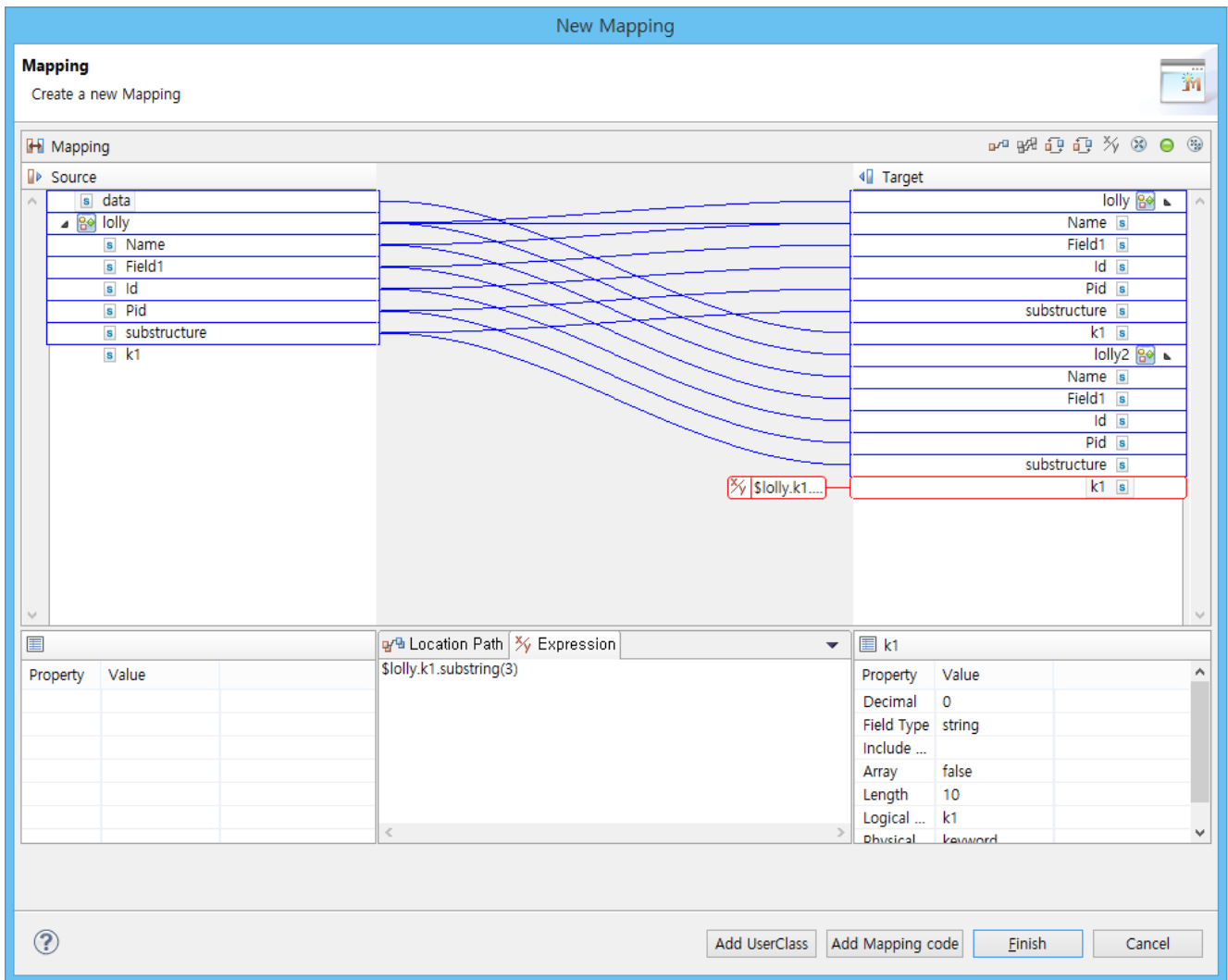
AnyLink 서비스 플로는 프로세스 내에서 공유하는 프로세스 변수를 선언할 수가 있다. 마찬가지로 블록 액티비티는 블록 내에서 공유하는 블록 변수를 선언할 수가 있다.

변수는 간단한 String, Float, Integer, DateTime, Boolean 자료형을 지원하고 이외에 구조화된 데이터 형태인 메시지 자료형을 지원한다. AnyLink 서비스 플로우에서 다루는 메시지 자료형은 Java 프로그래밍 언어에서 데이터를 나타내는 클래스로 표현된다. Java 프로그래밍 언어에서 Value Object 혹은 Data Transfer Object라고 부르는 형태의 클래스들이라고 생각하면 된다.

AnyLink 서비스 플로우에서 변수는 내부적으로 이렇게 Java 클래스로 변환되어 표현되고, 매핑은 실행 성능을 개선하기 위해 Java 객체들 간에 데이터를 변환하고 전달하는 Java 코드로 생성된다. 매핑의 가장 기본적인 형태는 소스 메시지와 타겟 메시지의 각 필드들을 연결하는 형태이다. 즉, 소스 메시지의 특정 필드값을 타겟 메시지의 특정 필드에 대입(assign)하는 형태가 가장 기본이다.

메시지의 필드 외에 표현식(expression)을 소스로 사용할 수 있는데 AnyLink 서비스 플로우 내에 사용 가능한 표현식에 대해서는 다음에서 별도로 설명한다. AnyLink 서비스 플로우의 매핑 액티비티는 이러한 프로세스 변수 혹은 블록 변수들 사이의 데이터 변환을 지시하는 액티비티이다.

소스 변수와 타겟 변수들을 지정하고 이들간의 변환 규칙을 그림으로 지정할 수가 있다.



매핑 액티비티에서 매핑 에디터

서비스 액티비티에서의 요청, 응답, 비정상 응답 매핑은 어댑터 룰이나 다른 서비스 플로우를 호출하는 액티비티, 그리고 메시지 이벤트를 통해 받은 요청 메시지에 대한 응답 메시지를 돌려주는 Reply 액티비티 등에서는 어댑터 룰이나 서비스 플로우의 서비스 요청 데이터와 응답 데이터 혹은 비정상 응답 데이터의 서식을 맞추기 위해 데이터 변환을 각각 할 수 있는 기능을 제공한다.

어댑터 룰이나 다른 서비스 플로우의 시작 이벤트를 호출할 때 필요한 요청 데이터를 위한 요청 매핑, 그리고 결과값을 받은 후 이의 서식을 변경하기 위해 필요한 응답 매핑, 그리고 비정상 응답 메시지가 전달되었을 때의 비정상 응답 매핑이 있다.

