HP Universal CMDB

소프트웨어 버전: 10.20

개발자 참조 안내서

문서 릴리스 날짜: 2015년 1월
소프트웨어 릴리스 날짜: 2015년 1월

법적 고지 사항

보증

HP 제품 및 서비스에 대한 모든 보증 사항은 해당 제품 및 서비스와 함께 제공된 명시적 보증서에 규정되어 있습니다. 여기에 수록된 어떤 내용도 추가 보증을 구성하는 것으로 해석될 수 없습니다. HP는 여기에 수록된 기술적 또는 편집상의 오류나 누락에 대해 책임지지 않습니다.

여기에 수록된 정보는 통지 없이 변경될 수 있습니다.

제한된 권한 범례

기밀 컴퓨터 소프트웨어. 보유, 사용 또는 복사에 필요한 HP에서 제공한 유효한 라이센스. FAR 12.211 및 12.212에 의거하여 상용 컴퓨터 소프트웨어, 컴퓨터 소프트웨어 문서 및 상용 품목에 대한 기술 데이터는 벤더의 표준 상용 라이센스 하에서 미국 정부에 사용이 허가되었습니다.

저작권 고지

상표 고지 사항

Adobe ® 는 Adobe Systems Incorporated의 상표입니다.

Microsoft ® 및 Windows ® 는 Microsoft Corporation의 미국 등록 상표입니다.

Oracle 및 Java는 Oracle 및/또는 계열사의 등록 상표입니다.

UNIX ® 는 The Open Group의 등록 상표입니다.

Linux ® 는 미국 및 기타 국가에서 Linus Torvalds의 등록 상표입니다.

오픈 소스 및 타사 확인의 전체 목록을 보려면 HP Software Support Online 웹 사이트를 방문하여 HP Service Manager Open Source and Third Party License Agreements라는 제품 설명서를 검색합니다.

문서 업데이트

이 문서의 제목 페이지에는 다음 식별 정보가 포함됩니다.

- 소프트웨어 버전 번호 - 소프트웨어 버전을 나타냅니다.
- 문서 릴리스 날짜 - 문서가 업데이트될 때마다 변경됩니다.
- 소프트웨어 릴리스 날짜 - 이 소프트웨어 버전의 릴리스 날짜를 나타냅니다.

최근 업데이트를 확인하거나 문서의 최신 버전을 사용하고 있는지 확인하려면 다음 웹 사이트를 방문하십시오. **https://softwaresupport.hp.com**

이 사이트를 사용하려면 HP Passport 사용자로 등록하여 로그인해야 합니다. HP Passport ID를 등록하려면 다음 웹 사이트를 방문하십시오. **https://hpp12.passport.hp.com/hppcf/createuser.do**

또는 HP Software 지원 페이지의 맨 위에서 **Register** 링크를 클릭합니다.

적절한 제품 지원 서비스에 가입할 경우 업데이트 버전이나 새 버전도 제공됩니다. 자세한 내용은 HP 판매 담당자에게 문의하십시오.

지원

다음 위치에서 HP Software 지원 온라인 웹 사이트를 방문하십시오. **https://softwaresupport.hp.com**

이 웹 사이트에서는 연락처 정보를 비롯하여 HP 소프트웨어에서 제공하는 제품, 서비스 및 지원에 대한 자세한 내용을 확인할 수 있습니다.

온라인 지원을 통해 사용자가 스스로 문제를 해결할 수 있습니다. 또한 업무 관리에 필요한 대화식 기술 지원 도구에 신속하고 효율적으로 액세스할 수 있습니다. 소중한 지원 고객으로서 지원 웹 사이트를 통해 다음과 같은 혜택을 누릴 수 있습니다.

- 관심 있는 지식 문서를 검색할 수 있습니다.
- 지원 사례 및 개선 요청을 제출하고 추적할 수 있습니다.
- 소프트웨어 패치를 다운로드할 수 있습니다.
- 지원 계약을 관리할 수 있습니다.
- HP 고객지원센터 연락처를 조회할 수 있습니다.
- 사용 가능한 서비스에 대한 정보를 검토할 수 있습니다.
- 다른 소프트웨어 고객과의 토론에 참여할 수 있습니다.
- 소프트웨어 교육을 조사하고 등록할 수 있습니다.

대부분의 지원 영역을 이용하려면 HP Passport 사용자로 등록하여 로그인해야 합니다. 이 영역에서는 지원 계약이 필요할 수도 있습니다. HP Passport ID를 등록하려면 다음 위치로 이동하십시오.

https://hpp12.passport.hp.com/hppcf/createuser.do

액세스 수준에 대한 자세한 내용을 보려면 다음 위치로 이동하십시오.

https://softwaresupport.hp.com/web/softwaresupport/access-levels

HP Software Solutions Now에서는 HPSW 솔루션 및 통합 포털 웹 사이트에 액세스합니다. 이 사이트에서는 비즈니스 요구 사항에 맞는 HP 제품 솔루션을 탐색할 수 있고 HP 제품 간의 전체 통합 목록 및 ITIL 프로세스 목록을 제공합니다. 이 웹 사이트의 URL은 **http://h20230.www2.hp.com/sc/solutions/index.jsp**입니다.

온라인 도움말의 이 PDF 버전 정보

이 문서는 온라인 도움말의 PDF 버전입니다. 이 PDF 파일은 도움말 정보에서 여러 항목을 손쉽게 인쇄하거나 PDF 형식으로 온라인 도움말을 읽을 수 있도록 제공됩니다. 이 컨텐츠는 원래 웹 브라우저에서 온라인 도움말로 보도록 작성되었으므로 일부 항목의 형식이 제대로 표시되지 않을 수 있습니다. 일부 대화형 항목이 PDF 버전에는 없을 수 있습니다. 이러한 항목은 온라인 도움말에서 정상적으로 인쇄할 수 있습니다.

목차

I부: 디스커버리 및 통합 어댑터 만들기

1장: 어댑터 개발 및 작성

이 장의 내용:

어댑터 개발 및 작성 개요

새 어댑터 개발에 대한 실제 계획을 시작하기에 앞서 일반적으로 이 개발과 연관된 프로세스 및 상호 작용을 이해해야 합니다.

다음 섹션에서는 디스커버리 개발 프로젝트를 성공적으로 관리하고 실행하기 위해 알아야 할 사항과 준비해야 할 사항을 파악하는 데 도움이 되는 정보를 제공합니다.

이 장 :

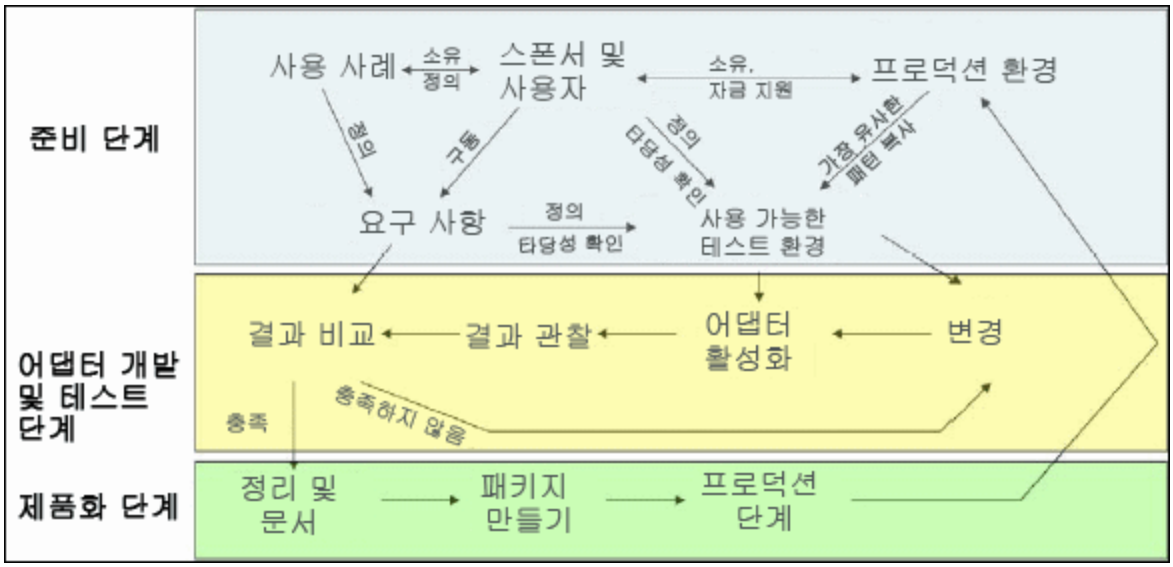
- HP Universal CMDB에 대한 실무 지식을 갖추고 있으며 시스템을 구성하는 요소에 어느 정도 익숙하다고 간주하고 설명합니다. 개발자의 학습 과정을 돕기 위한 안내서이므로 완벽한 지침을 제공하지는 않습니다.
- HP Universal CMDB의 새 디스커버리 컨텐츠에 대한 계획, 조사, 구현 단계와 지침 및 고려 사항을 다룹니다.
- 데이터 흐름 관리 프레임워크의 주요 API에 대한 정보를 제공합니다. 사용 가능한 API에 관한 전체 문서는 *HP Universal CMDB 데이터 흐름 관리 API 참조*를 참조하십시오. 다른 비공식 API도 있지만 이러한 API는 기본 어댑터에 사용하더라도 변경될 수 있습니다.

컨텐츠 만들기

이 섹션의 내용:

어댑터 개발 주기

다음 그림은 어댑터 작성을 위한 순서도를 보여 줍니다. 개발 및 테스트가 반복적으로 발생하는 중간 섹션에서 대부분의 시간이 소요됩니다.

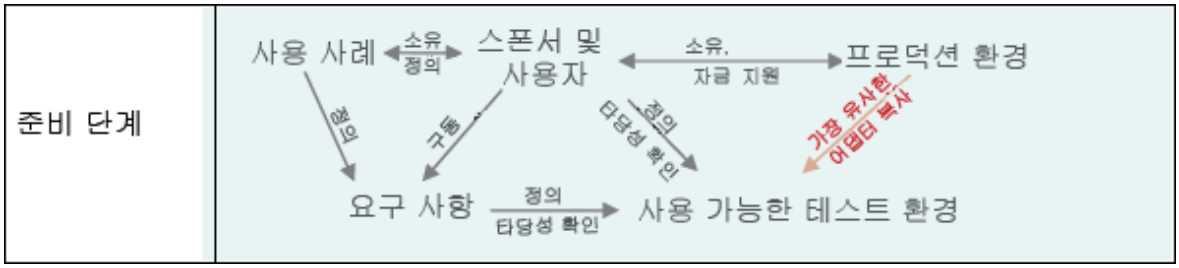


어댑터 개발의 각 단계가 마지막 단계에 빌드됩니다.

어댑터 모양 및 작동 방식이 마음에 들면 패키지화할 준비가 된 것입니다. UCMDB 패키지 관리자를 사용하거나 수동으로 구성 요소를 내보내어 패키지 *.zip 파일을 만듭니다. 생산을 위해 릴리스하기 전에, 모범 사례로서 모든 구성 요소를 고려하고 성공적으로 패키지화했는지 확인하기 위해 다른 UCMDB 시스템에 이 패키지를 배포하고 테스트해야 합니다. 패키지화에 대한 자세한 내용은 *HP Universal CMDB 관리 안내서*에서 패키지 관리자를 참조하십시오.

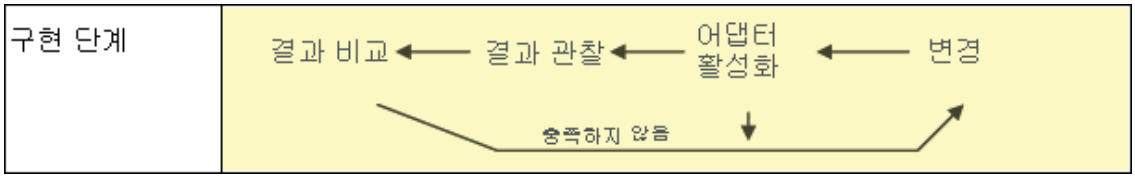
다음 섹션에서는 각 단계를 자세히 살펴보며 가장 중요한 단계와 모범 사례를 알아보겠습니다.

조사 및 준비 단계



조사 및 준비 단계는 개발을 추진하게 된 비즈니스 요구와 사용 사례를 포괄적으로 다루고, 어댑터를 개발하고 테스트하는 데 필요한 설비 확보에 대해서도 설명합니다.

1. **기존 어댑터를 수정하려는 경우 기술적으로 가장 먼저 수행할 단계는 해당 어댑터를 백업하여 원상태로 되돌릴 수 있도록 하는 것입니다. 새 어댑터를 만들려는 경우 가장 유사한 어댑터를 복사하여 적절한 이름으로 저장합니다. 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 리소스 창을 참조하십시오.**
2. **어댑터에서 데이터를 수집하는 데 사용할 방법을 조사합니다.**
   - **외부 도구/프로토콜을 사용하여 데이터 가져오기**
   - **어댑터에서 데이터를 기반으로 CI를 만드는 방법 개발**
   - **이제 유사한 어댑터 모양 파악**
3. **다음 기준에 따라 가장 비슷한 어댑터를 결정합니다.**
   - **만들어진 동일한 CI**
   - **사용된 동일한 프로토콜(SNMP)**
   - **동일한 종류의 대상(OS 유형, 버전 등)**
4. **전체 패키지를 복사합니다.**
5. **작업 영역에 패키지 컨텐츠의 압축을 풀고 어댑터(XML) 및 Jython(.py) 파일의 이름을 바꿉니다.**



어댑터 개발 및 테스트

어댑터 개발 및 테스트 단계는 반복이 매우 잦은 프로세스입니다. 어댑터가 형태를 갖추기 시작하면 최종 사용 사례에 대해 테스트를 시작하고 변경한 후 다시 테스트하며, 어댑터가 요구 사항을 충족할 때까지 이 프로세스를 반복합니다.

시작 및 사본 준비

- 어댑터의 XML 부분, 즉 1행의 이름(id), 만든 CI 유형, 호출된 Jython 스크립트 이름을 수정합니다.
- 원래 어댑터와 동일한 결과로 실행 중인 사본을 가져옵니다.
- 대부분의 코드, 특히 심각한 결과를 생성하는 코드를 주석 처리합니다.

개발 및 테스트

- 다른 샘플 코드를 사용하여 변경 내용을 개발합니다.
- 어댑터를 실행하여 테스트합니다.
- 복잡한 결과의 유효성을 검사하려면 전용 보기를 사용하고, 간단한 결과의 유효성을 검사하려면 검색을 수행합니다.

어댑터 패키지화 및 제품화

어댑터 패키지화 및 제품화 단계는 개발 마지막 단계에 해당합니다. 패키지화 단계로 넘어가기 전에, 모범 사례로서 나머지 디버깅 항목, 문서 및 주석을 정리하고 보안 고려 사항을 검토하는 등의 작업을 위해 최종 통과 절차를 거쳐야 합니다. 어댑터의 내부 작동을 설명할 수 있도록 최소한 추가 정보 문서는 항상 가지고 있어야 합니다. 누군가(개발자 자신도 포함)가 앞으로 이 어댑터를 살펴보아야 할 수 있으며, 이 경우 가장 간단한 문서조차 큰 도움이 될 것입니다.

정리 및 문서

- 디버깅 제거
- 주 섹션에서 모든 함수를 주석 처리하고 일부 시작 주석 추가
- 사용자가 테스트할 수 있는 샘플 TQL 및 보기 만들기

패키지 만들기

- 패키지 관리자를 사용하여 어댑터, TQL 등을 내보냅니다. 자세한 내용은 *HP Universal CMDB 관리 안내서*에서 패키지 관리자를 참조하십시오.
- 패키지가 다른 패키지에 대해 갖고 있는 종속 관계를 확인합니다. 예를 들어 해당 패키지로 만든 CI가 어댑터에 대한 입력 CI인지 확인합니다.
- 패키지 관리자를 사용하여 패키지 zip를 만듭니다. 자세한 내용은 *HP Universal CMDB 관리 안내서*에서 패키지 관리자를 참조하십시오.
- 새 컨텐츠의 일부를 제거하고 재배포하거나 다른 테스트 시스템에 배포하여 배포를 테스트합니다.

데이터 흐름 관리 및 통합

DFM 어댑터는 다른 제품과 통합할 수 있습니다. 다음 정의를 고려하십시오.

- DFM은 여러 대상으로부터 특정 컨텐츠를 수집합니다.
- 통합 과정은 한 시스템으로부터 다중 유형의 컨텐츠를 수집합니다.

이러한 정의는 컬렉션의 메서드를 구별하지 않으며 DFM도 마찬가지입니다. 새 어댑터를 개발하는 프로세스는 새 통합을 개발하는 프로세스와 같습니다. 동일한 조사를 실시하고 새로운 어댑터와 기존 어댑터에 대해 동일하게 선택한 후 동일한 방식으로 어댑터를 작성하는 등의 프로세스입니다. 바뀌는 점은 다음 몇 가지뿐입니다.

- 최종 어댑터의 일정을 정합니다. 통합 어댑터를 디스커버리보다 자주 실행할 수도 있지만 사용 사례에 따라 다릅니다.
- 입력 CI:
  - 통합: 입력하지 않고 실행할 CI가 아닌 트리거: 파일 이름 또는 원본이 어댑터 매개 변수를 통해 전달됩니다.
  - 디스커버리: 입력을 위해 일반 CMDB CI를 사용합니다.

통합 프로젝트의 경우 거의 항상 기존 어댑터를 다시 사용해야 합니다. 통합 방향(HP Universal CMDB에서 다른 제품으로 또는 다른 제품에서 HP Universal CMDB로)에 따라 개발 방법이 달라질 수 있습니다. 입증된 기술을 이용하여 자체적으로 사용할 수 있도록 복사 가능한 필드 패키지가 있습니다.

HP Universal CMDB에서 다른 프로젝트로 통합하려면 다음을 수행합니다.

- 내보낼 관계 및 CI를 생성하는 TQL을 만듭니다.
- 일반 래퍼 어댑터를 사용하여 TQL을 실행하고 외부 제품이 읽을 수 있도록 XML 파일에 결과를 작성합니다.