요청 매핑은 서비스 액티비티의 입력 파라미터를 소스로 하고 호출할 서비스(즉, 어댑터 룰이나 다른 서비스 플로우)의 요청 메시지 형태를 타겟으로 하는 매핑이다. 응답 매핑은 서비스의 응답 메시지를 소스로 하고, 서비스 액티비티의 출력 파라미터를 타겟으로 하는 매핑이다.

비정상 응답 매핑은 응답 매핑과 유사하며, 서비스에서 비정상 응답 메시지를 보낸 경우이다. 서비스의 비정상 응답 메시지를 소스로 하고, 서비스 액티비티의 비정상 출력 파라미터를 타겟으로 하는 매핑이다. Reply 액티비티에서는 응답 매핑과 비정상 응답 매핑을 지정할 수 있다.

## 4.7. 서비스 플로우 표현식

AnyLink 서비스 플로우의 조건 분기를 나타내는 식이나 코릴레이션 값 계산을 위한 식에서 변수에 해당하는 데이터



객체의 특정 부분값을 계층적으로 표현하기 위해 표현식(Expression)을 사용한다. 표현식은 서비스 플로우 내의 조건 분식 표현식이나 코릴레이션 계산식 외에도 멀티바인딩 라우터의 조건 표현식 등에서도 사용된다.

AnyLink 서비스 플로우 엔진은 자주 호출되는 표현식의 성능을 고려하여 표현식을 포함한 각 요소들을 Java 코드로 생성하여 컴파일하여 배포하는 방식으로 구현되어 있다.

### 4.7.1. 구조적 표현식

표현식의 가장 기본적인 서식은 '변수 이름'.'필드 이름' 형태이다. 예를 들어 child라는 변수가 있고 이 변수에 name이라는 필드가 정의되어 있다면 child.name으로 소스를 표현할 수 있다. 소스로 사용될 경우 child.name은 Java 코드로 변환될 때 child.getName()과 같은 형태로 변환된다. 이러한 구조적 형태를 표현하는 AnyLink 서비스 플로우의 표현식을 구조적 표현식(structured expression)이라고 한다.

매핑의 각 대입 연산(assignment operation)에서 소스와 타겟은 모두 구조적 표현식으로 지정할 수가 있다. 소스의 경우는 구조적 표현식 외에도 다양한 형태의 함수를 사용할 수가 있다. Java 코드로 생성되는 표현식의 특성을 고려하여 Java 메소드나 Java 연산자를 매핑 소스의 표현식에 사용할 수가 있게 되어 있다.

예를 들어 child라는 변수의 name 필드가 Java 코드 상으로 String 클래스라면 다음과 같은 표현식을 사용할 수가 있다.

```
child.name.substring(3) + "_ChildName"
```

위 표현식은 다음과 같은 Java 코드 형태로 변환된다.

```
child.getName().substring(3) + "_ChildName"
```

### 4.7.2. 변수 표현

변수 이름은 특별한 표기가 없으면 구조적 표현식의 시작 부분 토큰을 변수 이름으로 간주한다.

서비스 플로우에서 변수는 매핑 등의 상황에서 내부적으로 미리 정의되어 있어 이에 맞게 코드를 생성한다. 서비스 플로우 실행 도중에 서비스 플로우 변수나 블록 변수를 참조할 필요가 있을 수 있다. 이렇게 문맥 속에 정의된 변수는 '\$' 표기를 통해 변수를 표현한다. 예를 들어 dataField1이라는 변수가 선언되어 있다면 다음과 같이 사용될 수 있다.

```
$dataField1.content
```

표현식은 앞서도 설명한 대로 Java 코드로 생성된다. 변수나 필드의 자세한 자료형을 알 수 없으면 정확하게 Java 코드를 생성할 수가 없다.

서비스 플로우 내에 정의된 변수들은 그 정의에서 추출하여 내부적으로 표현식을 Java 코드로 변환할 때 실제 자료형을 추정하여 코드를 생성한다. 변수의 자료형을 정확하게 지정하고 싶은 경우에는 '<>'를 사용하여 지정할 수 있다. 예를 들어 dataField1이라는 변수의 정확한 자료형이 com.tmax.anylink.ContentContainer 클래스라면 다음과 같이 표현할 수 있다. AnyLink는 내부적으로 표현식을 Java 코드로 변환하여 컴파일하기 때문에 정확한 자료형을 알 수 없는 경우에는 배포할 때 에러가 발생한다. AnyLink 내부적으로 자료형을 추정할 수 없는 경우에

이렇게 명시적으로 자료형을 지정하여 컴파일 에러를 회피할 수 있다.

```
$dataField1<com.tmax.anylink.ContentContainer>.content
```

위 표현식은 다음과 같은 Java 코드 형태로 변환된다.

```
((com.tmax.anylink.ContentContainer) _varCtx.getVariable("dataField1")).getContent()
```

### 4.7.3. 매핑 표현식

서비스 액티비티에서 입출력 데이터를 변환하거나 서비스 플로우 내의 변수값을 변경하기 위한 **매핑 액티비티**에서 매핑을 사용한다. 매핑에서도 소스가 되는 데이터를 입력값이 아닌 별도의 표현식으로 표현할 수가 있는데 이때의 표현식은 지금 설명한 조건문이나 코릴레이션 값 계산식 등의 표현식과는 조금 형태가 다르다.

매핑의 소스에 사용되는 표현식의 기본 형태는 동일하게 구조적 표현식이다. 즉, 점(.)으로 구분하여 특정 객체 값의 필드를 지정한다.

점(.)으로 구분된 계층 구조에서 매핑 표현식의 첫 번째에 나타나는 부분은 소스 혹은 타겟으로 지정된 변수명이다. 매핑 표현식은 일반 표현식과 달리 '\$'로 시작하는 변수명을 지원하지 않고, 구조적 표현식의 시작 부분 토큰이 변수명이 된다. 즉, 아래 예에서 input1이 변수명이다.

```
input1.person.name
```

내부적으로는 소스에 사용된 필드 객체가 null이고, 하위 필드 객체를 다시 계층적으로 지정한 경우(즉, input1.getPerson() 값이 null일 때 input1.person.name을 지정한 경우) 해당하는 객체를 빈 객체로 생성하여 에러가 발생하지 않도록 동작한다.

이외에도 매핑의 필드가 배열일 경우 소스에 사용된 필드 이름과 타겟에 사용된 필드 이름 뒤에 애스터리스크(\*)를 붙여 배열 매핑을 표현할 수 있다. 소스의 중간 노드에 해당하는 필드가 null일 경우 빈 객체를 만들어주는 기능과 배열 매핑 외에도 함수를 표현하는 방법도 매핑 표현식은 일반 표현식과 형태가 다르다.

매핑 표현식에서는 '@'로 시작하는 형태로 함수를 표현한다.

```
@substring(input1.person.name, 0, 4)
```

매핑 표현식에서 사용할 수 있는 함수들은 다음과 같다.

함수	설명
@arraySize(arg1)	배열의 크기를 리턴한다. ◦ arg1 : 구조적 표현식이다.

함수	설명
@date(arg1?)	<p>인자가 없을 경우 기본 서식 형태(yyyy-MM-dd)로 현재 시간을 리턴한다.</p> <ul style="list-style-type: none"> <li>◦ arg1 : Java의 SimpleDateFormat에서 사용할 수 있는 날짜 서식 패턴 문자열이다.</li> </ul>
@time(arg1)	<p>인자가 나타내는 서식 형태로 현재 시간을 리턴한다.</p> <ul style="list-style-type: none"> <li>◦ arg1 : Java의 SimpleDateFormat에서 사용할 수 있는 날짜 서식 패턴 문자열이다.</li> </ul>
@strlen(arg1)	<p>문자열의 길이를 리턴한다.</p> <ul style="list-style-type: none"> <li>◦ arg1 : String 객체를 나타내는 구조적 표현식이다.</li> </ul>
@trim(arg1)	<p>문자열의 앞뒤에 존재하는 공백문자들을 제거한 문자열을 리턴한다.</p> <ul style="list-style-type: none"> <li>◦ arg1 : String 객체를 나타내는 구조적 표현식이다.</li> </ul>
@substring(arg1, arg2, arg3?)	<p>구조적 표현식이 나타내는 문자열의 부분 문자열을 리턴한다.</p> <ul style="list-style-type: none"> <li>◦ arg1 : 문자열을 나타내는 구조적 표현식이다.</li> <li>◦ arg2 : 시작 오프셋이다.</li> <li>◦ arg3 : 끝 오프셋이다.</li> </ul>
@concat(arg1, arg2?, arg3?, ...)	<p>여러 개의 문자열을 하나로 합친 문자열을 리턴한다. 각 인자들은 String, char, 혹은 숫자값을 나타내는 구조적 표현식들이다.</p>
@replace(arg1, arg2, arg3)	<p>문자열의 일부를 다른 문자열로 치환한 문자열을 리턴한다.</p> <ul style="list-style-type: none"> <li>◦ arg1 : 원 문자열을 나타내는 구조적 표현식이다.</li> <li>◦ arg2 : 찾는 부분 문자열이다.</li> <li>◦ arg3 : 대체할 부분 문자열을 각각 나타낸다.</li> </ul>
@dateformat(arg1, arg2, arg3)	<p>특정 날짜 서식으로 표현된 문자열을 다른 날짜 서식으로 변환한다.</p> <ul style="list-style-type: none"> <li>◦ arg1 : 특정 날짜 서식으로 표현된 문자열을 나타내는 구조적 표현식이다.</li> <li>◦ arg2 : 파싱할 특정 서식이다.</li> <li>◦ arg3 : 변환시킬 대상 날짜 서식을 각각 나타낸다.</li> </ul>
@charAt(arg1, arg2)	<p>문자열의 특정 위치 문자를 리턴한다.</p> <ul style="list-style-type: none"> <li>◦ arg1 : 문자열을 나타내는 구조적 표현식이다.</li> <li>◦ arg2 : 위치를 나타낸다.</li> </ul>
@isNull(arg1)	<p>구조적 표현식의 결과값이 null인지 여부를 리턴한다.</p> <ul style="list-style-type: none"> <li>◦ arg1 : 구조적 표현식이다.</li> </ul>
@numberformat(arg1, arg2)	<p>숫자값을 특정 서식으로 변환한다.</p> <ul style="list-style-type: none"> <li>◦ arg1 : 변환할 서식이다.</li> <li>◦ arg2 : 숫자를 나타내는 구조적 표현식이다.</li> </ul>

함수	설명
@urlEncode(arg1)	UTF8 문자셋으로 url-encode를 한 문자열을 리턴한다. ◦ arg1 : url-encode를 적용할 문자열을 나타내는 구조적 표현식이다.
@getNull()	null 객체를 리턴한다.
@equals(arg1, arg2)	두 객체의 동등 여부를 비교한다. ◦ arg1, arg2 : 구조적 표현식이다.
@sequence(arg1, arg2, arg3)	시퀀스를 획득한다. ◦ arg1 : 시퀀스의 ID를 나타낸다. ◦ arg2 : 시퀀스 문자열의 길이를 나타낸다. ◦ arg3 : 시퀀스 문자열의 왼쪽을 자리수만큼 패딩할 것인지를 선택한다.

## 4.8. 사용자 코드와 핸들러

AnyLink 서비스 플로우에서는 사용자가 직접 Java 코드를 작성하여 액티비티를 실행하거나 서비스 플로우와 액티비티의 특정 시점 혹은 에러 상황에서 추가적인 코드를 실행시키는 기능을 제공한다.

서비스 플로우 엔진에서 사용자 코드를 사용할 수 있는 부분은 다음과 같다.

### • 사용자 클래스 액티비티

사용자가 작성한 Java 코드를 실행하는 액티비티.

사용자가 작성한 Java 코드는 com.tmax.anylink.api.serviceflow 패키지의 DefaultUserActivity 클래스를 상속해야 한다.

```
// 사용자 액티비티의 주 메소드
void action(ActivityContext ctx) throws AnyLinkException;
```

### • 프로세스 핸들러

서비스 플로우의 시작, 종료 등의 시점에서 호출되는 핸들러 클래스이다.

사용자가 작성한 Java 코드는 com.tmax.anylink.api.serviceflow 패키지의 DefaultProcessHandler 클래스를 상속해야 한다.

```
// 서비스 플로우 시작 시 호출
void started(ProcessContext ctx) throws AnyLinkException;

// 서비스 플로우 종료 시 호출
void finished(ProcessContext ctx) throws AnyLinkException;

// 내부적으로만 정의됨. 실제 호출되지는 않음
void paused(ProcessContext ctx) throws AnyLinkException;
```

```
// 내부적으로만 정의됨. 실제 호출되지는 않음
void resumed(ProcessContext ctx) throws AnyLinkException;
```

## • 프로세스 에러 핸들러

서비스 플로우 실행 도중에 처리하지 못하는 에러가 발생할 경우 호출되는 핸들러 클래스이다.