> **참고:** 필드 패키지에 대한 예는 HP 소프트웨어 지원에 문의하십시오.

HP Universal CMDB에 다른 제품을 통합하는 경우 다른 제품에서 데이터를 표시하는 방법에 따라 통합 어댑터가 다르게 작동합니다.

통합 유형	다시 사용할 참조의 예
제품의 데이터베이스에 직접 액세스	HP ED
내보내기를 통해 생성한 csv 또는 xml 파일에서 읽기	HP ServiceCenter
제품의 API에 액세스	BMC Atrium/Remedy

디스커버리 개발과 비즈니스 가치 연관

새 디스커버리 컨텐츠 개발에 관한 사용 사례는 비즈니스 사례를 통해 추진해야 하고 비즈니스 가치를 창출하도록 계획해야 합니다. 즉, 시스템 구성 요소를 CI에 매핑하고 CMDB에 추가하는 것은 비즈니스 가치를 창출하기 위해서입니다.

응용 프로그램 매핑에 컨텐츠를 사용하는 것은 여러 사용 사례에서 일반적인 중간 단계이지만 그렇게 할 수 없는 경우도 있습니다. 컨텐츠를 최종적으로 어떻게 사용하든 관계없이 계획을 세울 때 다음 질문에 답해야 합니다.

- 소비자는 누구인가? 소비자는 CI(및 CI 간의 관계)가 제공하는 정보에 대해 어떤 반응을 보일 것인가? CI 및 관계를 보게 될 비즈니스 상황은 어떤 상황인가? 이러한 CI의 소비자는 사람인가, 제품인가, 아니면 둘 다인가?
- CMDB에 CI 및 관계의 완벽한 결합이 존재한다면 비즈니스 가치를 창출하기 위해 이들을 어떻게 사용할 계획인가?

- 완벽한 매핑은 어떤 형태여야 하는가?
  - 각 CI 간의 관계를 가장 의미 있게 설명하는 용어는 무엇인가?
  - 어떤 유형의 CI가 가장 중요한가?
  - 맵의 최종 용도는 무엇이고 최종 사용자는 누구인가?
- 완벽한 보고서 레이아웃은 어떤 것인가?

비즈니스 정당성을 확립했으면 다음 단계는 문서에 비즈니스 가치를 포함할 차례입니다. 즉, 그리기 도구를 사용하여 완벽한 맵을 그리고 그 영향 및 CI 간의 종속 관계, 보고서, 변경 내용 추적 방법, 중요한 변경 내용, 모니터링, 규정 준수 여부, 사용 사례에 따른 추가적인 비즈니스 가치를 파악해야 합니다.

이 드로잉(또는 모델)을 **청사진**이라고 합니다.

예를 들어 특정 구성 파일이 변경되었을 때 응용 프로그램에서 이를 인지해야 할 경우에는 그려진 맵에서 이 파일이 관련 있는 적절한 CI에 매핑되고 연결되어야 합니다.

개발된 컨텐츠의 최종 사용자인 해당 분야의 SME(내용 전문가)와 협력합니다. 이 전문가는 비즈니스 가치를 제공하기 위해 CMDB에 있어야 할 중요한 엔터티(특성 및 관계가 포함된 CI)를 지적해야 합니다.

한 가지 방법으로 응용 프로그램 소유자에게(이 경우에는 SME에게도) 질문서를 제공할 수 있어야 합니다. 소유자는 위의 목표와 청사진을 지정할 수 있어야 합니다. 소유자는 적어도 응용 프로그램의 현재 아키텍처를 제공해야 합니다.

불필요한 데이터는 제외하고 중요한 데이터만 매핑해야 합니다. 나중에 언제든지 어댑터를 보강할 수 있습니다. 실현 가능하고 가치를 제공하는 제한된 디스커버리를 설정하는 것을 목표로 해야 합니다. 대량의 데이터를 매핑하면 더욱 인상적인 맵을 만들 수 있지만 개발하는 데 시간이 많이 걸리고 혼란스러울 수 있습니다.

모델 및 비즈니스 가치가 분명하면 다음 단계로 넘어갑니다. 다음 단계에서 더욱 구체적인 정보가 제공되면 이 단계로 다시 돌아올 수 있습니다.

통합 요구 사항 조사

이 단계에서는 DFM에서 디스커버리해야 하는 CI 및 관계의 **청사진**(디스커버리할 특성을 포함해야 함)이 필요합니다. 자세한 내용은 "어댑터 개발 및 작성 개요"(12페이지)를 참조하십시오.

이 섹션에는 다음 항목이 포함됩니다.

기존 어댑터 수정

기본 어댑터 또는 필드 어댑터가 있지만 다음과 같은 경우에 기존 어댑터를 수정합니다.

- **필요한 특정 특성을 디스커버리하지 못한 경우**
- **특정 유형의 대상(OS)이 디스커버리되지 않거나 잘못 디스커버리되는 경우**
- **특정 관계를 디스커버리하거나 만들지 못하는 경우**

기존 어댑터가 작업 일부(작업 전체가 아니라)를 수행하면 우선 기존 어댑터를 평가하고 이러한 어댑터 중 하나가 필요한 작업을 거의 수행하는지 확인합니다. 필요한 작업을 수행하는 경우 기존 어댑터를 수정할 수 있습니다.

기존 필드 어댑터를 사용할 수 있는지도 평가해야 합니다. 필드 어댑터는 사용할 수 있지만 기본 어댑터가 아닌 디스커버리 어댑터입니다. 필드 어댑터의 현재 목록을 받으려면 HP 소프트웨어 지원에 문의하십시오.

새 어댑터 작성

다음과 같은 경우 새 어댑터를 개발해야 합니다.

- **어댑터를 작성하는 것이 정보(일반적으로 약 50~100개의 CI 및 관계)를 CMDB에 수동으로 삽입하는 것보다 빠르거나 일회성 작업이 아닌 경우**
- **새 어댑터를 개발할 필요가 있는 경우**
- **기본 어댑터 또는 필드 어댑터를 사용할 수 없는 경우**
- **결과를 다시 사용할 수 있는 경우**
- **대상 환경 또는 해당 데이터를 사용할 수 있는 경우(볼 수 없는 항목은 디스커버리할 수 없음)**

모델 조사

- **UCMDB 클래스 모델(CI 유형 관리자)을 찾아 청사진의 엔터티 및 관계를 기존 CIT에 연결합니다. 버전 업그레이드 중에는 복잡해질 수 있는 상황을 피하기 위해 현재 모델을 유지하는 것이 좋습니다. 모델을 확장해야 할 경우 업그레이드가 기본 CIT를 덮어쓸 수 있으므로 새 CIT를 만들어야 합니다.**
- **현재 모델에 일부 엔터티, 관계 또는 특성이 없는 경우 이들을 만들어야 합니다. HP Universal CMDB를 설치할 때마다 이러한 CIT를 배포할 수 있어야 하므로 이러한 CIT가 포함된 패키지(나중에 이 패키지와 관련된 모든 디스커버리, 보기 및 기타 아티팩트도 여기에 포함됨)를 만드는 것이 좋습니다.**

기술 조사

CMDB에 관련 CI가 있는지 확인했으면 다음 단계는 관련 시스템에서 이 데이터를 검색하는 방법을 결정할 차례입니다.

데이터 검색은 보통 프로토콜을 사용하여 응용 프로그램의 관리 부분, 응용 프로그램의 실제 데이터 또는 응용 프로그램과 관련된 구성 파일이나 데이터베이스에 액세스하여 수행합니다. 시스템에 대한 정보를 제공할 수 있는 데이터 원본은 모두 중요합니다. 기술 조사를 수행하려면 조사 대상 시스템에 대한 광범위한 지식과 경우에 따라 창의성이 필요합니다.

사내 작성 응용 프로그램의 경우 응용 프로그램 소유자에게 질문서 양식을 제공하면 도움이 될 수 있습니다. 이 양식에서 소유자는 청사진 및 비즈니스 가치에 필요한 정보를 제공할 수 있는 응용 프로그램의 모든 분야를 나열해야 합니다. 이 정보는 관리 데이터베이스, 구성 파일, 로그 파일, 관리 인터페이스, 관리 프로그램, 웹 서비스, 전송된 메시지 또는 이벤트 등을 포함해야 하며 이외 다른 정보도 포함할 수 있습니다.

기성 제품의 경우 제품의 문서, 포럼 또는 지원을 중심으로 살펴보아야 합니다. 관리(administration) 안내서, 플러그인 및 통합 안내서, 관리(management) 안내서 등을 검토하십시오. 데이터가 여전히 관리 인터페이스에 나타나지 않는 경우 응용 프로그램의 구성 파일, 레지스트리 항목, 로그 파일, NT 이벤트 로그 및 올바른 작동을 제어하는 응용 프로그램의 모든 아티팩트에 대해 읽어 보십시오.

데이터 액세스 방법 선택 지침

관련성: 대부분의 데이터를 제공하는 원본 또는 원본의 조합을 선택합니다. 대부분의 정보가 하나의 원본에 있고 나머지 정보는 여기저기 흩어져 있거나 액세스하기 어려운 경우 나머지 정보를 가져오기 위한 노력이나 위험에 비해 그 가치를 평가해 봅니다. 때때로 투자한 노력에 비해 그 가치나 비용이 적절하지 않을 경우 청사진을 축소하는 것으로 결정할 수도 있습니다.

다시 사용: HP Universal CMDB에 이미 특정 연결 프로토콜 지원이 포함되어 있는 경우 해당 지원을 사용하는 것이 좋습니다. 이는 DFM 프레임워크에서 연결을 위해 기존의 클라이언트 및 구성을 제공할 수 있음을 의미합니다. 그렇지 않은 경우 인프라 개발에 투자가 필요할 수도 있습니다. **데이터 흐름 관리 > Data Flow Probe 설정 > 도메인 및 프로브** 창에서 현재 지원되는 HP Universal CMDB 연결 프로토콜을 볼 수 있습니다. 각 프로토콜에 대한 자세한 내용은 *HP UCMDB 디스커버리 및 통합 컨텐츠 안내서*에서 지원되는 프로토콜을 설명하는 섹션을 참조하십시오.

모델에 새 CI를 추가하여 새 프로토콜을 추가할 수 있습니다. 자세한 내용은 HP 소프트웨어 지원에 문의하십시오.

참고: Windows 레지스트리 데이터에 액세스하기 위해 WMI 또는 NTCMD를 사용할 수 있습니다.

보안: 정보에 액세스하려면 보통 자격 증명(사용자 이름, 비밀번호)이 필요합니다. 자격 증명은 CMDB에 입력되고 제품 전체에서 안전하게 유지됩니다. 가능한 경우 그리고 보안을 강화해도 사용자가 설정한 다른 원칙과 충돌하지 않는 경우, 가장 덜 까다로우면서도 액세스에 필요한 답을 제공하는 자격 증명이나 프로토콜을 선택하십시오. 예를 들어 JMX(표준 관리 인터페이스, 제한됨) 및 Telnet 두 가지를 통해 정보를 사용할 수 있는 경우 JMX를 사용하는 것이 좋습니다. JMX는 기본적으로 제한된 액세스를 제공하며 보통 기본 플랫폼에 대해서는 액세스를 제공하지 않기 때문입니다.

편리성: 일부 관리 인터페이스에는 매우 진보된 기능이 포함될 수도 있습니다. 예를 들어 더욱 쉽게 쿼리(SQL, WMI)를 작성하여 정보 트리를 탐색하거나 구문 분석을 위한 정규식을 빌드할 수 있는 기능을 말합니다.

개발자 대상: 최종적으로 어댑터를 개발할 사람에게는 특정 기술을 선호하는 경향이 있을 수 있습니다. 두 가지 기술이 다른 요인에서 같은 비용으로 거의 같은 정보를 제공하는 경우 이 점도 고려할 수 있습니다.

요약

이 단계의 결과는 액세스 방법 및 각 방법에서 추출할 수 있는 관련 정보를 설명하는 문서입니다. 이 문서

에는 각 원본에서 관련된 각 청사진 데이터로 연결하는 매핑도 포함해야 합니다.

위 지침에 따라 각 액세스 방법을 표시해야 합니다. 이제 마지막으로, 디스커버리할 원본과 각 원본에서 청사진 모델(이제 해당 UCMDB 모델에 매핑되어 있어야 함)로 추출할 정보에 대한 계획을 세워야 합니다.

통합 컨텐츠 개발

새 통합을 만들기 전에 다음 질문을 통해 통합의 요구 사항을 파악해야 합니다.

- 이 통합은 데이터를 CMDB에 복사해야 하는가? 데이터를 기록으로 추적해야 하는가? 원본을 신뢰할 수 없는가?

  이러한 질문에 "예"라고 대답할 경우 **채우기**가 필요합니다.
- 통합은 보기 및 TQL 쿼리를 위해 데이터를 실시간으로 연합해야 하는가? 데이터 변경 내용의 정확성이 중요한가? CMDB에 복사하기에는 데이터 크기가 너무 크지만 일반적으로 요청되는 데이터 크기는 작은가?

  이러한 질문에 "예"라고 대답할 경우 **연합**이 필요합니다.
- 통합은 원격 데이터 원본에 데이터를 밀어 넣어야 하는가?

  이러한 질문에 "예"라고 대답할 경우 **데이터 밀어넣기**가 필요합니다.
- CI의 ID 길이가 61자 이상이어야 하는가?

  이 질문에 "예"라고 대답할 경우 관련된 모든 CI의 ID가 최대 길이 60자를 넘지 않도록 ID 길이를 **줄입니다.**

참고: 유연성을 최대화하기 위해 연합 흐름 및 채우기 흐름을 같은 통합에 구성할 수 있습니다.

다른 통합 유형에 대한 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 통합 스튜디오를 참조하십시오.

통합 어댑터를 만들 때 사용할 수 있는 옵션은 다음 다섯 가지입니다.

1. Jython 어댑터:
   - 기존 디스커버리 패턴
   - Jython으로 작성
   - 채우기에 사용됨

   자세한 내용은 "Jython 어댑터 개발"(36페이지)을 참조하십시오.
2. Java 어댑터:
   - 연합 SDK 프레임워크에서 어댑터 인터페이스 중 하나를 구현하는 어댑터입니다.
   - 필수 구현에 따라 하나 이상의 연합, 채우기 또는 데이터 밀어넣기에 사용할 수 있습니다.

- Java로 처음부터 작성하여 가능한 모든 원본 또는 대상에 연결할 코드를 작성할 수 있습니다.
- 하나의 데이터 원본 또는 대상에 연결되는 작업에 적합합니다.

자세한 내용은 "Java 어댑터 개발"(182페이지)을 참조하십시오.

3. 일반 DB 어댑터:
   - 연합 SDK 프레임워크를 사용하는 Java 어댑터를 기반으로 하는 추상 어댑터입니다.
   - 외부 데이터 저장소에 연결되는 어댑터를 만들 수 있습니다.
   - 연합과 채우기를 모두 지원합니다(변경 내용 지원을 위해 구현되는 Java 플러그인 사용).
   - 기본적으로 XML 및 속성 구성 파일에 기반하여 비교적 정의하기 쉽습니다.
   - 기본 구성은 UCMDB 클래스와 데이터베이스 열을 매핑하는 **orm.xml** 파일에 기반합니다.
   - 하나의 데이터 원본에 연결되는 작업에 적합합니다.

   자세한 내용은 "일반 데이터베이스 어댑터 개발"(112페이지)을 참조하십시오.
4. 일반 밀어넣기 어댑터:
   - Java 어댑터(연합 SDK 프레임워크) 및 Jython 어댑터에 기반한 추상 어댑터입니다.
   - 원격 대상에 데이터를 밀어 넣는 어댑터를 만들 수 있습니다.
   - UCMDB 클래스 및 XML과, 대상에 데이터를 밀어 넣는 Jython 스크립트 간의 매핑만 정의하면 되므로 비교적 정의하기 쉽습니다.
   - 하나의 데이터 대상에 연결되는 작업에 적합합니다.
   - 데이터 밀어넣기에 사용됩니다.

   자세한 내용은 "밀어넣기 어댑터 개발"(216페이지)을 참조하십시오.
5. 강화된 일반 밀어넣기 어댑터:
   - 일반 밀어넣기 어댑터의 위 기능을 모두 포함합니다.
   - 루트 요소 기반 어댑터입니다.
   - UCMDB 트리 데이터 구조를 대상 트리 데이터 구조에 매핑합니다.

   자세한 내용은 "일반 어댑터를 사용하여 데이터 밀어넣기 수행"(262페이지)을 참조하십시오.

다음 표는 각 어댑터의 기능을 보여 줍니다.

흐름/어댑터	Jython 어댑터	Java 어댑터	일반 DB 어댑터	일반 밀어넣기 어댑터	강화된 일반 밀어넣기 어댑터

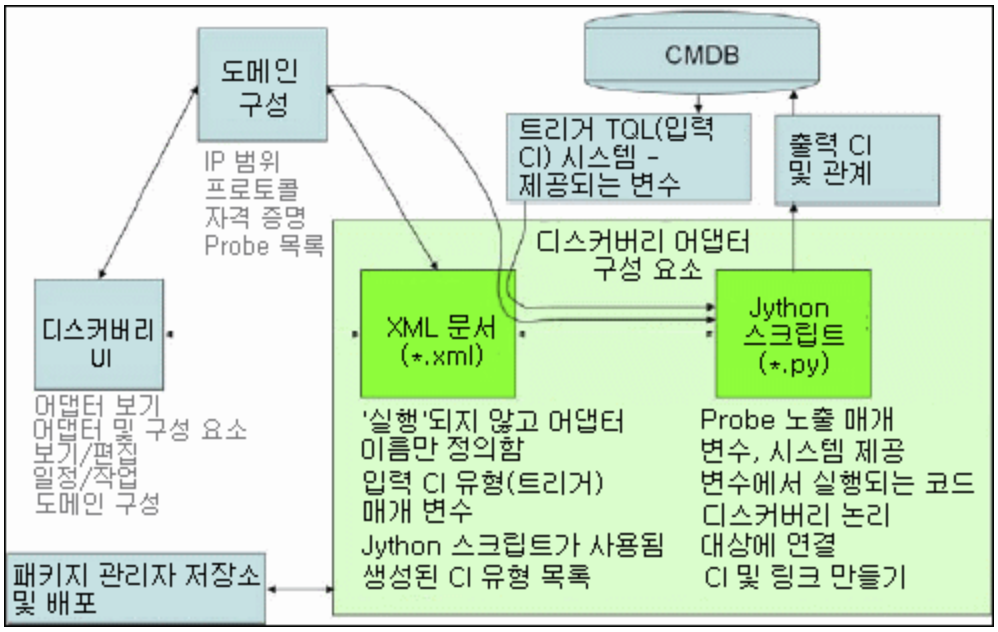
채우기	✓	✓	✓	×	×
연합	×	✓	✓	×	×
데이터 밀어넣기	×	✓	×	✓	✓

디스커버리 컨텐츠 개발

이 섹션의 내용:

디스커버리 어댑터 및 관련 구성 요소

다음 다이어그램은 어댑터의 구성 요소 및 이러한 구성 요소가 디스커버리를 실행하기 위해 상호 작용하는 구성 요소를 보여 줍니다. 녹색 구성 요소는 실제 어댑터이고 파란색 구성 요소는 어댑터와 상호 작용하는 구성 요소입니다.



어댑터의 최소 개념은 XML 문서와 Jython 스크립트라는 두 개의 파일입니다. 입력 CI, 자격 증명 및 사용자 제공 라이브러리를 포함한 디스커버리 프레임워크는 런타임 시 어댑터에 표시됩니다. 이 두 가지 디스커버리 어댑터 구성 요소는 데이터 흐름 관리를 통해 관리됩니다. 이러한 구성 요소는 작업 중에 CMDB 자체에 저장되며 외부 패키지가 남아 있어도 작업에 참조되지 않습니다. 패키지 관리자에서는 새 디스커버리 및 통합 컨텐츠 기능을 유지할 수 있습니다.

어댑터에 대한 입력 CI는 TQL에 의해 제공되며, 시스템 제공 변수에 있는 어댑터 스크립트에 표시됩니다. 어댑터 매개 변수는 대상 데이터로도 제공되므로 어댑터의 특정 함수에 따라 어댑터의 작업을 구성할 수 있습니다.

DFM 응용 프로그램은 새 어댑터를 만들고 테스트하는 데 사용됩니다. 어댑터 작성 중에는 Universal Discovery, 어댑터 관리 및 Data Flow Probe 설정 페이지를 사용합니다.

어댑터는 패키지로 저장되고 전송됩니다. 패키지 관리자 응용 프로그램 및 JMX 콘솔은 새로 만든 어댑터에서 패키지를 만들고 새 시스템에 어댑터를 배포하는 데 사용됩니다.

어댑터 구분

하나의 어댑터에서 전체 디스커버리를 정의할 수 있습니다. 그러나 좋은 설계는 복잡한 시스템을 더욱 간단하고, 보다 관리하기 쉬운 구성 요소로 분리할 수 있어야 합니다.

다음은 어댑터 프로세스를 분리하기 위한 지침 및 모범 사례입니다.

- 단계에 따라 디스커버리를 수행해야 합니다. 각 단계는 시스템의 영역 또는 계층을 매핑해야 하는 어댑터에 의해 표시됩니다. 어댑터는 디스커버리할 이전 단계 또는 계층에 따라 시스템의 디스커버리를 계속해야 합니다. 예를 들어 어댑터 A는 응용 프로그램 서버 TQL 결과에 의해 트리거되고 응용 프로그램 서버 계층을 매핑합니다. 이 매핑의 일부로 JDBC 연결 구성 요소가 매핑됩니다. 어댑터 B는 JDBC 연결 구성 요소를 트리거 TQL로 등록하며 어댑터 A의 결과를 사용하여 데이터베이스 계층에 액세스(예를 들어 JDBC URL 특성을 통해)하고 데이터베이스 계층을 매핑합니다.
- **2단계 연결 패러다임**: 대부분의 시스템에서 데이터에 액세스하려면 자격 증명이 필요합니다. 즉, 이러한 시스템에 대해 사용자/비밀번호 조합을 사용해야 합니다. DFM 관리자는 안전한 방법으로 시스템에 자격 증명 정보를 제공하고 우선 순위가 지정된 몇 가지 로그인 자격 증명을 제공할 수 있습니다. 이를 **프로토콜 사전**이라고 합니다. 어떠한 이유로든 시스템에 액세스할 수 없는 경우 추가 디스커버리는 수행할 필요가 없습니다. 연결에 성공하면 이후의 디스커버리 권한을 위해 성공적으로 사용된 자격 증명 설정이 무엇인지를 나타낼 방법이 있어야 합니다.

  다음과 같은 경우 이러한 두 단계를 통해 두 어댑터가 분리됩니다.

  - **연결 어댑터**: 초기 트리거를 수락하는 어댑터로, 해당 트리거에서 원격 에이전트가 있는지 찾습니다. 프로토콜 사전에서 이 에이전트의 유형과 일치하는 모든 항목을 입력하여 찾습니다. 찾는 데 성공하면 이 어댑터가 그 결과로 원격 에이전트 CI(SNMP, WMI 등)를 제공합니다. 또한 이러한 CI는 이후에 연결할 수 있도록 프로토콜 사전에서 올바른 항목을 지정합니다. 이 에이전트 CI는 컨텐츠 어댑터에 대한 트리거의 일부입니다.

  - **컨텐츠 어댑터**: 이 어댑터의 사전 조건은 이전 어댑터의 성공적인 연결입니다(TQL에서 지정한 사전 조건). 이러한 유형의 어댑터는 원격 에이전트 CI에서 올바른 자격 증명을 가져와 이들을 사용하여 디스커버리된 시스템에 로그인하는 방법이 있으므로 더 이상 모든 프로토콜 사전을 찾아볼 필요가 없습니다.

- 다른 일정 고려 사항은 디스커버리 부문에도 영향을 줄 수 있습니다. 예를 들어 시스템에서 휴식 시간 중에만 쿼리할 수 있으므로 다른 시스템을 디스커버리하는 어댑터에 어댑터를 참가시킬 수 있는 경우에도 일정이 다르면 두 개의 어댑터를 만들어야 함을 의미합니다.
- 같은 시스템을 디스커버리하기 위한 서로 다른 관리 인터페이스 또는 기술의 디스커버리는 각각 별도의 어댑터에 배치되어야 합니다. 그래야 각 시스템 또는 조직에 적합한 액세스 방법을 활성화할 수 있

습니다. 예를 들어 일부 조직에는 컴퓨터에 대한 WMI 액세스 권한이 있지만 SNMP 에이전트는 설치되어 있지 않습니다.

디스커버리 어댑터 구현

DFM 작업은 원격(또는 로컬) 시스템에 액세스하고, 추출한 데이터를 CI로 모델링하고, 해당 CI를 CMDB에 저장하기 위해 수행합니다. DFM 작업은 다음 단계로 구성됩니다.

1. **어댑터 만들기**

   어댑터에 포함할 스크립트를 선택하여 컨텍스트, 매개 변수 및 결과 유형이 포함된 어댑터 파일을 구성합니다. 자세한 내용은 "1단계: 어댑터 만들기"(27페이지)를 참조하십시오.

2. **디스커버리 작업 만들기**

   일정 정보 및 트리거 쿼리로 작업을 구성합니다. 자세한 내용은 "2단계: 어댑터에 작업 할당"(33페이지)을 참조하십시오.

3. **디스커버리 코드 편집**

   어댑터 파일에 포함되어 있고 DFM 프레임워크를 참조하는 Jython 또는 Java 코드를 편집할 수 있습니다. 자세한 내용은 "3단계: Jython 코드 만들기"(34페이지)를 참조하십시오.

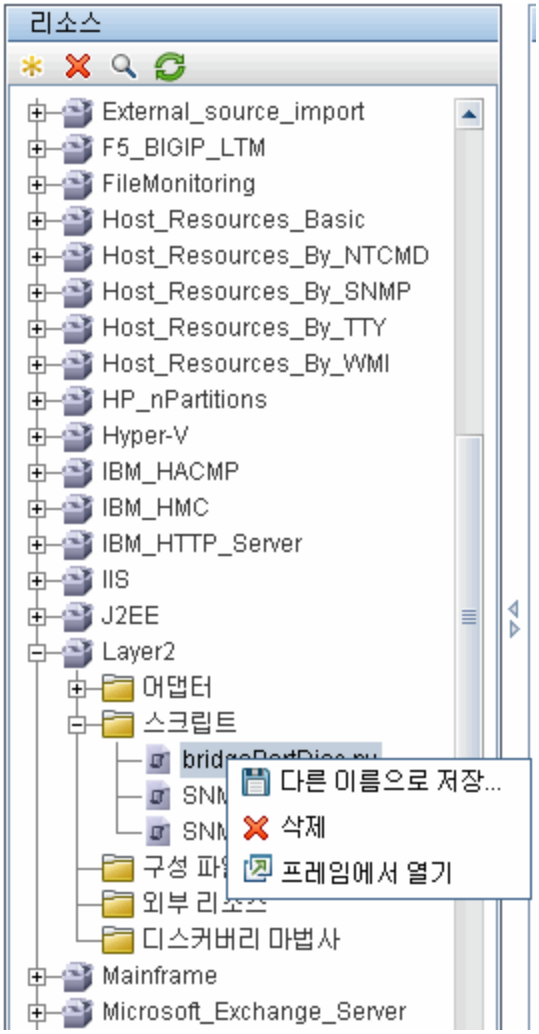
새 어댑터를 작성하려면 위의 각 구성 요소를 만듭니다. 각 구성 요소는 이전 단계의 구성 요소에 자동으로 바인딩됩니다. 예를 들어 작업을 만들고 관련 어댑터를 선택하고 나면 어댑터 파일이 작업에 바인딩됩니다.

어댑터 코드

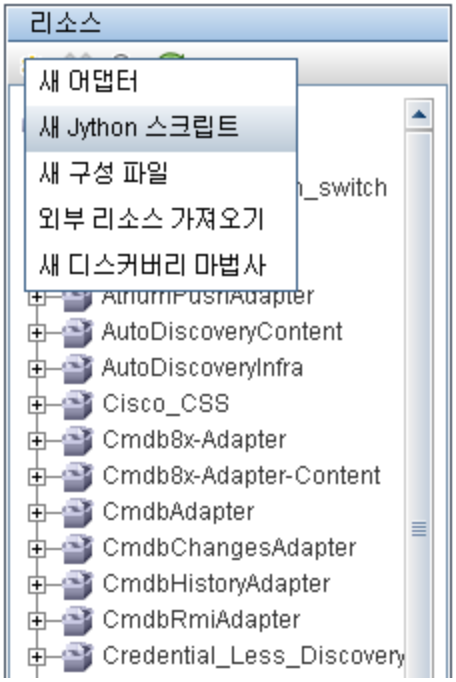
원격 시스템에 연결하고, 해당 데이터를 쿼리하고, CMDB 데이터로 매핑하는 등 Jython 코드로 실제 구현을 수행합니다. 예를 들어, 코드에는 데이터베이스에 연결하고 이 데이터베이스에서 데이터를 추출하는 논리가 포함되어 있습니다. 이 경우 코드는 JDBC URL, 사용자 이름, 비밀번호, 포트 등을 받도록 되어 있습니다. 이러한 매개 변수는 TQL 쿼리에 답하는 데이터베이스의 각 인스턴스별로 지정됩니다. 어댑터(트리거 CI 데이터)에서 이러한 변수를 정의하면, 작업을 실행할 때 이러한 특정 세부 정보가 실행할 코드에 전달됩니다.

어댑터에서 Java 클래스 이름 또는 Jython 스크립트 이름으로 이 코드를 참조할 수 있습니다. 이 섹션에서는 DFM 코드를 Jython 스크립트로 작성하는 방법을 설명합니다.

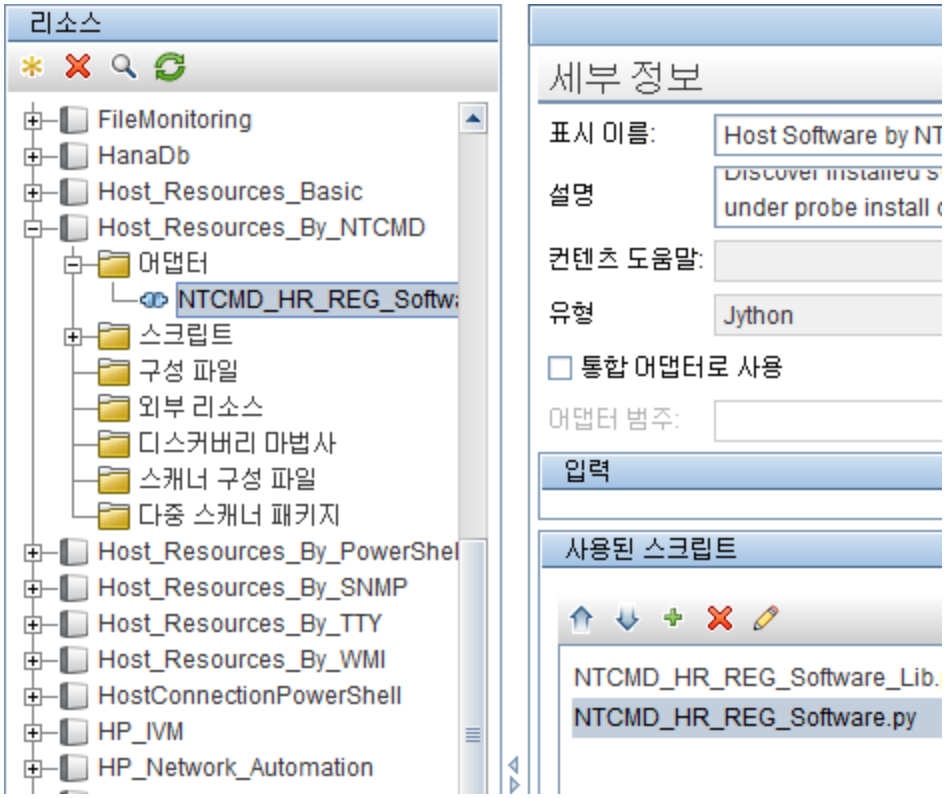
어댑터에는 디스커버리를 실행할 때 사용할 스크립트 목록을 포함할 수 있습니다. 새 어댑터를 만들 때는 보통 새 스크립트를 작성하고 어댑터에 할당합니다. 새 스크립트에는 기본 템플릿이 포함되지만 다른 스크립트 중 하나를 마우스 오른쪽 버튼으로 클릭하고 **다른 이름으로 저장**을 선택하여 그 스크립트를 템플릿으로 사용할 수 있습니다.



새 Jython 스크립트를 작성하는 방법에 대한 자세한 내용은 "3단계: Jython 코드 만들기"(34페이지)를 참조하십시오. 리소스 창에서 다음과 같이 스크립트를 추가합니다.



스크립트 목록은 어댑터에 정의된 순서대로 하나씩 실행됩니다.



참고: 스크립트는 또 다른 스크립트에 의해 단독으로 라이브러리로 사용되고 있더라도 지정되어야 합니다. 이 경우 라이브러리 스크립트를 사용하는 스크립트 전에 먼저 라이브러리 스크립트를 정의

해야 합니다. 이 예에서는 processdbutils.py 스크립트가 마지막 host_processes.py 스크립트에서 사용하는 라이브러리입니다. 라이브러리는 DiscoveryMain() 함수가 없어 실행 가능한 일반 스크립트와 구별됩니다.

1단계: 어댑터 만들기

어댑터는 함수의 정의라고 간주할 수 있습니다. 이 함수는 입력 정의를 정의하고, 입력에 대한 논리를 실행하며, 출력을 정의하고, 결과를 제공합니다.

각 어댑터는 입력과 출력을 지정합니다. 입력과 출력은 모두 어댑터에 특별히 정의된 트리거 CI입니다. 어댑터는 입력 트리거 CI에서 데이터를 추출하고 이 데이터를 코드에 매개 변수로 전달합니다. 관련 CI의 데이터가 코드에 전달되는 경우도 가끔 있습니다. 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 "관련 CI 창"을 참조하십시오. 어댑터의 코드는 코드에 전달되는 이러한 특정 입력 트리거 CI 매개 변수와는 별개의 일반 코드입니다.

입력 구성 요소에 대한 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 "데이터 흐름 관리 개념"을 참조하십시오.

이 섹션에는 다음 항목이 포함됩니다.

1. 어댑터 입력(트리거 CIT 및 입력 쿼리) 정의

   트리거 CIT 및 입력 쿼리 구성 요소를 사용하여 특정 CI를 어댑터 입력으로 정의합니다.

   - 트리거 CIT는 어댑터의 입력으로 사용되는 CIT를 정의합니다. 예를 들어 IP를 디스커버리할 어댑터의 경우 입력 CIT는 네트워크가 됩니다.

   - 입력 쿼리는 편집 가능한 일반 쿼리로, CMDB에 대한 쿼리를 정의합니다. 입력 쿼리는 CIT에 대한 추가 제약 조건을 정의하고(예: 작업에 hostID 또는 application_ip 특성이 필요한 경우), 어댑터에 필요한 경우 CI 데이터를 추가로 정의할 수 있습니다.

     어댑터에 트리거 CI와 관련된 CI의 추가 정보가 필요한 경우 입력 TQL에 노드를 더 추가할 수 있습니다. 자세한 내용은 *HP Universal CMDB 모델링 안내서*에서 "TQL 쿼리에 쿼리 노드 및 관계를 추가하는 방법"을 참조하십시오.

   - 트리거 CI 데이터에는 트리거 CI에 대한 모든 필수 정보뿐 아니라, 입력 TQL에 다른 노드가 정의된 경우 해당 노드의 정보도 포함됩니다. DFM은 변수를 사용하여 CI에서 데이터를 검색합니다. 프로브로 작업을 다운로드할 때 트리거 CI 데이터 변수가 실제 CI 인스턴스의 특성에 존재하는 실제 값으로 바뀝니다.