사용자가 작성한 Java 코드는 com.tmax.anylink.api.serviceflow 패키지의 DefaultProcessErrorHandler 클래스를 상속해야 한다.

```
// 에러 핸들러가 정의된 서비스 플로우 실행 도중 에러가 발생할 때 호출
void handle(ProcessContext context, Throwable error) throws AnyLinkException;
```

## • 액티비티 핸들러

액티비티의 시작, 종료, 취소 종료, 에러 종료 등의 시점에서 호출되는 핸들러 클래스이다.

사용자가 작성한 Java 코드는 com.tmax.anylink.api.serviceflow 패키지의 DefaultActivityHandler 클래스를 상속해야 한다.

```
// 핸들러가 정의된 액티비티의 시작 시점에 호출
void started(ActivityContext ctx) throws AnyLinkException;

// 핸들러가 정의된 액티비티의 완료 시점에 호출
void finished(ActivityContext ctx) throws AnyLinkException;

// 핸들러가 정의된 액티비티의 취소 완료 시점에 호출
void cancelled(ActivityContext ctx, String cause) throws AnyLinkException;

// 핸들러가 정의된 액티비티의 에러 완료 시점에 호출
void errorOccurred(ActivityContext ctx, Throwable t) throws AnyLinkException;
```

## • 액티비티 에러 핸들러

액티비티 실행 도중에 처리하지 못하는 에러가 발생할 경우 호출되는 핸들러 클래스이다.

사용자가 작성한 Java 코드는 com.tmax.anylink.api.serviceflow 패키지의 DefaultActivityErrorHandler 클래스를 상속해야 한다.

```
// 에러 핸들러가 정의된 액티비티 실행 도중 에러가 발생할 때 호출
void handle(ActivityContext context, Throwable error) throws AnyLinkException;
```

## • 에러 코드 매퍼

서비스 플로우에서는 에러 이벤트를 발생시키거나 잡을 때 에러 코드를 사용한다. 즉, End 이벤트로 작성된 에러 이벤트는 지정한 에러 코드의 에러 이벤트를 발생시키는 역할을 하고, 바운더리 에러 이벤트는 해당 액티비티 내에서 발생한 에러를 에러 코드에 따라 잡는 역할을 한다. 에러 코드 매퍼는 액티비티 수행 도중 발생하는 Java Exception 객체들을 에러 코드로 변환하는 기능을 수행한다. 에러 코드 매퍼 클래스는 서비스 플로우에서

지정할 수 있다.

사용자가 작성한 Java 코드는 com.tmax.anylink.api.serviceflow 패키지의 DefaultErrorCodeMapper 클래스를 상속해야 한다.

```
// 발생한 Exception을 보고 에러 코드를 판별하여주는 이 메소드를 구현해야 한다.  
String getErrorCode(Throwable throwable);
```

#### • 서비스 액티비티 핸들러

액티비티 핸들러의 기능에 더하여 서비스를 호출하기 전과 호출한 후의 시점에서 추가로 호출되는 핸들러 클래스이다.

사용자가 작성한 Java 코드는 com.tmax.anylink.api.serviceflow 패키지의 DefaultServiceActivityHandler 클래스를 상속해야 한다.

```
// 핸들러가 정의된 서비스 액티비티의 서비스 호출 직전에 호출  
void beforeServiceCall(ActivityContext ctx, MessageContext mctx) throws AnyLinkException;  
  
// 핸들러가 정의된 서비스 액티비티의 서비스 호출 직후에 호출  
void afterServiceCall(ActivityContext ctx, MessageContext mctx) throws AnyLinkException;
```

## 4.9. 코릴레이션

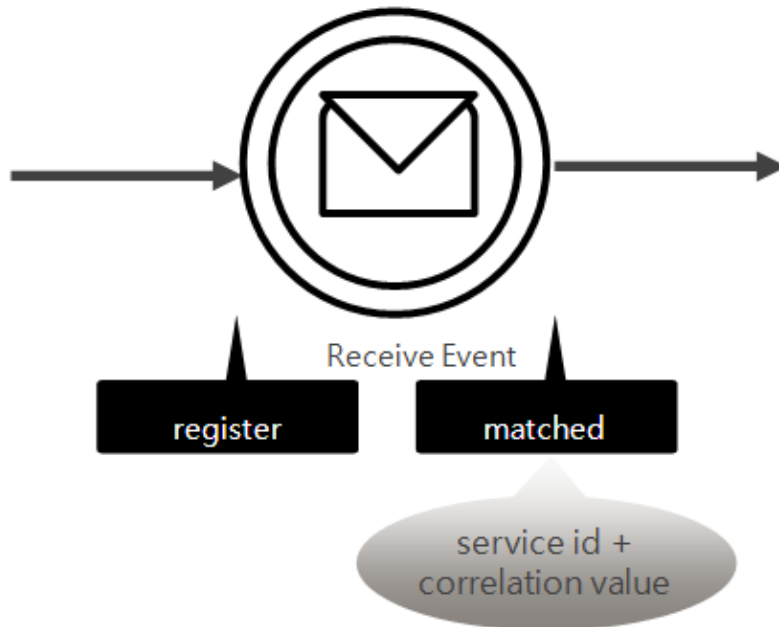
코릴레이션(Correlation)은 독립적인 개체들을 연관시키는 것을 의미한다.

AnyLink에서 코릴레이션은 TCP 어댑터 등에서 요청 메시지와 이에 대응하는 응답 메시지를 찾아주기 위해 사용하기도 한다. AnyLink 서비스 플로우 엔진에서 얘기하는 코릴레이션은 특정한 서비스를 나타내는 이벤트가 발생하였을 때 이미 실행 중인 서비스 플로우 인스턴스들 중에서 현재 발생한 이벤트와 관련있는 인스턴스를 찾아서 이벤트를 해당 서비스 플로우 인스턴스에게 전달해주는 일을 뜻하는 것으로 서비스 플로우 인스턴스와 발생 이벤트를 연관시키는 행위라고 볼 수 있다.

보통 서비스 플로우 엔진에서 이벤트가 발생하면 이벤트가 시작 이벤트(Start Event) 즉, 서비스 플로우의 처음 이벤트일 경우에는 서비스 플로우 인스턴스를 새로 생성하는 역할을 하게 된다.

코릴레이션을 사용하는 이벤트들은 주로 서비스 플로우가 실행 도중에 이벤트 발생을 대기하게 되는 중간 이벤트(Intermediate Event)들이다. 보통의 중간 이벤트일 수도 있고, 바운더리 이벤트일 수도 있다.

서비스 플로우 엔진에서 코릴레이션 이벤트는 기대하는 코릴레이션 값을 등록(registering correlation)하는 과정과 해당 서비스에 해당하는 서비스가 발생했을 때 일치하는 코릴레이션 이벤트 인스턴스를 찾는 매칭(matching) 과정 두 가지 절차를 거쳐 서비스 플로우 인스턴스와 서비스와의 코릴레이션을 수행하게 된다.



#### 서비스 코릴레이션의 두 가지 단계

서비스 플로우가 트랜지션을 통해 코릴레이션을 사용한 중간 이벤트에 도달하게 되면 AnyLink 서비스 플로우 엔진은 내부적으로 코릴레이션을 위한 값을 계산하여 서비스 ID와 함께 내부적으로 가지고 있는 코릴레이션 정보 테이블에 등록한다.

AnyLink 딜리버리 채널을 통해 서비스 플로우 엔진으로 메시지가 전달되면 서비스 플로우 엔진은 다음의 순서로 처리한다.

1. 해당 서비스가 서비스 플로우의 시작 메시지 이벤트에 해당하는 서비스인 경우에는 서비스 플로우 인스턴스를 새로 만들게 된다.
2. 해당 서비스가 코릴레이션이 필요한 메시지 이벤트에 해당하는 서비스인 경우에는 코릴레이션 매칭을 시도한다. 코릴레이션 값이 매칭되는 메시지 이벤트가 발견되면 해당 이벤트가 속한 서비스 플로우 인스턴스로 메시지를 전달하여 해당 플로우 인스턴스가 다음으로 트랜지션되도록 한다.
3. 코릴레이션 매칭에 실패하면 코릴레이션 없이 해당 서비스를 기다리고 있는 메시지 이벤트를 찾아서 해당 이벤트가 속한 서비스 플로우 인스턴스 메소드를 전달하여 해당 플로우 인스턴스가 다음으로 트랜지션되도록 한다.
4. 해당 서비스를 기다리는 메시지 이벤트들이 하나도 등록되어 있지 않은 경우에는 서비스 플로우 엔진 내부적으로 메시지 이벤트가 등록되기 전에 먼저 도달했다고 가정하고 조기 도달 메시지 맵(Early arrived message map)에 저장해둔다. 이벤트 등록보다 먼저 도달한 메시지를 기다려주는 시간은 기본적으로 3초이다.

AnyLink 서비스 플로우 엔진에서 코릴레이션 값의 매칭이 발생하는 경우는 서비스 ID(서비스 플로우 ID와 해당 메시지 이벤트 ID의 조합으로 구성)가 같고, 계산된 코릴레이션 값이 서비스 플로우 인스턴스가 등록한 코릴레이션 값과 같을 때이다.

이를 위해 서비스 플로우의 중간 이벤트들은 코릴레이션을 위해 등록 표현식(registration expression)과 매칭 표현식(matching expression)을 지정할 수 있다.

만약 서로 다른 서비스 플로우 혹은 메시지 이벤트들 간에 코릴레이션이 필요한 경우에는 멀티바인딩 라우터를 사용할 수가 있다. 멀티바인딩 라우터의 라우팅 메소드를 서비스 플로우 코릴레이션으로 선택하면 서로 다른 서비스를 사용하여 코릴레이션 값 매칭을 할 수가 있다.

## 5. 멀티바인딩 라우터

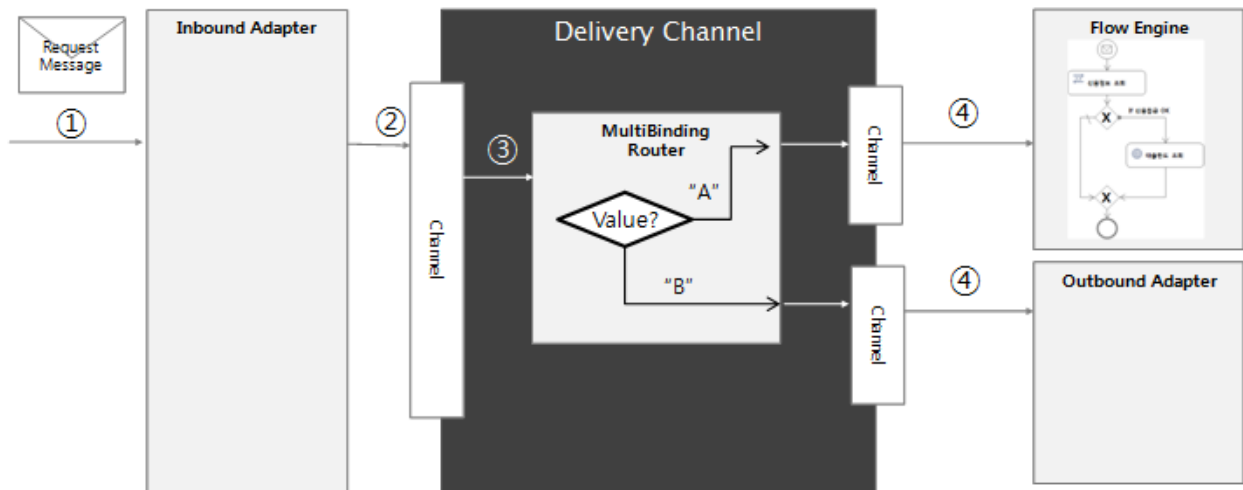
본 절에서는 멀티바인딩 라우터에서 서비스 분기방법과 룰에 대해서 설명한다.

### 5.1. 개요

AnyLink의 서비스들을 여러 가지 조건에 따라 선택적으로 분기하거나 멀티캐스트할 수 있는 기능을 제공하는 컴포넌트가 멀티바인딩 라우터이다.

멀티바인딩 라우터가 분기 혹은 멀티캐스트할 수 있는 서비스들은 AnyLink 내부 서비스들 즉, 서비스 플로우의 Receive 메시지 이벤트들, 어댑터의 아웃바운드 룰들, 그리고 또다른 멀티바인딩 라우터 룰 등을 모두 포함한다.