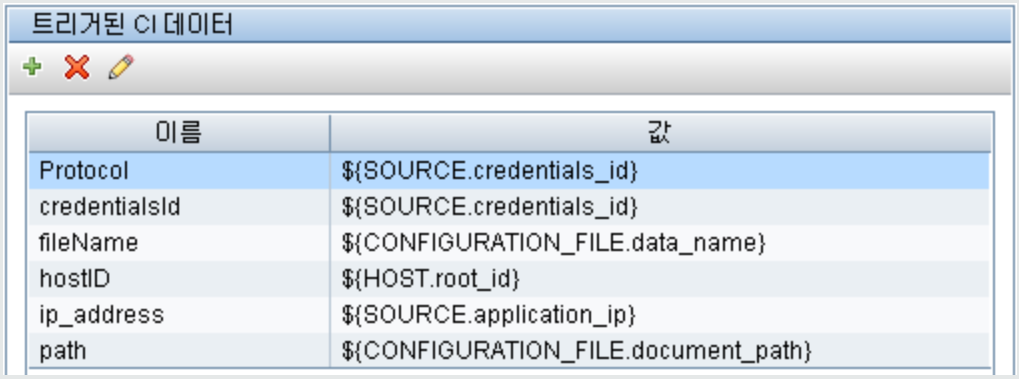
- 대상 데이터의 값이 목록이면 해당 목록에서 프로브로 전송될 항목 수를 정의할 수 있습니다. 이 값을 정의하려면 기본값에 이어지는 항목 수 다음에 콜론을 추가합니다. 대상 데이터에 기본값이 없으면 콜론을 두 개 입력합니다.

  예를 들어, 대상 데이터를 name=portId, value= ${PHYSICALPORT.root_id:NA:1} 또는 name=portId, value= ${PHYSICALPORT.root_id::1}과 같이 입력하는 경우에는 포트 목록에서 첫 번째 포트만 프로브로 전송됩니다.

변수를 실제 데이터로 바꾸는 작업의 예:

이 예제에서는 변수에 의해 **IpAddress** CI 데이터가 시스템의 실제 **IpAddress** CI 인스턴스에 존재하는 실제 값으로 바뀝니다.

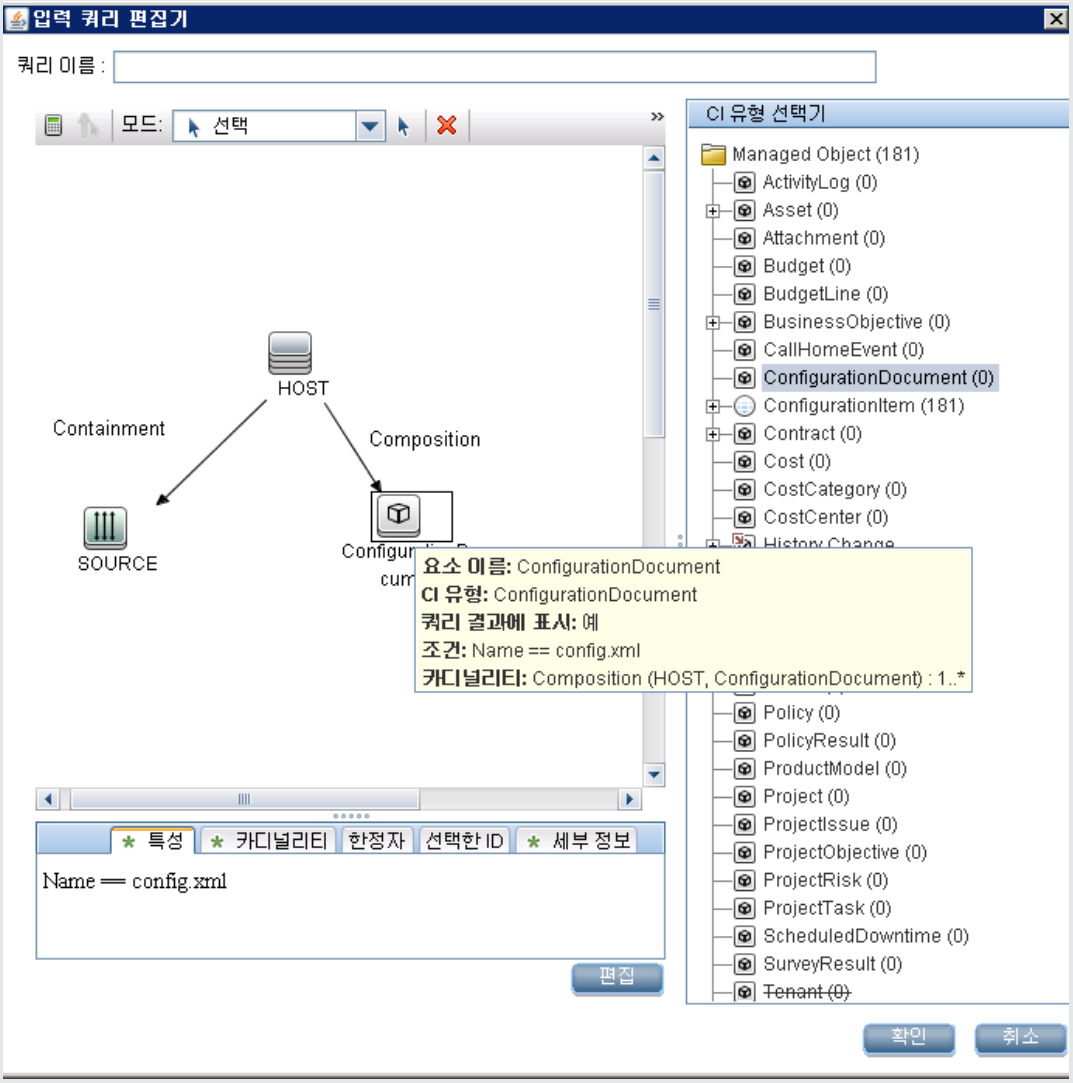
IpAddress CI의 트리거된 CI 데이터에는 fileName 변수가 포함됩니다. 이 변수를 사용하면 입력 TQL의 **CONFIGURATION_DOCUMENT** 노드를 호스트에 있는 구성 파일의 실제 값으로 바꿀 수 있습니다.



모든 변수가 실제 값으로 바뀐 트리거 CI 데이터가 프로브에 업로드됩니다. 어댑터 스크립트에는 정의된 변수의 실제 값을 검색하기 위해 DFM Framework를 사용하는 명령이 포함됩니다.

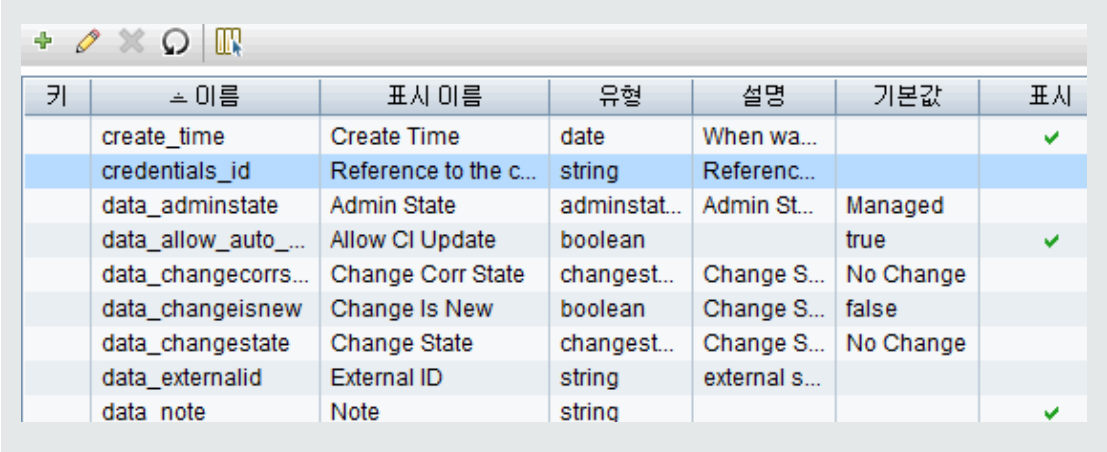
```
Framework.getTriggerCIData ('ip_address')
```

fileName 및 path 변수는 **CONFIGURATION_DOCUMENT** 노드(입력 쿼리 편집기에 정의됨 - 이전 예 참조)의 data_name 및 document_path 특성을 사용합니다.



전체 크기의 그림을 보려면 썸네일을 클릭합니다.

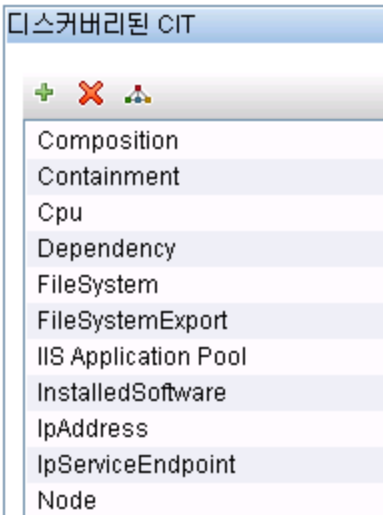
Protocol, credentialsId 및 ip_address 변수는 root_class, credentials_id 및 application_ip 특성을 사용합니다.



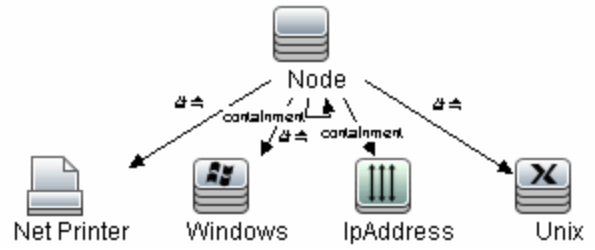
키	이름	표시 이름	유형	설명	기본값	표시
	create_time	Create Time	date	When wa...		✔
	credentials_id	Reference to the c...	string	Referenc...		
	data_adminstate	Admin State	adminstat...	Admin St...	Managed	
	data_allow_auto_...	Allow CI Update	boolean		true	✔
	data_changecorrs...	Change Corr State	changest...	Change S...	No Change	
	data_changeisnew	Change Is New	boolean	Change S...	false	
	data_changestate	Change State	changest...	Change S...	No Change	
	data_externalid	External ID	string	external s...		
	data_note	Note	string			✔

2. 어댑터 출력 정의

어댑터의 출력은 디스커버리된 CI의 목록(데이터 흐름 관리 > 어댑터 관리 > 어댑터 정의 탭 > 디스커버리된 CIT)과 그 CI 사이의 링크입니다.



CIT를 토폴로지 맵, 즉 구성 요소와 구성 요소가 서로 링크된 방식으로 볼 수도 있습니다(디스커버리된 CIT를 맵으로 보기 버튼 클릭).



DFM 코드(즉, Jython 스크립트)에 의해 디스커버리된 CI가 UCMDB의 ObjectStateHolderVector 형식으로 반환됩니다. 자세한 내용은 "Jython 스크립트로 결과 생성"(41페이지)을 참조하십시오.

어댑터 출력의 예:

이 예제에서는 IP CI 출력에 포함할 CIT를 정의합니다.

a. **데이터 흐름 관리 > 어댑터 관리**에 액세스합니다.

b. 리소스 창에서 **Network > 어댑터 > NSLOOKUP_on_Probe**를 선택합니다.

c. 어댑터 정의 탭에서 디스커버리된 CIT 창으로 이동합니다.

d. 어댑터 출력에 포함할 CIT가 나열됩니다. 목록에서 CIT를 추가하거나 제거합니다. 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 "어댑터 정의 탭"을 참조하십시오.

3. 어댑터 매개 변수 다시 정의

둘 이상의 작업에서 어댑터를 구성하려는 경우 어댑터 매개 변수를 다시 정의할 수 있습니다. 예를 들어 SQL_NET_Dis_Connection 어댑터를 SQL로 MSSQL 연결 작업과 SQL로 Oracle 연결 작업 모두에서 사용합니다.

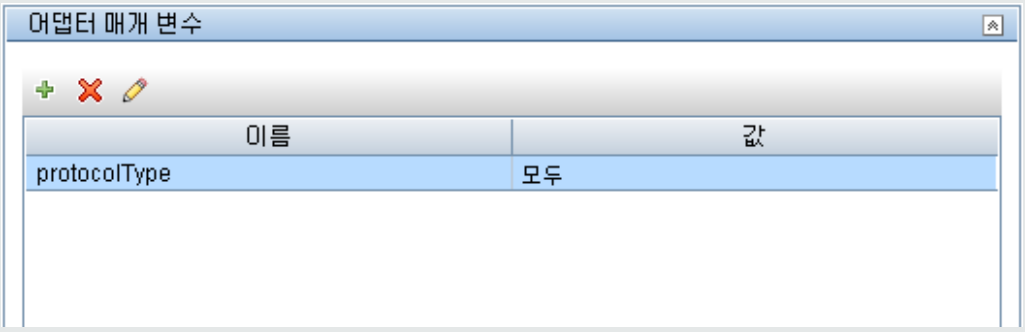
어댑터 매개 변수를 다시 정의하는 작업의 예:

이 예제에서는 어댑터 하나를 사용하여 Microsoft SQL Server 및 Oracle 데이터베이스를 모두 디스커버리할 수 있도록 어댑터 매개 변수를 다시 정의하는 작업을 보여 줍니다.

a. **데이터 흐름 관리 > 어댑터 관리**에 액세스합니다.

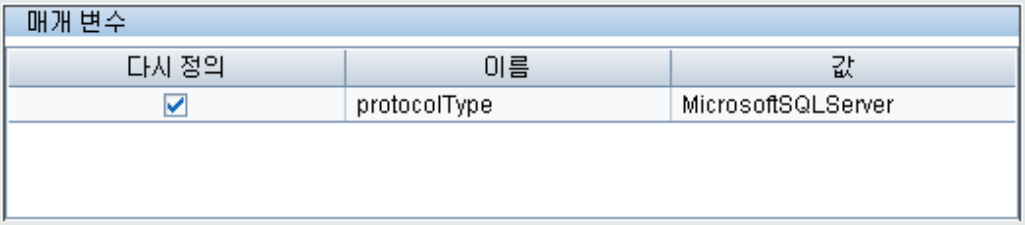
b. 리소스 창에서 **Database_Basic > 어댑터 > SQL_NET_Dis_Connection**을 선택합니다.

c. 어댑터 정의 탭에서 **어댑터 매개 변수** 창으로 이동합니다. protocolType 매개 변수의 값은 **모두**입니다.



d. **SQL_NET_Dis_Connection_MsSql** 어댑터를 마우스 오른쪽 버튼으로 클릭하고 **디스커버리 작업으로 이동 > SQL로 MSSQL 연결**을 선택합니다.

e. **속성** 탭을 표시합니다. 매개 변수 창으로 이동합니다.



MicrosoftSQLServer 값이 모두 값을 덮어씁니다.

참고: **SQL로 Oracle 연결** 작업에도 같은 매개 변수가 포함되어 있지만 매개 변수 값은 Oracle 값으로 덮어쓰게 됩니다.

어댑터 매개 변수 창에서 매개 변수를 추가, 삭제 또는 편집하는 방법에 대한 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 "어댑터 정의 탭"을 참조하십시오.

DFM에서 이 매개 변수에 따라 Microsoft SQL Server 인스턴스를 찾기 시작합니다.

4. 프로브 선택 다시 정의 - 선택 사항

UCMDB 서버에는 UCMDB에 의해 검색된 트리거 CI를 사용하고 다음 옵션 중 하나에 따라 각 트리거 CI에 대해 작업을 실행해야 하는 프로브를 자동으로 선택하는 발송 메커니즘이 있습니다.

- **IP 주소 CI 유형의 경우:** 이 IP에 대해 정의된 프로브를 사용합니다.
- **실행 중인 소프트웨어 CI 유형의 경우: application_ip** 및 **application_ip_domain** 특성을 사용하고 관련 도메인의 IP에 대해 정의된 프로브를 선택합니다.
- **기타 CI 유형의 경우:** CI의 관련 노드(있는 경우)에 따라 노드의 IP를 사용합니다.

자동 프로브 선택은 CI의 관련 노드에 따라 수행됩니다. CI의 관련 노드를 가져온 후에 발송 메커니즘은 노드의 IP 중 하나를 선택하고, 프로브의 네트워크 범위 정의에 따라 프로브를 선택합니다.

다음과 같은 경우에는 프로브를 수동으로 지정해야 하며 자동 발송 메커니즘을 사용하지 않습니다.

- 어댑터에 대해 실행해야 하는 프로브를 이미 알고 있고 프로브를 선택하기 위해 자동 발송 메커니즘을 사용할 필요가 없는 경우(예: 트리거 CI가 프로브 게이트웨이인 경우)
- 자동 프로브 선택에 실패할 수 있는 경우 다음과 같은 상황에서 발생할 수 있습니다.
  - 트리거 CI에 network CIT와 같은 관련 노드가 없는 경우
  - 트리거 CI의 노드에 IP가 여러 개 있고 각 IP가 서로 다른 프로브에 속하는 경우

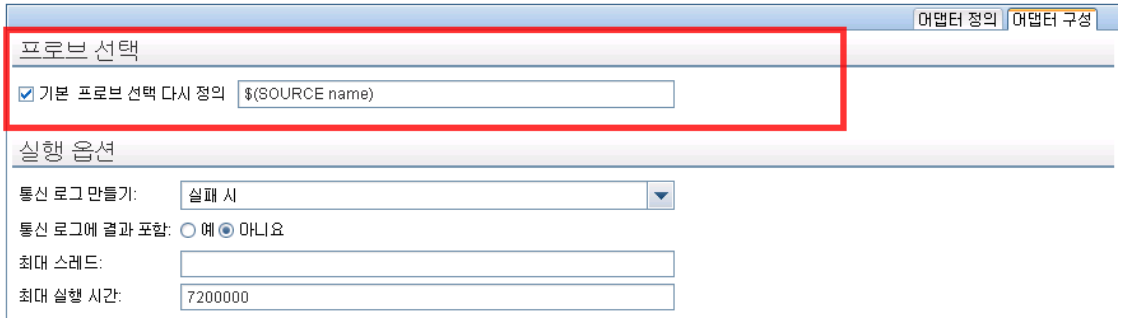
어댑터와 함께 사용할 프로브를 수동으로 지정하려면 다음을 수행합니다.

- 어댑터를 선택하고 **어댑터 구성** 탭을 클릭합니다.
- **트리거 발송 옵션** 아래에서 **기본 Probe 선택 다시 정의**를 선택합니다.
- 상자에 다음 형식 중 하나로 프로브를 입력합니다.

프로브 이름	프로브의 이름입니다.

IP 주소	프로브의 IP 주소입니다. IPv4 또는 IPv6 형식으로 정의할 수 있습니다.
IP, 도메인	IPv4 형식: **16.59.63.86,DefaultDomain** IPv6 형식: **2001:0:9d46:953c:34a9:1e6b:f2ff:fffe,CustomDomain**
도메인 이름	프로브를 선택해야 할 도메인입니다.

예:



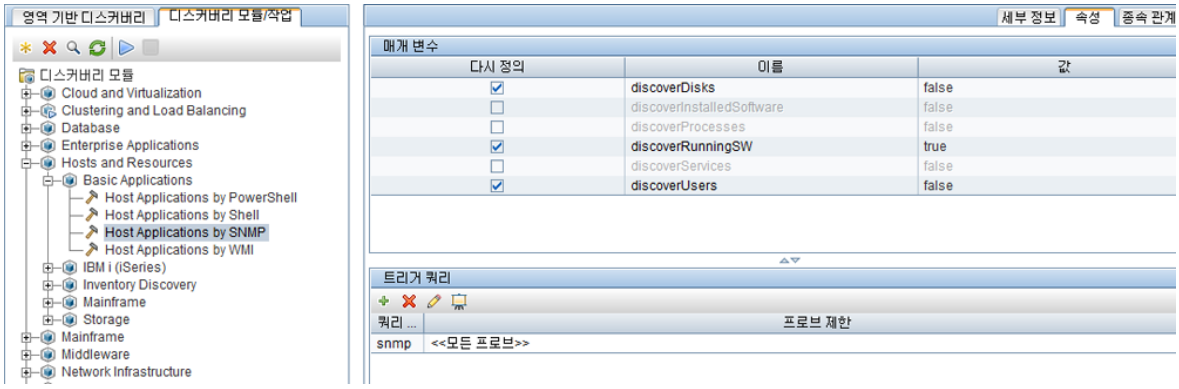
5. 원격 프로세스에 사용할 클래스 경로 구성 - 선택 사항

   자세한 내용은 "원격 프로세스 실행 구성"(35페이지)을 참조하십시오.

2단계: 어댑터에 작업 할당

각 어댑터에는 실행 정책을 정의하는 연관된 작업이 하나 이상 있습니다. 작업을 사용하면 트리거된 CI의 서로 다른 집합에 대해 같은 어댑터의 일정을 다르게 지정할 수 있으며 각 집합에 서로 다른 매개 변수를 제공할 수도 있습니다.

아래 그림에서와 같이, 디스커버리 모듈 트리에 작업이 나타나면 이것이 사용자가 활성화한 엔터티입니다.



트리거 TQL 선택

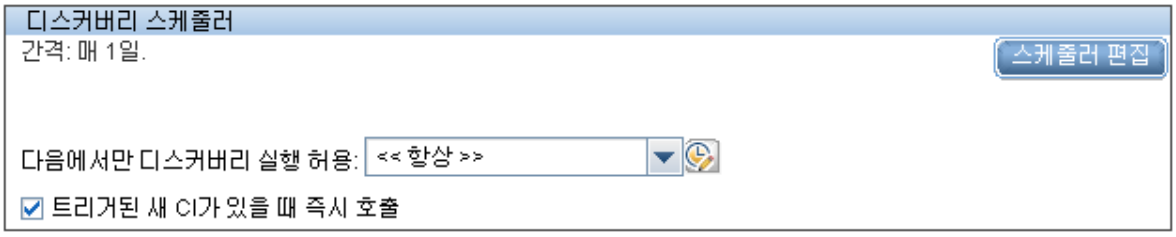
각 작업은 트리거 TQL과 연관되어 있습니다. 이러한 트리거 TQL은 이 작업의 어댑터에 대해 입력 트리거 CI로 사용된 결과를 게시합니다.

트리거 TQL은 입력 TQL에 제약 조건을 추가할 수 있습니다. 예를 들어 입력 TQL의 결과가 SNMP에 연결된 IP이면 트리거 TQL의 결과는 195.0.0.0-195.0.0.10 범위의 SNMP에 연결된 IP일 수 있습니다.

참고: 트리거 TQL은 입력 TQL이 참조하는 동일한 개체를 참조해야 합니다. 예를 들어 입력 TQL이 SNMP를 실행하는 IP를 쿼리하는 경우 같은 작업의 트리거 TQL을 정의할 때 호스트에 연결된 IP를 쿼리하도록 정의할 수 없습니다. 일부 IP는 입력 TQL에서 요구하는 것처럼 SNMP 개체에 연결되어 있지 않을 수 있기 때문입니다.

일정 정보 설정

프로브에 대한 일정 정보는 트리거 CI에서 코드를 실행할 시간을 지정합니다. **트리거된 새 CI가 있을 때 즉시 호출** 확인란을 선택한 경우 코드 또한 이후의 일정 설정에 관계없이 프로브에 도달할 때 각 트리거 CI에서 한 번 실행됩니다.



각 작업에 대해 정해진 일정이 발생할 때마다 프로브에서 해당 작업에 대해 축적된 모든 트리거 CI에 대해 코드를 실행합니다. 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 디스커버리 스케줄러 대화 상자를 참조하십시오.

어댑터 매개 변수 다시 정의

작업을 구성할 때 어댑터 매개 변수를 다시 정의할 수 있습니다. 자세한 내용은 "어댑터 매개 변수 다시 정의"(31페이지)를 참조하십시오.

3단계: Jython 코드 만들기

HP Universal CMDB 에서는 Jython 스크립트를 사용하여 어댑터를 작성합니다. 예를 들어 SNMP를 사용하여 시험하거나 컴퓨터에 연결할 때 SNMP_NET_Dis_Connection 어댑터에서 SNMP_Connection.py 스크립트를 사용합니다. Jython은 Python을 기반으로 하며 Java로 구동되는 언어입니다.

Jython 사용 방법에 대한 자세한 내용은 다음 웹 사이트를 참조하십시오.

- http://www.jython.org
- http://www.python.org

자세한 내용은 "Jython 코드 만들기"(36페이지)를 참조하십시오.

원격 프로세스 실행 구성

Data Flow Probe의 프로세스와 별개인 프로세스에서 디스커버리 작업을 위한 디스커버리를 실행할 수 있습니다.

예를 들어, 해당 작업에서 프로브의 라이브러리와 버전이 다르거나 프로브의 라이브러리와 호환되지 않는 **.jar** 라이브러리를 사용하는 경우 별도의 원격 프로세스에서 작업을 실행할 수 있습니다.

작업에 잠재적으로 사용되는 메모리 양이 많고(많은 데이터 사용) 프로브에 **OutOMemory** 문제가 발생하지 않도록 하려는 경우에도 별도의 원격 프로세스에서 작업을 실행할 수 있습니다.

원격 프로세스로 실행되도록 작업을 구성하려면 어댑터의 구성 파일에서 다음 매개 변수를 정의합니다.

매개 변수	설명
remoteJVMArgs	원격 Java 프로세스에 대한 JVM 매개 변수입니다.
runInSeparateProcess	**true**로 설정하면 별도의 프로세스에서 디스커버리 작업이 실행됩니다.
remoteJVMClasspath	(선택 사항) 원격 프로세스의 클래스 경로를 사용자 지정하여 기본 프로브 클래스 경로를 다시 정의할 수 있습니다. 이 매개 변수는 프로브의 jar와 사용자 정의된 디스커버리에 필수인 사용자 지정 jar의 버전이 호환되지 않을 경우에 유용합니다. **remoteJVMClasspath** 매개 변수가 정의되지 않았거나 비어 있을 경우 기본 프로브 클래스 경로가 사용됩니다. 새 디스커버리 작업을 개발하고 프로브 jar 라이브러리 버전이 해당 작업의 jar 라이브러리와 충돌하지 않게 하려면 기본 디스커버리를 실행하는 데 필요한 최소 클래스 경로 이상을 사용해야 합니다. 최소 클래스 경로는 **basic_discovery_minimal_classpath** 매개 변수의 **DataFlowProbe.properties** 파일에 정의됩니다. **remoteJVMClasspath** 사용자 지정의 예: • 사용자 지정 jar을 기본 프로브 클래스 경로 앞에 붙이거나 뒤에 추가하려면 다음과 같이 **remoteJVMClasspath** 매개 변수를 사용자 지정합니다. custom1.jar;%classpath%;custom2.jar - 이 경우, **custom1.jar**은 기본 프로브 클래스 경로 앞에 오고 **custom2.jar**은 프로브 클래스 경로 뒤에 추가됩니다. • 최소 클래스 경로를 사용하려면 다음과 같이 **remoteJVMClasspath** 매개 변수를 사용자 지정합니다. custom1.jar;%minimal_classpath%;custom2.jar

2장: Jython 어댑터 개발

이 장의 내용:

HP 데이터 흐름 관리 API 참조

사용 가능한 API에 관한 전체 문서는 *HP Universal CMDB 데이터 흐름 관리 API 참조*를 참조하십시오. 이러한 파일은 다음 폴더에 있습니다.

<UCMDB 설치 디렉터리>\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\DDM_JavaDoc\index.html

Jython 코드 만들기

HP Universal CMDB 에서는 Jython 스크립트를 사용하여 어댑터를 작성합니다. 예를 들어 SNMP를 사용하여 컴퓨터에 연결하려고 할 때 **SNMP_NET_Dis_Connection** 어댑터에서 **SNMP_Connection.py** 스크립트를 사용합니다. Jython은 Python을 기반으로 하며 Java로 구동되는 언어입니다.

Jython 사용 방법에 대한 자세한 내용은 다음 웹 사이트를 참조하십시오.

- http://www.jython.org
- http://www.python.org

다음 섹션에서는 DFM 프레임워크 내에서 실제로 Jython 코드를 작성하는 방법을 설명합니다. 이 섹션에서는 Jython 스크립트와 이 스크립트가 호출하는 프레임워크 사이의 연결 지점에 대해 구체적으로 설명하고, 가능하면 항상 사용해야 하는 Jython 라이브러리 및 유틸리티에 대해서도 설명합니다.

참고:

- Universal Discovery용으로 작성된 스크립트는 Jython 버전 2.5.3과 호환되어야 합니다.
- 사용 가능한 API에 관한 전체 문서는 *HP Universal CMDB 데이터 흐름 관리 API 참조*를 참조하십시오.

이 섹션에는 다음 항목이 포함됩니다.

Jython 내에서 외부 Java JAR 파일 사용

새 Jython 스크립트를 개발할 때 외부 Java 라이브러리(JAR 파일) 또는 타사 실행 파일이 때때로 Java 유틸리티 아카이브, 연결 아카이브(예: JDBC 드라이버 JAR 파일) 또는 실행 파일(예: 자격 증명 없는 디스커버리에 사용되는 **nmap.exe**)로 필요합니다.

이러한 리소스는 패키지의 **외부 리소스** 폴더 아래에 번들로 포함되어야 합니다. 이 폴더에 있는 리소스는 HP Universal CMDB 서버에 연결하는 모든 프로브에 자동으로 전송됩니다.

그리고 디스커버리를 시작하면 JAR 파일 리소스가 Jython의 클래스 경로로 로드되어 그 안의 모든 클래스를 가져오거나 사용할 수 있게 됩니다.

코드 실행

작업을 활성화하면 모든 필수 정보를 포함한 작업이 프로브로 다운로드됩니다.

프로브는 작업에 지정된 정보를 사용하여 DFM 코드를 실행하기 시작합니다.

Jython 코드 흐름은 스크립트의 기본 항목부터 실행하여 CI를 디스커버리할 코드를 실행한 후 디스커버리된 CI의 벡터 결과를 제공합니다.

기본 스크립트 수정

기본 스크립트를 수정할 때는 최소한만 수정하고 필요한 메서드는 모두 외부 스크립트에 포함합니다. 그러면 최신 버전의 HP Universal CMDB로 이동할 때 변경 내용을 더욱 효과적으로 추적할 수 있으며 코드를 덮어쓰지 않습니다.

예를 들어 기본 스크립트에서 다음 한 줄의 코드는 응용 프로그램별로 웹 서버 이름을 계산하는 메서드를 호출합니다.

```
serverName = iplanet_cspecific.PlugInProcessing(serverName, transportHN, mam_utils)
```

이 이름을 계산하는 방법을 결정하는 더욱 복잡한 논리는 다음과 같이 외부 스크립트에 포함됩니다.

```
# implement customer specific processing for 'servername' attribute of httpplugin
#
```

```
def PlugInProcessing(servername, transportHN, mam_utils_handle):
    # support application-specific HTTP plug-in naming
    if servername == "appsrv_instance":
        # servername is supposed to match up with the j2ee server name, however some groups do
strange things with their
        # iPlanet plug-in files. this is the best work-around we could find. this join can't be done with IP
address:port
        # because multiple apps on a web server share the same IP:port for multiple websphere
applications
        logger.debug('httpcontext_webapplicationserver attribute has been changed from [' +
servername + '] to [' + transportHN[:5] + '] to facilitate websphere enrichment')
        servername = transportHN[:5]
    return servername
```

외부 스크립트는 외부 리소스 폴더에 저장합니다. 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 리소스 창을 참조하십시오. 이 스크립트를 패키지에 추가하면 다른 작업에도 이 스크립트를 사용할 수 있습니다. 패키지 관리자를 사용하는 방법에 대한 자세한 내용은 *HP Universal CMDB 관리 안내서*에서 패키지 관리자를 참조하십시오.

변경한 코드 한 줄은 업그레이드하는 동안 새 버전의 기본 스크립트가 덮어쓰게 되므로 이 줄을 바꿔야 합니다. 그러나 외부 스크립트는 덮어쓰지 않습니다.

Jython 파일의 구조

Jython 파일은 특정 순서에 따라 세 부분으로 구성됩니다.

1. 가져오기
2. 기본 함수 - DiscoveryMain
3. 함수 정의(선택 사항)

다음은 Jython 스크립트의 예입니다.

```
#   imports section
from appilog.common.system.types import ObjectStateHolder
from appilog.common.system.types.vectors import ObjectStateHolderVector
# Function definition
def foo:
  # do something
#   Main Function
def DiscoveryMain(Framework):
  OSHVResult = ObjectStateHolderVector()
  ## Write implementation to return new result CIs here...
  return OSHVResult
```

가져오기

Jython 클래스는 계층 구조 네임스페이스 전체에 사용됩니다. 버전 7.0 이상에서는 이전 버전과 달리 암시적 가져오기가 없기 때문에 사용하는 모든 클래스를 명시적으로 가져와야 합니다. 성능상의 이유와, 필요한 세부 내용을 표시하여 Jython 스크립트에 대한 이해를 돕기 위해 이렇게 변경되었습니다.

- Jython 스크립트를 가져오려면:

  import logger

- Java 클래스를 가져오려면:

  from appilog.collectors.clients import ClientsConsts

기본 함수 -DiscoveryMain

각 Jython 실행 가능 스크립트 파일에는 기본 함수인 DiscoveryMain이 포함되어 있습니다.

DiscoveryMain 함수는 스크립트의 기본 항목으로, 처음 실행되는 함수입니다. 기본 함수는 다음과 같이 스크립트에 정의된 다른 함수를 호출할 수도 있습니다.

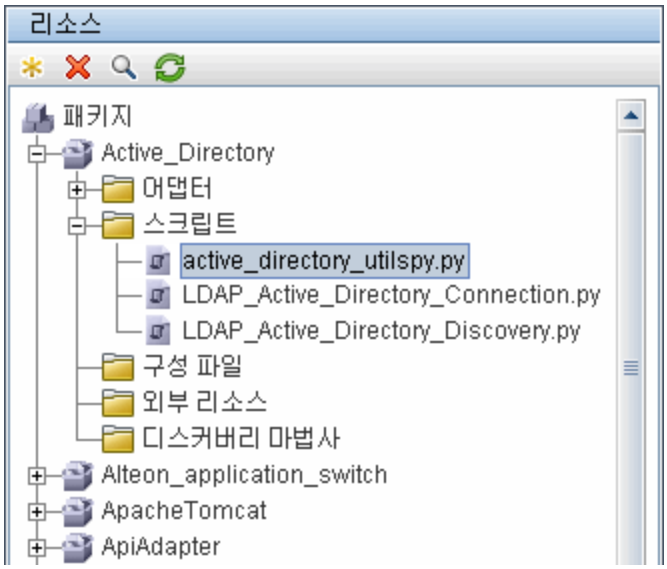
def DiscoveryMain(Framework):

Framework 인수는 기본 함수 정의에 지정되어 있어야 합니다. 이 인수는 기본 함수에서 스크립트를 실행하는 데 필요한 정보(예: 트리거 CI와 매개 변수에 대한 정보)를 가져오기 위해 사용하며, 스크립트를 실행하는 동안 발생하는 오류를 보고하는 데에도 사용할 수 있습니다.

Jython 스크립트는 기본 메서드 없이도 만들 수 있습니다. 그러한 스크립트는 다른 스크립트에서 호출하는 라이브러리 스크립트로 사용됩니다.

함수 정의

각 스크립트는 기본 코드에서 호출하는 추가 함수를 포함할 수 있습니다. 그러한 함수는 각각 현재 스크립트나 다른 스크립트에 있는 다른 함수를 호출할 수 있습니다(import 문 사용). 다른 스크립트를 사용하려면 패키지의 Scripts 섹션에 추가해야 합니다.