멀티바인딩 라우터 룰은 그 자체로서 하나의 AnyLink의 서비스로 간주된다. 즉, 어댑터의 파싱 룰을 통해서 멀티바인딩 룰을 호출할 수 있으며, 서비스 플로우의 서비스 액티비티들 중 멀티바인딩 액티비티, 또다른 멀티바인딩 룰을 통해서도 호출할 수 있다.



멀티바인딩 라우터 룰에 따른 서비스 분기

### 5.2. 멀티바인딩 룰

멀티바인딩 라우터의 룰은 다음과 같은 요소를 기본으로 구성된다.

- 멀티바인딩 룰을 서비스로 노출하는 요청 메시지, 응답 메시지, 비정상 응답 메시지
- 라우팅 규칙에 따라 호출할 대상 서비스 정보들
- 라우팅 메소드와 서비스별 파라미터들

com.tmax.anylink.api.multibinding.DefaultRoutingHandler 클래스는 다음과 같은 메소드를 선언하고 있다.

```
String route(MessageContext ctx) throws AnyLinkException;
```



멀티바인딩 라우터에서 사용되는 라우팅 메소드와 파라미터의 의미는 다음과 같다.

라우팅 메소드	설명
Value	사용자가 변수를 정의하고 해당 변수의 값에 따라 라우팅한다.
RoundRobin	라운드로빈(round-robin) 방식으로 각 서비스들을 순차적으로 라우팅한다.
WeightBased	가중치를 고려한 라운드 로빈 라우팅이다.
Multicast	one-way 방식의 서비스일 때에만 사용 가능하며, 등록된 모든 엔트리로 전송한다.
TimeRange	지정된 시간대에 해당하는 서비스를 호출한다.
FlowCorrelation	서비스 플로우의 코릴레이션 방식이다.  값 매칭을 사용하여 가장 먼저 일치하는 서비스 항목으로 라우팅한다. 이때는 대상 서비스가 코릴레이션을 지원하는 서비스 플로우 메시지 이벤트여야 한다.
Handler	사용자 핸들러 클래스이다.



메소드에 대한 상세설명은 해당 절을 참고한다.

### 5.2.1. Value

사용자가 변수를 정의하고 해당 변수의 값에 따라 라우팅을 하는 메소드로 표현식을 계산한 값과 일치하는 서비스 항목으로 라우팅을 한다. 요청 메시지를 표현식에서 input이라는 변수로 사용할 수 있다.

#### ◦ 형식

```
$input.field1 (변수명)
```

#### ◦ 예

```
1, 2, svc1
```

### 5.2.2. WeightBased

가중치를 고려한 라운드 로빈 라우팅이다.

#### ◦ 형식

```
가중치. 정수값
```

### 5.2.3. TimeRange

지정된 시간대에 해당하는 서비스를 호출. 맞는 시간대의 서비스 항목이 없으면 default로 정의된 서비스 항목으로 라우팅한다.

- 형식

```
HH:MM - HH:MM
```

뒤의 시간이 앞의 시간보다 클 경우에는 현재 시간이 그 가운데에 해당하면 라우팅하며, 앞의 시간이 뒤의 시간보다 클 경우에는 현재 시간이 뒤의 시간보다 작거나 앞의 시간보다 클 경우에 라우팅한다.

- 예

```
09:30-15:10 혹은 default
```

### 5.2.4. Handler

사용자 핸들러 클래스이다. 핸들러는 DefaultRoutingHandler 클래스를 상속하여 구현한 사용자 핸들러 클래스를 호출한 결과값과 각 서비스 항목의 값을 비교하여 일치하는 서비스 항목으로 라우팅한다.

- 형식

```
telco.binding.TaxRRBinding (클래스명)
```

- 예

```
a, b
```

# 부록 A: 서버 API

본 부록에서는 AnyLink 사용자 API에 대해서 설명한다. 사용자 코드를 사용하는 표준 API들 중 서비스 플로우와 멀티바인딩 라우터에 관련된 API들 중심으로 기술한다.



설명된 인터페이스들은 대부분 Default로 시작하는 기본 구현 사용자 클래스와 함께 정의되어 있다. 인터페이스를 직접 구현하기보다는 Default 추상 클래스를 상속하여 구현할 것을 권장한다.

## A.1. com.tmax.anylink.api 패키지

여러 패키지에서 공통으로 사용하는 MessageContext 인터페이스가 정의되어 있다.

<MessageContext>

```
package com.tmax.anylink.api;

/**
 * 메시지를 나타내는 표준 인터페이스
 */
public interface MessageContext {
    /**
     * @return 메시지의 내용 데이터를 리턴한다
     */
    Object getContent();

    /**
     * @return 메시지의 속성 값을 리턴한다
     */
    Object getProperty(String propertyName);

    /**
     * 메시지의 속성 값을 지정한다.
     *
     * @param propertyName 속성 이름
     * @param propertyValue 속성 값
     */
    void setProperty(String propertyName, Object propertyValue);
}
```

## A.2. com.tmax.anylink.api.serviceflow 패키지

서비스 플로우의 프로세스나 액티비티에서 사용하는 사용자 클래스, 핸들러 등에 해당하는 인터페이스와 기본 구현 클래스가 정의되어 있다. 예를 들어 사용자 액티비티에 해당하는 인터페이스는 UserActivity 인터페이스에 정의되어 있고, 이를 구현하는 기본 클래스는 DefaultUserActivity 추상 클래스로 정의되어 있다.

인터페이스	설명
<a href="#">ActivityContext</a>	액티비티의 실행 문맥을 나타낸다.
<a href="#">ActivityErrorHandler</a>	액티비티에서 에러가 발생할 때 이를 처리할 수 있는 핸들러 인터페이스이다.
<a href="#">ActivityHandler</a>	AnyLink 액티비티 핸들러 인터페이스이다.
<a href="#">BlockContext</a>	현재 수행되는 액티비티를 둘러싸고 있는 블록 액티비티의 실행 문맥을 나타낸다.
<a href="#">ErrorCodeMapper</a>	내부적으로 발생한 에러를 AnyLink 서비스 플로우가 이해하는 에러 코드로 변환해준다.
<a href="#">UserMapping</a>	AnyLink 서비스 플로우 내에서 사용자가 직접 데이터 변환 코드를 작성할 수 있게 해주는 인터페이스이다.
<a href="#">ProcessContext</a>	현재 실행되는 서비스 플로우의 실행 문맥을 나타낸다.
<a href="#">ProcessHandler</a>	AnyLink 프로세스 핸들러 인터페이스이다.
<a href="#">ServiceActivityHandler</a>	AnyLink 서비스 액티비티 핸들러 인터페이스이다.
<a href="#">UserActivity</a>	사용자 액티비티를 표현하는 인터페이스이다.
<a href="#">VariableContext</a>	서비스 플로우의 변수에 해당하는 인터페이스를 정의한다.