다른 함수를 호출하는 함수의 예:

다음 예에서 기본 코드는 내부 메서드 doOSUserOSH(..)를 호출하는 doQueryOSUsers(..) 메서드를 호출합니다.

```
def doOSUserOSH(name):
   sw_obj = ObjectStateHolder('winosuser')

   sw_obj.setAttribute('data_name', name)
   # return the object
   return sw_obj
def doQueryOSUsers(client, OSHVResult):
   _hostObj = modeling.createHostOSH(client.getIpAddress())
   data_name_mib = '1.3.6.1.4.1.77.1.2.25.1.1,1.3.6.1.4.1.77.1.2.25.1.2,string'
   resultSet = client.executeQuery(data_name_mib)
   while resultSet.next():
      UserName = resultSet.getString(2)
      ########## send object ###############
      OSUserOSH = doOSUserOSH(UserName)
      OSUserOSH.setContainer(_hostObj)
      OSHVResult.add(OSUserOSH)
def DiscoveryMain(Framework):
   OSHVResult = ObjectStateHolderVector()
   try:
      client = Framework.createClient(Framework.getTriggerCIData(BaseClient.CREDENTIALS_ID))
   except:
      Framework.reportError('Connection failed')
   else:
      doQueryOSUsers(client, OSHVResult)
      client.close()
   return OSHVResult
```

이 스크립트가 여러 어댑터와 관련된 글로벌 라이브러리라면 각 어댑터에 추가하는 대신 jythonGlobalLibs.xml 구성 파일(**어댑터 관리 > 리소스 창 > AutoDiscoveryContent > 구성 파일**)의 스크립트 목록에 추가할 수 있습니다.

Jython 스크립트로 결과 생성

각 Jython 스크립트는 특정 트리거 CI에서 실행되고 그 결과로 DiscoveryMain 함수의 반환 값을 반환합니다.

스크립트 결과는 실제로 CMDB에 삽입하거나 업데이트할 CI와 링크 그룹입니다. 이 스크립트는 ObjectStateHolderVector 형식의 CI와 링크 그룹을 반환합니다.

ObjectStateHolder 클래스는 CMDB에 정의된 개체 또는 링크를 나타내는 수단입니다. ObjectStateHolder 개체에는 CIT 이름 그리고 특성과 특성 값의 목록이 들어 있습니다. ObjectStateHolderVector는 ObjectStateHolder 인스턴스의 벡터입니다.

ObjectStateHolder 구문

이 섹션에서는 DFM 결과를 UCMDB 모델로 빌드하는 방법을 설명합니다.

CI에 대해 특성을 설정하는 작업의 예:

ObjectStateHolder 클래스는 DFM 결과 그래프를 설명합니다. 각 CI와 링크(관계)는 ObjectStateHolder 클래스의 인스턴스 내에 다음과 같은 Jython 코드 샘플로 포함됩니다.
siebel application server 1 appServerOSH = ObjectStateHolder('siebelappserver') 2 appServerOSH.setStringAttribute('data_name', sblsvrName) 3 appServerOSH.setStringAttribute ('application_ip', ip) 4 appServerOSH.setContainer(appServerHostOSH)

- 첫 번째 줄은 **siebelappserver** 유형의 CI를 만듭니다.
- 두 번째 줄은 **data_name**이라는 특성과 **sblsvrName** 값(서버 이름에 대해 디스커버리된 값으로 설정된 Jython 변수)을 만듭니다.
- 세 번째 줄은 키 특성이 아니면서 CMDB에서 업데이트되는 특성을 설정합니다.
- 네 번째 줄은 제약을 빌드하는 과정입니다(결과는 그래프). 이 줄은 이 응용 프로그램 서버가 호스트(범위의 다른 ObjectStateHolder 클래스) 내에 포함되도록 지정합니다.

참고: Jython 스크립트에서 보고하는 각 CI는 CI의 CI 유형에 대한 모든 키 특성의 값을 포함해야 합니다.

관계(링크)의 예:

다음의 링크 예는 그래프 표시 방법을 설명합니다.
1 linkOSH = ObjectStateHolder('route') 2 linkOSH.setAttribute('link_end1', gatewayOSH) 3 linkOSH.setAttribute('link_end2', appServerOSH)

- 첫 번째 줄은 링크를 만듭니다. 이 링크는 ObjectStateHolder 클래스의 링크이기도 합니다. 둘 사이의 유일한 차이점은 route가 링크 CI 유형이라는 점입니다.

- 두 번째 줄과 세 번째 줄은 각 링크의 끝에 노드를 지정합니다. 이 작업은 각 링크의 최소 키 특성이기 때문에 지정해야 하는 링크의 **end1** 및 **end2** 특성을 사용하여 수행됩니다. 특성 값은 ObjectStateHolder 인스턴스입니다. 끝 1과 끝 2에 대한 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 링크를 참조하십시오.

주의: 링크는 방향을 가지고 있습니다. End 1 및 End 2 노드가 각 끝의 유효한 CIT에 해당하는지 확인해야 합니다. 노드가 유효하지 않으면 결과 개체가 유효성 검사에 실패하여 제대로 보고되지 않습니다. 자세한 내용은 *HP Universal CMDB 모델링 안내서*에서 CI 유형 관계를 참조하십시오.

벡터(CI 수집)의 예:

특성이 있는 개체와 양끝에 개체가 있는 링크를 만들었으면 이제 이들을 그룹화해야 합니다. 해당 개체와 링크를 다음과 같이 ObjectStateHolderVector 인스턴스에 추가하면 됩니다.

```
oshvMyResult = ObjectStateHolderVector()
oshvMyResult.add(appServerOSH)
oshvMyResult.add(linkOSH)
```

이 복합 결과를 CMDB 서버로 전송할 수 있도록 프레임워크에 보고하는 방법에 대한 자세한 내용은 sendObjects 메서드를 참조하십시오.

결과 그래프가 ObjectStateHolderVector 인스턴스에 어셈블되고 나면 CMDB에 삽입될 수 있도록 DFM 프레임워크에 반환되어야 합니다. 이 작업은 DiscoveryMain() 함수의 결과로 ObjectStateHolderVector 인스턴스를 반환함으로써 수행됩니다.

참고: 공통 CIT의 **OSH**를 만드는 방법에 대한 자세한 내용은 "modeling.py"(60페이지)를 참조하십시오.

대량의 데이터 보내기

대량의 데이터(대개 20KB 이상)를 보내면 UCMDB에서 처리하기 어렵습니다. 이 크기의 데이터는 UCMDB로 보내기 전에 더 작은 청크로 분할해야 합니다. 모든 청크가 UCMDB에 올바르게 삽입되려면 각 청크에 청크의 CI에 필요한 식별 정보가 포함되어 있어야 합니다. 이는 Jython 통합을 배포할 때의 일반적인 시나리오입니다. sendObjects 메서드는 청크의 결과를 보내는 데 사용됩니다. Jython 스크립트에서 다수의 결과가 전송되는 경우(기본값은 20,000이지만 DataFlowProbe.properties 파일에서 **appilog.agent.local.maxTaskResultSize** 키를 사용하여 이 값을 구성할 수 있음) 토폴로지에 따라 결과를 청크로 분할해야 합니다. 이러한 청크 분할은 결과가 UCMDB에 올바르게 입력되도록 식별 규칙을 고려하여 수행해야 합니다. Jython 스크립트가 결과를 청크로 분할하지 않을 경우 프로브에서 결과를 청크로 분할하려고 하지만, 이 경우 결과 집합이 크면 성능이 저하될 수 있습니다.

참고: 청크는 일반 디스커버리 작업이 아닌 Jython 통합 어댑터에 사용해야 합니다. 이는 디스커버리 작업이 일반적으로 특정 트리거와 관련된 정보를 디스커버리하고 대량의 정보는 보내지 않기 때문입니다. Jython 통합의 경우 대량의 데이터는 통합의 단일 트리거에서 디스커버리됩니다.

또한 적은 수의 결과에 대해서도 청크를 사용할 수 있습니다. 이 경우 서로 다른 청크의 CI 간에 관계가 있으며 Jython 스크립트 개발자는 다음 두 가지 옵션 중에서 선택할 수 있습니다.

- CI에 대한 링크가 포함된 모든 청크 내의 전체 CI와 모든 CI 식별 정보를 다시 보냅니다.
- CI의 UCMDB ID를 사용합니다. 이렇게 하려면 Jython 스크립트는 UCMDB ID를 얻기 위해 UCMDB 서버에서 각 청크가 처리될 때까지 기다려야 합니다. 이 모드(동기 결과 보내기라고 함)를 사용하도록 설정하려면 SendJythonResultsSynchronously 태그를 어댑터에 추가합니다. 이 태그는 청크 보내기를 마칠 때 청크에 있는 CI의 UCMDB ID를 프로브에서 이미 받았는지 확인합니다. 어댑터 개발자는 다음 청크를 생성하는 데 UCMDB ID를 사용할 수 있습니다. UCMDB ID를 사용하려면 프레임워크 API getIdMapping을 사용합니다.

getIdMapping 사용 예

첫 번째 청크에서 노드를 보내고, 두 번째 청크에서 프로세스를 보냅니다. 프로세스의 루트 컨테이너는 노드입니다. 프로세스 root_container 특성에서 노드의 전체 objectStateHolder를 보내는 대신 getIdMapping API를 사용하여 노드의 UCMDB ID를 가져오고 프로세스 root_container 특성에서 노드 ID만 사용하여 청크를 더 작게 만들 수 있습니다.

Framework 인스턴스

Framework 인스턴스는 Jython 스크립트의 기본 함수에서 제공하는 유일한 인수입니다. 이 인수는 스크립트를 실행하는 데 필요한 정보(예: 트리거 CI와 어댑터 매개 변수에 대한 정보)를 가져오기 위해 사용할 수 있는 인터페이스이며, 스크립트를 실행하는 동안 발생하는 오류를 보고하는 데에도 사용됩니다. 자세한 내용은 "HP 데이터 흐름 관리 API 참조"(36페이지)를 참조하십시오.

Framework 인스턴스의 올바른 사용법은 이 인스턴스를 사용하는 각 메서드에 인수로 전달됩니다.

예:

```
def DiscoveryMain(Framework):
       OSHVResult = helperMethod (Framework)
         return OSHVResult
def helperMethod (Framework):
   ....
    probe_name    = Framework.getDestinationAttribute('probe_name')
    ...
    return result
```

이 섹션에서는 다음과 같이 가장 중요한 Framework 사용법을 설명합니다.

- "Framework.getTriggerCIData(String attributeName)"(43페이지)
- "Framework.createClient(credentialsId, props)"(44페이지)
- "Framework.getParameter (String parameterName)"(45페이지)
- "Framework.reportError(String message) 및 Framework.reportWarning(String message)"(46페이지)

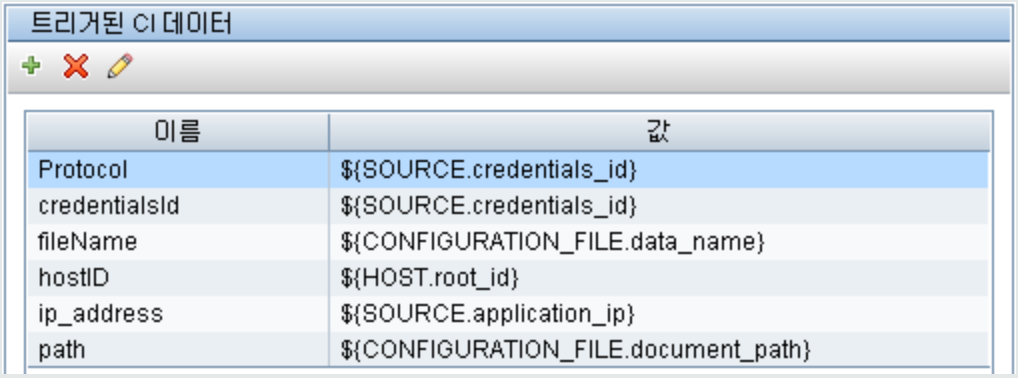
Framework.getTriggerCIData(String attributeName)

이 API는 어댑터에 정의된 트리거 CI 데이터와 스크립트에 정의된 트리거 CI 데이터 사이의 중간 단계를

제공합니다.

자격 증명 정보를 검색하는 작업의 예:

다음 트리거 CI 데이터 정보를 요청하는 경우:



작업에서 자격 증명 정보를 검색하려면 다음 API를 사용합니다.

```
credId = Framework.getTriggerCIData('credentialsId')
```

Framework.createClient(credentialsId, props)

클라이언트 개체를 만들고 해당 클라이언트에서 명령을 실행하여 원격 컴퓨터에 연결합니다. 클라이언트를 만들려면 ClientFactory 클래스를 검색합니다. getClientFactory() 메서드는 요청한 클라이언트 프로토콜 유형을 수신합니다. 프로토콜 상수는 ClientsConsts 클래스에 정의되어 있습니다. 자격 증명 및 지원되는 프로토콜에 대한 자세한 내용은 *HP UCMDB 디스커버리 및 통합 컨텐츠 안내서*를 참조하십시오.

자격 증명 ID에 대한 Client 인스턴스를 만드는 작업의 예:

자격 증명 ID에 대한 Client 인스턴스를 만들려면:

```
properties = Properties()
codePage = Framework.getCodePage()
properties.put( BaseAgent.ENCODING, codePage)
client = Framework.createClient(credentailsID ,properties)
```

이제 Client 인스턴스를 사용하여 관련 컴퓨터 또는 응용 프로그램에 연결할 수 있습니다.

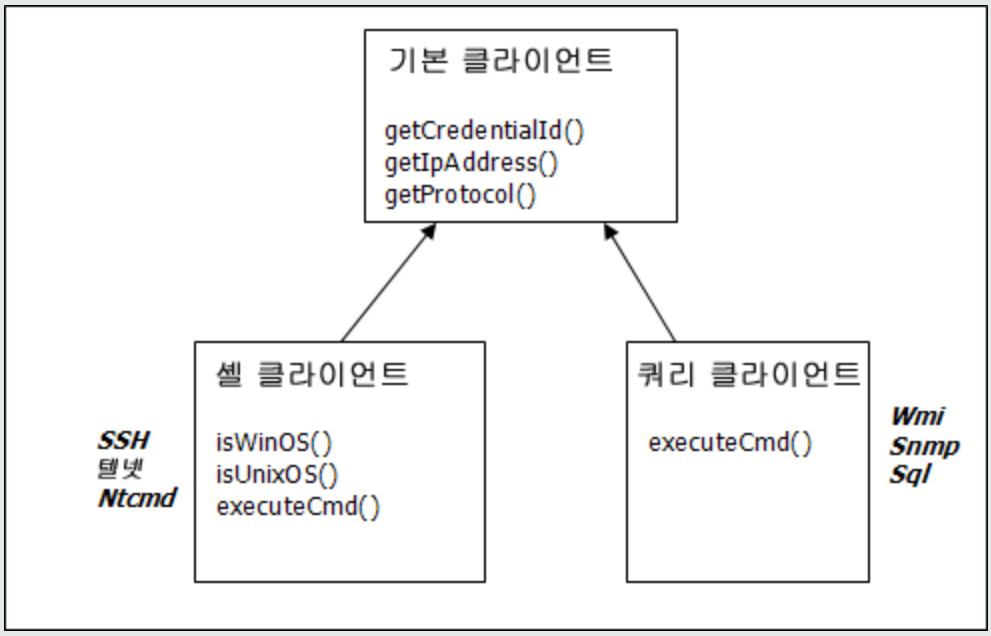
WMI 클라이언트를 만들고 WMI 쿼리를 실행하는 작업의 예:

WMI 클라이언트를 만들고 그 클라이언트를 사용하여 WMI 쿼리를 실행하려면:

```
wmiClient = Framework.createClient(credential)
resultSet = wmiClient. executeQuery("SELECT TotalPhysicalMemory
                                    FROM   Win32_LogicalMemoryConfiguration")
```

참고: createClient() API가 작동하도록 하려면 트리거된 CI 데이터 창에서 트리거 CI 데이터 매개 변수에 **credentialsId = ${SOURCE.credentials_id}** 매개 변수를 추가합니다. 아니면 다음 함수를 호출할 때 수동으로 자격 증명 ID를 추가할 수도 있습니다.
wmiClient = clientFactory().createClient(credentials_id).

다음 다이어그램은 일반적으로 지원되는 API에서 클라이언트의 계층 구조를 나타냅니다.



클라이언트 및 지원되는 API에 대한 자세한 내용은 DFM Framework의 BaseClient, ShellClient 및 QueryClient를 참조하십시오. 이러한 파일은 다음 폴더에 있습니다.

<UCMDB 루트 디렉터리>\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\DDM_Schema\webframe.html

Framework.getParameter (String parameterName)

경우에 따라서는 트리거 CI에 대한 정보를 검색할 뿐 아니라 어댑터 매개 변수 값도 검색해야 합니다. 예:

매개 변수

다시 정의	이름	값
☑	protocolType	MicrosoftSQLServer

protocolType 매개 변수의 값을 검색하는 작업의 예:

Jython 스크립트에서 protocolType 매개 변수의 값을 검색하려면 다음 API를 사용합니다.

```
protocolType = Framework.getParameterValue('protocolType')
```

Framework.reportError(String message) 및 Framework.reportWarning(String message)

스크립트를 실행하는 동안 몇 가지 오류(예: 연결 실패, 하드웨어 문제, 시간 제한)가 발생할 수 있습니다. 이러한 오류가 감지되면 프레임워크에서 문제를 보고할 수 있습니다. 보고된 메시지는 서버에 도달하여 사용자에게 표시됩니다.

오류 및 메시지 보고의 예:

다음 예는 reportError(<오류 메시지>) API의 사용법을 보여 줍니다.

```
try:
   client = Framework.createClient(Framework.getTriggerCIData(BaseClient.CREDENTIALS_ID))

except:
   strException = str(sys.exc_info()[1]).strip()
   Framework. reportError ('Connection failed: %s' % strException)
```

API(Framework.reportError(String message), Framework.reportWarning(String message)) 중 하나를 사용하여 문제를 보고할 수 있습니다. 두 API의 차이점은 오류를 보고할 때 프로브에서 전체 세션의 매개 변수가 포함된 통신 로그 파일을 파일 시스템에 저장한다는 점입니다. 이런 방식으로 세션을 추적하고 오류를 더욱 잘 파악할 수 있습니다.

오류 메시지에 대한 자세한 내용은 "오류 메시지"(61페이지)를 참조하십시오.

올바른 자격 증명 찾기(연결 어댑터용)

원격 시스템에 연결하려는 어댑터는 가능한 모든 자격 증명을 찾아야 합니다. 클라이언트를 만들 때 필요한 매개 변수 중 하나는 자격 증명 ID입니다. 연결 스크립트는 가능한 자격 증명 설정에 대한 액세스 권한을 얻고 Framework.getAvailableProtocols() 메서드를 사용하여 하나씩 연결해 봅니다. 자격 증명 설정 하나가 성공하면 어댑터가 이 트리거 CI(IP에 해당하는 자격 증명 ID 포함)의 호스트에 대한 CI 연결 개체를 CMDB에 보고합니다. 이후의 어댑터는 이 연결 개체 CI를 사용하여 자격 증명 설정에 직접 연결할 수 있습니다. 즉, 어댑터가 가능한 모든 자격 증명을 다시 연결해 볼 필요가 없습니다.

참고: 다음 프로토콜 유형의 경우 중요한 데이터(비밀번호, 개인 키 등)에 대한 액세스가 차단됩니다.

sshprotocol, ntadminprotocol, as400protocol, vmwareprotocol, wmiprotocol, vcloudprotocol, sapjmxprotocol, websphereprotocol, siebelgtwyprotocol, sapprotocol, ldapprotocol, udaprotocol, ntcmdprotocol, snmpprotocol, jbossprotocol, telnetprotocol, powershellprotocol, sqlprotocol, weblogicprotocol

이러한 프로토콜 유형을 사용하려면 전용 클라이언트를 사용해야 합니다.

다음 예는 SNMP 프로토콜의 모든 항목을 가져오는 방법을 보여 줍니다. 여기서 IP는 트리거 CI 데이터에서 가져온 것입니다(# 트리거 CI 데이터 값 가져오기).

연결 스크립트는 가능한 모든 프로토콜 자격 증명을 요청하고(# 모든 프로토콜 자격 증명 검토) 하나가 성공할 때까지 반복해서 연결을 시도합니다(resultVector). 자세한 내용은 "어댑터 구분"(23페이지)에서 **2단계 연결 패러다임** 항목을 참조하십시오.

예

```
import logger
import netutils
import sys
import errorcodes
import errorobject

# Java imports
from java.util import Properties
from com.hp.ucmdb.discovery.common import CollectorsConstants
from appilog.common.system.types.vectors import ObjectStateHolderVector
from com.hp.ucmdb.discovery.library.clients import ClientsConsts
from com.hp.ucmdb.discovery.library.scope import DomainScopeManager

TRUE = 1
FALSE = 0

def mainFunction(Framework, isClient, ip_address = None):
    _vector = ObjectStateHolderVector()
    errStr = ''
    ip_domain  = Framework.getDestinationAttribute('ip_domain')
    # Get the Trigger CI data values
    ip_address = Framework.getDestinationAttribute('ip_address')

            if (ip_domain == None):
              ip_domain = DomainScopeManager.getDomainByIp(ip_address, None)

            protocols = netutils.getAvailableProtocols(Framework, ClientsConsts.SNMP_PROTOCOL_
            NAME, ip_address, ip_domain)
            if len(protocols) == 0:
               errStr = 'No credentials defined for the triggered ip'
               logger.debug(errStr)
               errObj = errorobject.createError(errorcodes.NO_CREDENTIALS_FOR_TRIGGERED_IP,
            [ClientsConsts.SNMP_PROTOCOL_NAME], errStr)
               return (_vector, errObj)

            connected = 0
            # Go over all the protocol credentials
            for protocol in protocols:
               client = None
               try:
                  try:
                     logger.debug('try to get snmp agent for: %s:%s' % (ip_address, ip_domain))
```

```
                if (isClient == TRUE):
                    properties = Properties()
                    properties.setProperty(CollectorsConstants.DESTINATION_DATA_IP_ADDRESS,
            ip_address)
                    properties.setProperty(CollectorsConstants.DESTINATION_DATA_IP_DOMAIN,
            ip_domain)
                    client = Framework.createClient(protocol, properties)
                else:
                    properties = Properties()
                    properties.setProperty(CollectorsConstants.DESTINATION_DATA_IP_ADDRESS,
            ip_address)
                    client = Framework.createClient(protocol, properties)
                logger.debug('Running test connection queries')
                testConnection(client)
                Framework.saveState(protocol)
                logger.debug('got snmp agent for: %s:%s' % (ip_address, ip_domain))
                isMultiOid = client.supportMultiOid()
                logger.debug('snmp server isMultiOid state=%s' %isMultiOid)

        client.close()
        client = None

            except:
                if client != None:
                    client.close()
                    client = None
                logger.debugException('Unexpected SNMP_AGENT Exception:')
                lastExceptionStr = str(sys.exc_info()[1]).strip()
            finally:
                if client != None:
                    client.close()
                    client = None

    return (_vector, error)
```

Java에서 예외 처리

일부 Java 클래스에서는 오류 발생 시 예외가 발생합니다. 예외를 포착하여 처리하는 것이 좋습니다. 그렇게 하지 않으면 어댑터가 예상치 않게 종료됩니다.

알려진 예외를 포착할 때는 대부분 로그에 대한 스택 추적을 인쇄하고 UI에 적절한 메시지를 표시해야 합니다.

> **참고:** Python에도 동일한 이름의 기본 예외 클래스가 있기 때문에 다음 예에 표시된 대로 Java 기본 예외 클래스를 가져오는 것이 매우 중요합니다.

```
from java.lang import Exception as JException
try:
```

```
    client = Framework.createClient(Framework.getTriggerCIData(BaseClient.CREDENTIALS_ID))
except JException, ex:
    # process java exceptions only
    Framework.reportError('Connection failed')
    logger.debugException(str(ex))
     return
```

심각한 예외가 아니라서 스크립트를 계속 실행할 수 있으면 reportError() **메서드 호출을 생략하고 스크립트를 계속 실행해야 합니다.**

Jython 버전 2.1에서 2.5.3으로의 마이그레이션 문제 해결

Universal Discovery는 이제 Jython 버전 2.5.3을 사용합니다. 모든 기본 스크립트가 올바르게 마이그레이션되었습니다. 이 업그레이드 전에 디스커버리에서 사용할 자체 Jython 스크립트를 개발한 경우 다음과 같은 문제가 발생할 수 있습니다. 이 경우 표시된 대로 수정해야 합니다.

> **참고: 다음과 같이 내용을 변경하려면 숙련된 Jython 개발자여야 합니다.**

문자열 형식 지정

- **오류 메시지:** TypeError: int argument required
- **가능한 원인: 정수 데이터가 포함된 문자열 변수의 10진수에 문자열 형식 지정 사용**
- **문제가 있는 Jython 2.1 코드:**

  ```
  variable = "43"
  print "%d" % variable
  ```

- **올바른 Jython 2.5.3 코드:**

  ```
  variable = "43"
  print "%s" % variable
  ```

  또는

  ```
  variable = "43"
  print "%d" % int(variable)
  ```

문자열 유형 확인

아래 코드는 입력에 유니코드 문자열이 포함된 경우 제대로 작동하지 않을 수 있습니다.

- **문제가 있는 Jython 2.1 코드:** isinstance(unicodeStringVariable,'')
- **올바른 Jython 2.5.3 코드:** isinstance(unicodeStringVariable,basestring)

  basestring**을 통해 비교하여 개체가** str**의 인스턴스인지 또는 유니코드의 인스턴스인지 테스트해야 합니다.**

파일에 ASCII가 아닌 문자 포함

- **오류 메시지:**
  SyntaxError: Non-ASCII character in file 'x', , but no encoding declared; see

http://www.python.org/peps/pep-0263.html for details

- **올바른 Jython 2.5.3 코드:** (파일의 첫 번째 줄에 다음 항목을 추가합니다.)

  # coding: utf-8

하위 패키지 가져오기

- **오류 메시지:**
  AttributeError: 'module' object has no attribute 'sub_package_name'
- **가능한 원인:** import 문에 하위 패키지의 이름을 명시적으로 지정하지 않고 하위 패키지를 가져왔습니다.
- **문제가 있는 Jython 2.1 코드:**

  import a
  print dir(a.b)

  하위 패키지를 명시적으로 가져오지 않습니다.
- **올바른 Jython 2.5.3 코드:**

  import a.b

  또는

  from a import b

반복기 변경 사항

Jython 2.2부터는 **__iter__** 메서드를 사용하여 **for-in** 블록의 범위에서 컬렉션을 반복합니다. 반복기는 **next** 메서드를 구현하여 적절한 요소를 반환해야 합니다. 그렇지 않으면 컬렉션의 끝에 도달한 경우 **StopIteration** 오류가 발생합니다. **__iter__** 메서드가 구현되지 않은 경우에는 대신 **getitem** 메서드가 사용됩니다.

예외 발생

- **예외를 발생시키는 Jython 2.1 메서드는 더 이상 사용되지 않습니다.**
  raise Exception, 'Failed getting contents of file'
- **예외를 발생시키는 권장 Jython 2.5.3 메서드는 다음과 같습니다.**
  raise Exception('Failed getting contents of file')

Jython 어댑터에서 지역화 지원

다국어 로캘 기능을 사용하면 DFM을 여러 OS(운영 체제) 언어로 사용할 수 있고 런타임 시 적절하게 사용자 지정할 수 있습니다.

이 섹션의 내용:

- "새로운 언어에 대한 지원 추가"(51페이지)
- "기본 언어 변경"(52페이지)
- "인코딩할 문자 집합 결정"(52페이지)

새로운 언어에 대한 지원 추가

이 작업에서는 새로운 언어에 대한 지원을 추가하는 방법을 설명합니다.

이 작업에는 다음 단계가 포함됩니다.

1. 리소스 번들(*.properties 파일) 추가

   실행할 작업에 따라 리소스 번들을 추가합니다. 다음 표에는 DFM 작업 및 각 작업에서 사용되는 리소스 번들이 나열되어 있습니다.

작업	리소스 번들의 기본 이름
셸을 통해 파일 모니터링	langFileMonitoring
셸별 호스트 리소스 및 응용 프로그램	langHost_Resources_By_TTY, langTCP
DNS 서버에서 NSLOOKUP을 사용하여 셸을 통해 호스팅	langNetwork
셸을 통해 연결 호스팅	langNetwork
셸 또는 SNMP를 통해 네트워크 데이터 수집	langTCP
SNMP를 통해 리소스 및 응용 프로그램 호스팅	langTCP
NTCMD를 통한 Microsoft Exchange 연결, NTCMD를 통한 Microsoft Exchange 토폴로지	msExchange
NTCMD를 통한 MS 클러스터	langMsCluster

   번들에 대한 자세한 내용은 "리소스 번들 사용"(54페이지)을 참조하십시오.

2. 언어 개체 선언 및 등록

   새 언어를 정의하려면 현재 지원되는 모든 언어 목록이 포함된 shellutils.py 스크립트에 다음 두 줄의 코드를 추가합니다. 이 스크립트는 AutoDiscoveryContent 패키지에 포함되어 있습니다. 이 스크립트를 보려면 어댑터 관리 창에 액세스합니다. 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 어댑터 관리 창을 참조하십시오.

a. 다음과 같이 언어를 선언합니다.

LANG_RUSSIAN = Language(LOCALE_RUSSIAN, 'rus', ('Cp866', 'Cp1251'), (1049,), 866)

클래스 언어에 대한 자세한 내용은 "API 참조"(55페이지)를 참조하십시오. 클래스 로캘 개체에 대한 자세한 내용은 http://java.sun.com/j2se/1.5.0/docs/api/java/util/Locale.html을 참조하십시오. 기존 로캘을 사용하거나 새 로캘을 정의할 수 있습니다.

b. 다음 컬렉션에 언어를 추가하여 등록합니다.

LANGUAGES = (LANG_ENGLISH, LANG_GERMAN, LANG_SPANISH, LANG_RUSSIAN, LANG_JAPANESE)

기본 언어 변경

OS 언어를 확인할 수 없는 경우 기본 언어가 사용됩니다. 기본 언어는 **shellutils.py** 파일에 지정되어 있습니다.

```
#default language for fallback
DEFAULT_LANGUAGE = LANG_ENGLISH
```

기본 언어를 변경하려면 DEFAULT_LANGUAGE 변수를 다른 언어로 초기화합니다. 자세한 내용은 "새로운 언어에 대한 지원 추가"(51페이지)를 참조하십시오.

인코딩할 문자 집합 결정

명령 출력 디코딩에 적합한 문자 집합은 런타임 시 결정됩니다. 다국어 솔루션은 다음 사실과 가정에 따라 결정됩니다.

1. 로캘과는 별도로 OS 언어를 결정할 수 있습니다. 예를 들어 Windows에서 **chcp** 명령을 실행하거나 Linux에서 **locale** 명령을 실행하여 결정할 수 있습니다.
2. 관계 언어 인코딩은 잘 알려져 있으며 정적으로 정의할 수 있습니다. 예를 들어 러시아어는 가장 일반적인 두 가지 인코딩인 Cp866 및 Windows-1251을 사용합니다.
3. 언어마다 하나의 문자 집합이 기본 설정됩니다. 예를 들어 러시아어의 기본 문자 집합은 Cp866입니다. 이는 대부분의 명령이 이 인코딩으로 출력됨을 의미합니다.
4. 다음 명령 출력이 제공되는 인코딩은 예측할 수 없지만 주어진 언어에 가능한 인코딩 중 하나입니다. 예를 들어 러시아어 로캘이 설정된 Windows 컴퓨터를 사용하는 경우 시스템에서 **ver** 명령 출력은 Cp866으로 제공되지만 **ipconfig** 명령은 Windows-1251로 제공됩니다.
5. 알려진 명령은 출력에서 알려진 키워드를 생성합니다. 예를 들어 **ipconfig** 명령에는 **IP-Address** 문자열이 변환된 형식으로 포함됩니다. 따라서 영어 OS의 경우 **IP-Address**, 러시아어 OS의 경우 IP-Адрес, 독일어 OS의 경우 **IP-Adresse** 등이 **ipconfig** 명령 출력에 포함됩니다.

명령 출력이 어떤 언어로 생성되었는지 디스커버리되면(# 1) 가능한 문자 집합이 하나 또는 둘로 제한됩니다(# 2). 뿐만 아니라 이 출력에 포함된 키워드를 알 수 있게 됩니다(# 5).

그러므로 해결책은 결과에서 키워드를 검색하여 가능한 인코딩 중 하나로 명령 출력을 디코딩하는 것입니다. 키워드가 검색되면 현재 문자 집합은 올바른 인코딩으로 간주됩니다.

지역화된 데이터를 사용할 새 작업 정의

이 작업에서는 지역화된 데이터를 사용할 수 있는 새 작업을 작성하는 방법을 설명합니다.

Jython 스크립트는 보통 명령을 실행하고 출력을 구문 분석합니다. 올바르게 디코딩된 방법으로 이 명령 출력을 받으려면 ShellUtils 클래스에 대한 API를 사용합니다. 자세한 내용은 "HP Universal CMDB 웹 서비스 API 개요"(317페이지)를 참조하십시오.

이 코드는 보통 다음 형식으로 사용됩니다.

```
client = Framework.createClient(protocol, properties)
shellUtils = shellutils.ShellUtils(client)
languageBundle = shellutils.getLanguageBundle ('langNetwork', shellUtils.osLanguage, Framework)
strWindowsIPAddress = languageBundle.getString('windows_ipconfig_str_ip_address')
ipconfigOutput = shellUtils.executeCommandAndDecode('ipconfig /all', strWindowsIPAddress)
#Do work with output here
```

1. **클라이언트를 만듭니다.**

   ```
   client = Framework.createClient(protocol, properties)
   ```

2. **ShellUtils 클래스의 인스턴스를 만들고 여기에 운영 체제 언어를 추가합니다. 언어가 추가되지 않으면 기본 언어가 사용됩니다(대개 영어가 사용됨).**

   ```
   shellUtils = shellutils.ShellUtils(client)
   ```

   개체를 초기화하는 동안 DFM에서 컴퓨터 언어를 자동으로 검색하고 미리 정의된 Language 개체에서 기본 인코딩을 설정합니다. 기본 인코딩은 인코딩 목록에서 가장 먼저 나타나는 인스턴스입니다.

3. **getLanguageBundle 메서드를 사용하여 shellclient에서 적절한 리소스 번들을 검색합니다.**

   ```
   languageBundle = shellutils.getLanguageBundle ('langNetwork', shellUtils.osLanguage, Framework)
   ```

4. **리소스 번들에서 특정 명령에 적합한 키워드를 검색합니다.**

   ```
   strWindowsIPAddress = languageBundle.getString('windows_ipconfig_str_ip_address')
   ```

5. **executeCommandAndDecode 메서드를 호출하고 ShellUtils 개체에서 이 메서드로 키워드를 전달합니다.**

   ```
   ipconfigOutput = shellUtils.executeCommandAndDecode('ipconfig /all', strWindowsIPAddress)
   ```

   사용자를 API 참조(여기에 이 메서드에 대한 자세한 설명이 있음)에 연결하려면 ShellUtils object도 필요합니다.

6. **일반적인 방법으로 출력을 구문 분석합니다.**

키워드를 사용하지 않는 명령 디코딩

현재의 지역화 방법은 키워드를 사용하여 모든 명령 출력을 디코딩합니다. 자세한 내용은 "지역화된 데이터를 사용할 새 작업 정의"(53페이지)의 리소스 번들에서 키워드 검색에 대한 단계를 참조하십시오.

그러나 다른 방법은 첫 번째 명령 출력만 키워드를 사용하여 디코딩하고 그 밖의 명령은 첫 번째 명령을 디코딩하는 데 사용한 문자 집합으로 디코딩합니다. 그렇게 하려면 **ShellUtils** 개체의 **getCharsetName** 및 **useCharset** 메서드를 사용합니다.