사용자들은 기본 추상 클래스를 상속하여 클래스를 작성할 것을 권고한다.

## A.2.1. ActivityContext

액티비티의 실행 문맥을 나타낸다.

```
package com.tmax.anylink.api.serviceflow;

/**
 * 액티비티의 실행 문맥을 나타내는 인터페이스
 */

public interface ActivityContext {
    /**
     * @return 액티비티 ID를 리턴한다.
     */
    String getActivityId();

    /**
     * @return 이 액티비티 scope에서 보이는 지정된 이름의 변수 문맥을 리턴한다.
     */
    VariableContext getVariable(String variableName);

    /**
     * @return 이 액티비티가 포함된 서비스 플로우의 문맥을 리턴한다.
     */
    ProcessContext getProcessContext();

    /**
     * @return 이 액티비티를 둘러싼 블록 액티비티의 문맥을 리턴한다.
     */
}
```

```

    * 블록 액티비티가 돌려싸고 있지 않다면 null을 리턴한다.
    */
    BlockContext getBlockContext();
}

```

## A.2.2. ActivityErrorHandler

액티비티에서 에러가 발생할 때 이를 처리할 수 있는 핸들러 인터페이스이다.

```

package com.tmax.anylink.api.serviceflow;

import com.tmax.anylink.common.AnyLinkException;

/**
 * 액티비티에서 에러가 발생할 때 이를 처리할 수 있는 핸들러 인터페이스
 */

public interface ActivityErrorHandler {
    /**
     * 핸들러가 정의된 액티비티 실행 도중 어떤 에러가 발생하면 호출되는 메소드
     */
    void handle(ActivityContext context, Throwable error) throws AnyLinkException;
}

```

## A.2.3. ActivityHandler

AnyLink 액티비티 핸들러 인터페이스이다.

```

package com.tmax.anylink.api.serviceflow;

import com.tmax.anylink.common.AnyLinkException;

/**
 * AnyLink 액티비티 핸들러 인터페이스.
 * 이 인터페이스를 구현한 객체를 해당 액티비티의 핸들러로 등록하면
 * 액티비티 인스턴스의 라이프사이클 시점에 맞추어 대응하는 메소드들이 각각 호출된다.
 */

public interface ActivityHandler {
    /**
     * 액티비티 시작 시점에서 호출되는 메소드
     */
    void started(ActivityContext ctx) throws AnyLinkException;

    /**
     * 액티비티 정상 종료 시점에서 호출되는 메소드
     */
    void finished(ActivityContext ctx) throws AnyLinkException;

    /**
     * 액티비티가 종료되지 않고 취소될 때 호출되는 메소드.
     * 액티비티는 몇 가지 이유로 취소될 수 있는데 주로 서비스 플로우 내의

```

```

    * 다른 액티비티나 이벤트에 의해 취소된다.
    */
    void cancelled(ActivityContext ctx, String cause) throws AnyLinkException;

    /**
     * 액티비티가 어떤 에러에 의해 종료될 때 호출되는 메소드
     */
    void errorOccurred(ActivityContext ctx, Throwable t) throws AnyLinkException;
}

```

## A.2.4. BlockContext

현재 수행되는 액티비티를 둘러싸고 있는 블록 액티비티의 실행 문맥을 나타 낸다.

```

package com.tmax.anylink.api.serviceflow;

/**
 * 현재 수행되는 액티비티를 둘러싸고 있는 블록 액티비티의 실행 문맥을 나타내는 인터페이스
 */

public interface BlockContext {
    /**
     * @return 이 블록 scope에서 보이는 지정된 이름의 변수 문맥을 리턴한다.
     */
    VariableContext getVariable(String variableName);

    /**
     * @return 이 블록 액티비티를 둘러싸고 있는 부모 블록 액티비티의 실행 문맥을 리턴한다.
     * 둘러싸고 있는 부모 블록 액티비티가 없다면 null을 리턴한다.
     */
    BlockContext getParentBlockContext();

    /**
     * @return 이 블록 액티비티가 포함된 서비스 플로우의 문맥을 리턴한다.
     */
    ProcessContext getProcessContext();
}

```

## A.2.5. ErrorCodeMapper

내부적으로 발생한 에러를 AnyLink 서비스 플로우가 이해하는 에러 코드로 변환해준다.

```

package com.tmax.anylink.api.serviceflow;

/**
 * 내부적으로 발생한 에러를 AnyLink 서비스 플로우가 이해하는
 * 에러 코드로 변환해주기 위한 인터페이스이다.
 * AnyLink 서비스 플로우는 모든 예외를 잡으려고 하지 않을 경우에는
 * 에러를 처리하기 위해 에러 코드를 알 필요가 있다.
 * 이 인터페이스는 발생한 자바 객체를 서비스 플로우의 에러 이벤트가 잡을 수 있도록
 * 에러 코드로 변환해주는 역할을 한다.
 */

```

```
public interface ErrorCodeMapper {
    /**
     * @param throwable 발생한 자바 예외 객체
     * @return 서비스 플로우의 에러 이벤트가 처리할 에러 코드값
     */
    String getErrorCode(Throwable throwable);
}
```

## A.2.6. UserMapping

AnyLink 서비스 플로우 내에서 사용자가 직접 데이터 변환 코드를 작성할 수 있게 해주는 인터페이스이다.

```
package com.tmax.anylink.api.serviceflow;

/**
 * AnyLink 서비스 플로우 내에서 사용자가 직접 데이터 값을
 * 변환하는 코드를 작성할 수 있게 해주는 인터페이스이다.
 * 요청 매핑, 응답 매핑, 비정상 응답 매핑, 커스텀 로그 매핑 등에서
 * 사용자가 직접 작성한 코드를 사용하여 매핑하고자 할 때 사용한다.
 */

public interface UserMapping {
    /**
     * 사용자 작성 매핑 메소드
     *
     * @param ctx 현재 액티비티 컨텍스트
     * @param inputParams 입력 파라미터들이 이 변수를 통해 전달된다
     * @return 매핑한 결과 데이터. 결과 파라미터 목록 갯수와 배열 크기가 같아야 한다.
     */
    Object[] mapping(ActivityContext ctx, Object[] inputParams) throws AnyLinkException;
}
```

## A.2.7. ProcessContext

현재 실행되는 서비스 플로우의 실행 문맥을 나타낸다.

```
package com.tmax.anylink.api.serviceflow;

import com.tmax.anylink.logging.Logger;

/**
 * 현재 실행되는 서비스 플로우의 실행 문맥을 나타내는 인터페이스
 */

public interface ProcessContext {
    /**
     * @return 서비스 플로우 ID를 리턴한다.
     */
    String getProcessId();

    /**

```

```

    * @return 서비스 플로우의 인스턴스 ID를 리턴한다.
    */
    String getProcessInstanceId();

    /**
     * @return 시스템 로거 객체를 리턴한다.
     */
    Logger getUserLogger();

    /**
     * @return 이 서비스 플로우의 scope에서 지정된 이름의 변수 정보를 리턴한다.
     */
    VariableContext getVariable(String variableName);

    /**
     * AnyLink 실행 환경 변수 값을 리턴한다.
     * server.name, cluster.name, bizsystem.id 등이 현재 지원되는 환경 변수 키이다.
     *
     * @param key 환경변수 키
     * @return 키에 해당하는 환경변수 값
     */
    String getenv(String key);
}

```

## A.2.8. ProcessHandler

AnyLink 프로세스 핸들러 인터페이스이다.

```

package com.tmax.anylink.api.serviceflow;

import com.tmax.anylink.common.AnyLinkException;

/**
 * AnyLink 프로세스 핸들러 인터페이스.
 * 이 인터페이스를 구현한 객체를 해당 서비스 플로우의 핸들러로 등록하면
 * 서비스 플로우의 각 라이프사이클 시점에 맞추어 대응하는 메소드들이 각각 호출된다.
 */

public interface ProcessHandler {
    /**
     * 서비스 플로우가 시작할 때 호출되는 메소드
     */
    void started(ProcessContext ctx) throws AnyLinkException;

    /**
     * 서비스 플로우가 종료할 때 호출되는 메소드
     */
    void finished(ProcessContext ctx) throws AnyLinkException;

    /**
     * 서비스 플로우가 일시 중지될 때 호출되는 메소드. 이 메소드는 현재 사용되지 않는다.
     */
    void paused(ProcessContext ctx) throws AnyLinkException;

    /**
     * 일시 중지된 서비스 플로우가 재개될 때 호출되는 메소드. 이 메소드는 현재 사용되지 않는다.
     */
}

```



```

    */
    void resumed(ProcessContext ctx) throws AnyLinkException;
}

```

## A.2.9. ServiceActivityHandler

AnyLink 서비스 액티비티 핸들러 인터페이스이다.

```

package com.tmax.anylink.api.serviceflow;

import com.tmax.anylink.api.MessageContext;
import com.tmax.anylink.common.AnyLinkException;

/**
 * AnyLink 서비스 액티비티 핸들러 인터페이스.
 * 이 인터페이스는 ActivityHandler 인터페이스에 더하여
 * 서비스 호출의 각 시점에 해당하는 추가적인 메소드들을 정의한다.
 */

public interface ServiceActivityHandler extends ActivityHandler {
    /**
     * 해당 서비스를 호출하기 직전에 호출되는 메소드
     */
    void beforeServiceCall(ActivityContext ctx, MessageContext context) throws AnyLinkException;

    /**
     * 해당 서비스가 호출된 직후에 호출되는 메소드
     */
    void afterServiceCall(ActivityContext ctx, MessageContext context) throws AnyLinkException;
}

```

## A.2.10. UserActivity

사용자 액티비티를 표현하는 인터페이스이다.

```

package com.tmax.anylink.api.serviceflow;

import com.tmax.anylink.common.AnyLinkException;

/**
 * 사용자 액티비티를 표현하는 인터페이스.
 * 사용자 클래스 액티비티를 만들려면 사용자들은 이 인터페이스를 구현한
 * DefaultUserActivity 추상 클래스를 상속하는 클래스를 작성하여 지정하면 된다.
 */

public interface UserActivity {
    /**
     * 사용자 액티비티 클래스가 구현해야 할 주 메소드
     */
    void action(ActivityContext ctx) throws AnyLinkException;
}

```

## A.2.11. VariableContext

서비스 플로우의 변수에 해당하는 인터페이스를 정의한다.

```
package com.tmax.anylink.api.serviceflow;

import com.tmax.anylink.api.MessageContext;

/**
 * 서비스 플로우의 변수에 해당하는 인터페이스를 정의한다.
 */

public interface VariableContext extends MessageContext {
    /**
     * 변수 데이터를 지정한다
     *
     * @param content 변수에 지정할 데이터
     */
    void setContent(Object content);
}
```

## A.3. com.tmax.anylink.api.multibinding 패키지

멀티바인딩 라우터에서 사용하는 핸들러 인터페이스들이 정의되어 있다.

### RoutingHandler

```
package com.tmax.anylink.api.multibinding;

import com.tmax.anylink.api.MessageContext;
import com.tmax.anylink.common.AnyLinkException;

/**
 * 멀티바인딩 라우터의 핸들러 인터페이스.
 * 이 핸들러 클래스는 멀티바인딩 룰의 라우팅 메소드를 "Handler"로 지정했을 때 사용할 수 있다.
 */

public interface RoutingHandler {
    /**
     * @return 라우팅 엔트리 이름을 리턴한다. 엔트리 이름은 라우터 룰의
     * 각 엔트리 값들 중 적어도 하나와 일치해야 처리가 가능하다
     */
    String route(MessageContext ctx) throws AnyLinkException;
}
```

## A.4. Default 추상 클래스 목록

앞의 각 인터페이스들을 구현하고 있는 추상 클래스 목록이다.

- `com.tmax.anylink.api.serviceflow.DefaultUserActivity`
- `com.tmax.anylink.api.serviceflow.DefaultActivityHandler`
- `com.tmax.anylink.api.serviceflow.DefaultProcessHandler`
- `com.tmax.anylink.api.serviceflow.DefaultActivityErrorHandler`
- `com.tmax.anylink.api.serviceflow.DefaultProcessErrorHandler`
- `com.tmax.anylink.api.serviceflow.DefaultErrorCodeMapper`
- `com.tmax.anylink.api.multibinding.DefaultRoutingHandler`



가능하면 인터페이스를 직접 구현하기보다는 추후 확장성 등을 감안하여 기본 추상 클래스를 상속하여 구현할 것을 권고한다.