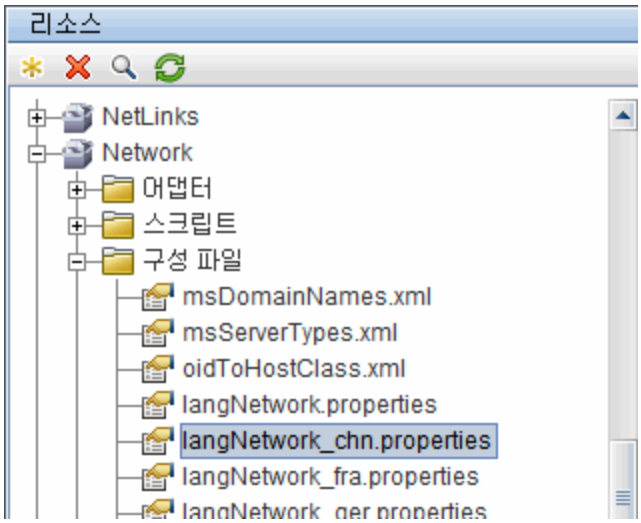
일반적인 사용 예는 다음과 같습니다.

1. **executeCommandAndDecode** 메서드를 한 번 호출합니다.
2. **getCharsetName** 메서드를 통해 최근에 사용한 문자 집합 이름을 가져옵니다.
3. **ShellUtils** 개체의 **useCharset** 메서드를 호출하여 **shellUtils**에서 기본적으로 이 문자 집합을 사용하도록 합니다.
4. **ShellUtils**의 **execCmd** 메서드를 한 번 또는 여러 번 호출합니다. 이전 단계에서 지정한 문자 집합을 사용하여 출력이 반환됩니다. 추가로 발생되는 디코딩 작업이 없습니다.

리소스 번들 사용

리소스 번들은 확장자가 properties(***.properties**)인 파일입니다. 속성 파일은 key = value 형식으로 데이터를 저장하는 사전이라고 생각할 수 있습니다. 속성 파일의 각 행에는 하나의 key = value 연관이 있습니다. 리소스 번들의 기본 기능은 키에 따라 값을 반환하는 것입니다.

리소스 번들은 프로브 컴퓨터의 **C:\hp\UCMDB\DataFlowProbe\runtime\probeManager\discoveryConfigFiles** 아래에 있습니다. 리소스 번들은 UCMDB 서버에서 다른 구성 파일로 다운로드됩니다. 그리고 리소스 창에서 편집하거나, 추가하거나, 제거할 수 있습니다. 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 구성 파일 창을 참조하십시오.



대상을 디스커버리할 때 DFM은 일반적으로 명령 출력이나 파일 컨텐츠에서 텍스트를 구문 분석해야 합니다. 이 구문 분석은 대개 정규식에 따라 수행됩니다. 언어가 다르면 다른 정규식을 사용하여 구문 분석해야 합니다. 모든 언어에 대해 코드를 한 번씩 작성하려면 모든 언어별 데이터를 리소스 번들로 추출해야 합니다. 언어마다 하나의 리소스 번들이 있습니다. 하나의 리소스 번들에 여러 언어의 데이터를 포함할 수 있지만 DFM에서는 하나의 리소스 번들에 한 언어의 데이터만 포함됩니다.

Jython 스크립트 자체는 하드 코드된 특정 언어 관련 데이터(예: 특정 언어 관련 정규식)를 포함하지 않습니다. 스크립트는 원격 시스템의 언어를 확인하고 적절한 리소스 번들을 로드하며 특정 키로 모든 언어별 데이터를 가져옵니다.

DFM에서 리소스 번들은 <base_name>_<language_identifier>.properties와 같은 특정 이름 형식을 사용합니다(예: langNetwork_spa.properties). 기본 리소스 번들은 <base_name>.properties와 같은 형식을 사용합니다(예: langNetwork.properties).

base_name 형식은 이 번들의 용도를 반영합니다. 예를 들어 **langMsCluster**는 리소스 번들에 MS 클러스터 작업에 사용되는 언어별 리소스가 포함되어 있음을 나타냅니다.

language_identifier 형식은 언어를 식별하는 데 사용되는 세 자로 된 머리글자어입니다. 예를 들어 rus는 러시아어를 나타내고 ger은 독일어를 나타냅니다. 이 언어 식별자는 Language 개체의 선언에 포함됩니다.

API 참조

이 섹션의 내용:

언어 클래스

이 클래스는 언어에 대한 정보(예: 리소스 번들 접미사, 가능한 인코딩 등)를 캡슐화합니다.

필드

이름	설명
locale	로캘을 나타내는 Java 개체입니다.
bundlePostfix	리소스 번들 접미사입니다. 이 접미사는 리소스 번들 파일 이름에서 언어를 식별하는 데 사용됩니다. 예를 들어 **langNetwork_ger.properties** 번들에는 **ger** 번들 접미사가 포함되어 있습니다.
charsets	이 언어를 인코딩하는 데 사용되는 문자 집합입니다. 각 언어에는 몇 개의 문자 집합이 있을 수 있습니다. 예를 들어 러시아어는 일반적으로 Cp866 및 Windows-1251 인코딩으로 인코딩됩니다.
wmiCodes	Microsoft Windows OS에서 언어를 식별하는 데 사용되는 WMI 코드의 목록입니다. 가능한 코드는 http://msdn.microsoft.com/en-us/library/aa394239(VS.85).aspx의 OSLanguage 섹션에 모두 나열되어 있습니다. OS 언어를 식별하는 방법 중 하나는

이름	설명
	WMI 클래스 OS에서 OSLanguage 속성을 쿼리하는 것입니다.
codepage	특정 언어에 사용되는 코드 페이지입니다. 예를 들어 866은 러시아어 컴퓨터에 사용되고 437은 영어 컴퓨터에 사용됩니다. OS 언어를 식별하는 방법 중 하나는 chcp 명령으로 기본 코드 페이지를 검색하는 것입니다.

executeCommandAndDecode 메서드

이 메서드는 비즈니스 논리 Jython 스크립트에서 사용하도록 되어 있으며, 디코딩 작업을 캡슐화하고 디코딩된 명령 출력을 반환합니다.

인수

이름	설명
cmd	실행할 실제 명령입니다.
keyword	디코딩 작업에 사용할 키워드입니다.
framework	DFM에서 실행 가능한 모든 Jython 스크립트에 전달되는 프레임워크 개체입니다.
timeout	명령 시간 제한입니다.
waitForTimeout	시간 제한이 초과될 때 클라이언트에서 대기할지 여부를 지정합니다.
useSudo	sudo를 사용할지 여부를 지정합니다(UNIX 컴퓨터 클라이언트에만 해당).
language	언어를 자동으로 검색하는 대신 직접 언어를 지정할 수 있습니다.

getCharsetName 메서드

이 메서드는 최근에 사용한 문자 집합의 이름을 반환합니다.

useCharset 메서드

이 메서드는 ShellUtils 인스턴스에 대한 문자 집합을 설정합니다. 그러면 초기 데이터 디코딩에 이 문자 집합이 사용됩니다.

인수

이름	설명
charsetName	문자 집합의 이름(예: windows-1251 또는 UTF-8)입니다.

"getCharsetName 메서드"(56페이지)도 참조하십시오.

getLanguageBundle 메서드

이 메서드는 올바른 리소스 번들을 가져오는 데 사용해야 합니다. 이 메서드는 다음 API를 바꿉니다.

Framework.getEnvironmentInformation().getBundle(...)

인수

이름	설명
baseName	언어 접미사가 없는 번들의 이름(예: langNetwork)입니다.
language	언어 개체입니다. ShellUtils.osLanguage를 여기에 전달해야 합니다.
framework	DFM에서 실행 가능한 모든 Jython 스크립트에 전달되는 프레임워크 공통 개체입니다.

osLanguage 필드

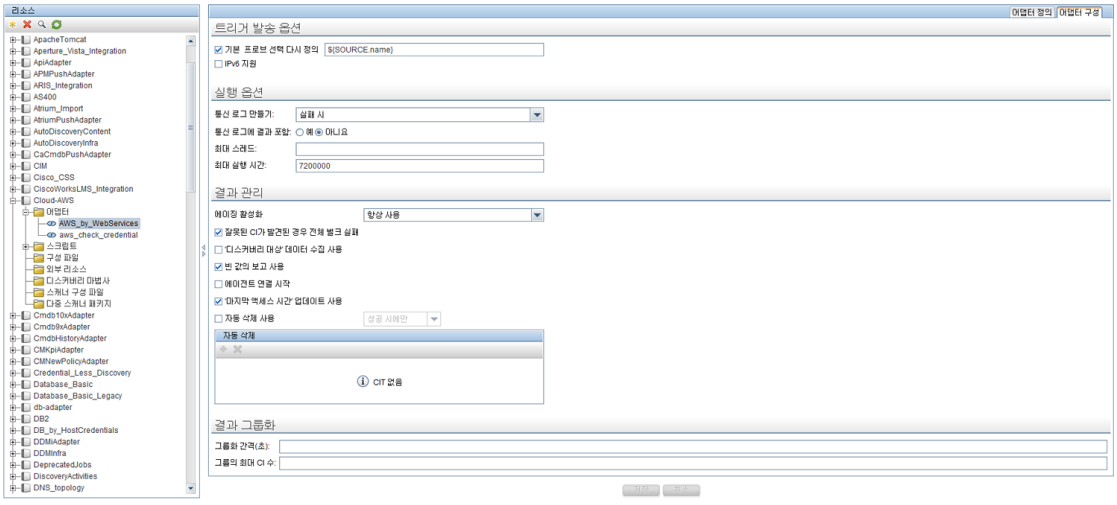
이 필드에는 언어를 나타내는 개체가 포함되어 있습니다.

DFM 코드 기록

전체 실행(모든 매개 변수 포함)을 기록하면 코드를 디버깅하고 테스트하는 경우 등에 매우 유용할 수 있습니다. 이 작업에서는 모든 관련 변수를 포함한 전체 실행을 기록하는 방법을 설명합니다. 그뿐 아니라 일반적으로 디버그 수준에서조차 로그 파일에 기록되지 않는 기타 디버그 정보를 볼 수 있습니다.

DFM 코드를 기록하려면 다음을 수행합니다.

1. **데이터 흐름 관리 > Universal Discovery**에 액세스합니다. 실행을 기록해야 하는 작업을 마우스 오른쪽 버튼으로 클릭하고 **어댑터로 이동**을 선택하여 어댑터 관리 응용 프로그램을 엽니다.
2. 아래와 같이 **어댑터 구성** 탭에서 **실행 옵션** 창을 찾습니다.

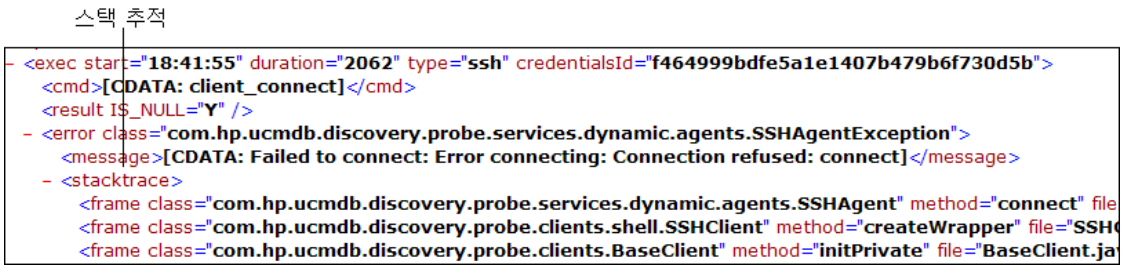


3. **통신 로그 만들기** 상자를 **항상**으로 변경합니다. 로깅 옵션을 설정하는 방법에 대한 자세한 내용은

*HP Universal CMDB 데이터 흐름 관리 안내서*에서 실행 옵션 창을 참조하십시오.

다음 예는 **셸을 통해 연결 호스팅** 작업이 실행되고 **통신 로그 만들기** 상자를 **항상** 또는 **실패 시**로 설정한 경우 생성되는 XML 로그 파일입니다.

다음 예는 메시지 및 스택 추적 매개 변수를 보여 줍니다.



Jython 라이브러리 및 유틸리티

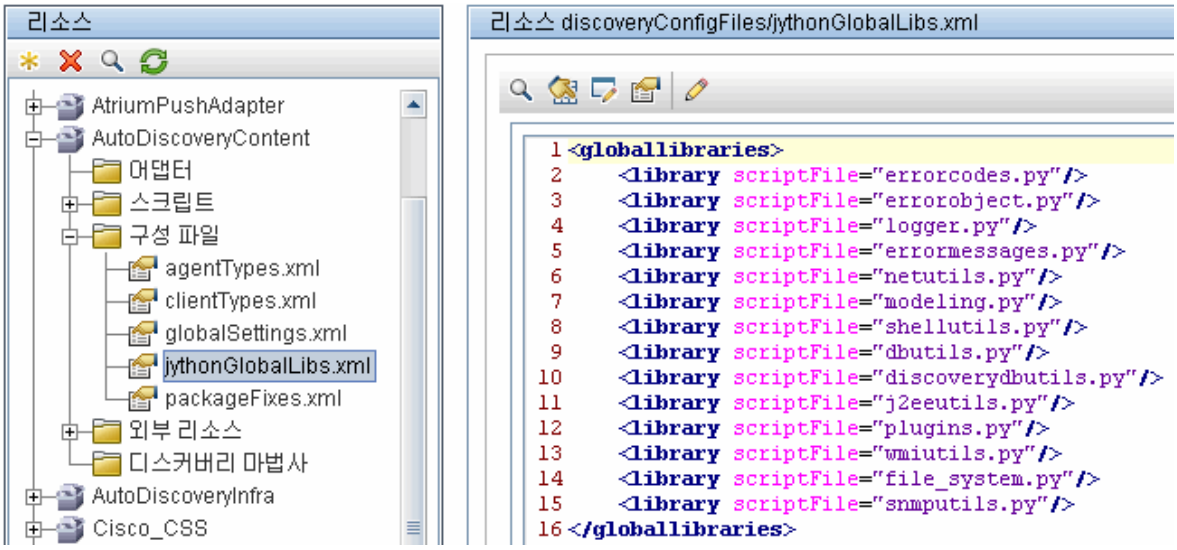
어댑터에서는 몇 가지 유틸리티 스크립트가 널리 사용됩니다. 이러한 스크립트는 AutoDiscovery 패키지의 일부이며 프로브에 다운로드된 다른 스크립트와 함께 **C:\hp\UCMDB\DataFlowProbe\runtime\probeManager\discoveryScripts** 아래에 있습니다.

참고: 프로브가 작업을 시작하면 discoveryScript 폴더가 동적으로 만들어집니다.

유틸리티 스크립트 중 하나를 사용하려면 스크립트의 import 섹션에 다음 import 줄을 추가합니다.

import <스크립트 이름>

AutoDiscovery Python 라이브러리에는 Jython 유틸리티 스크립트가 포함되어 있습니다. 이러한 라이브러리 스크립트는 DFM의 외부 라이브러리로 간주되며, **구성 파일** 폴더에 있는 jythonGlobalLibs.xml 파일에 정의되어 있습니다.



jythonGlobalLibs.xml 파일에 나타나는 각 스크립트는 프로브를 시작할 때 기본적으로 로드되므로 어댑터 정의에서 명시적으로 사용할 필요가 없습니다.

이 섹션에는 다음 항목이 포함됩니다.

- "logger.py"(59페이지)
- "modeling.py"(60페이지)
- "netutils.py"(60페이지)
- "shellutils.py"(60페이지)

logger.py

logger.py 스크립트에는 오류 보고를 위한 로그 유틸리티 및 helper 함수가 포함되어 있습니다. 이 스크립트의 디버그, 정보 및 오류 API를 호출하여 로그 파일에 쓸 수 있습니다. 로그 메시지는 **C:\hp\UCMDB\DataFlowProbe\runtime\log**에 기록됩니다.

메시지는 **C:\hp\UCMDB\DataFlowProbe\conf\log\probeMgrLog4j.properties** 파일에서 PATTERNS_DEBUG 어펜더에 대해 정의된 디버그 수준에 따라 로그 파일에 입력됩니다. 기본 수준은 DEBUG입니다. 자세한 내용은 "오류 심각도 수준"(64페이지)을 참조하십시오.

```
#########################################################################
#################     PATTERNS_DEBUG log          ####################
#########################################################################
log4j.category.PATTERNS_DEBUG=DEBUG, PATTERNS_DEBUG
log4j.appender.PATTERNS_DEBUG=org.apache.log4j.RollingFileAppender
log4j.appender.PATTERNS_DEBUG.File=C:\hp\UCMDB\DataFlowProbe\runtime\log/probeMgr-
patternsDebug.log
log4j.appender.PATTERNS_DEBUG.Append=true
log4j.appender.PATTERNS_DEBUG.MaxFileSize=15MB
log4j.appender.PATTERNS_DEBUG.Threshold=DEBUG
log4j.appender.PATTERNS_DEBUG.MaxBackupIndex=10
```

```
log4j.appender.PATTERNS_DEBUG.layout=org.apache.log4j.PatternLayout
log4j.appender.PATTERNS_DEBUG.layout.ConversionPattern=<%d> [%-5p] [%t] - %m%n
log4j.appender.PATTERNS_DEBUG.encoding=UTF-8
```

명령 프롬프트 콘솔에는 정보 및 오류 메시지도 나타납니다.

API에는 다음 두 가지 집합이 있습니다.

- logger.<debug/info/warn/error>
- logger.<debugException/infoException/warnException/errorException>

첫 번째 집합은 적절한 로그 수준에서 모든 문자열 인수의 연결을 발급하고 두 번째 집합은 연결을 발급할 뿐 아니라 최근에 발생한 예외의 스택 추적도 발급하여 더 자세한 정보를 제공합니다. 예:

```
logger.debug('found the result')
logger.errorException('Error in discovery')
```

modeling.py

modeling.py 스크립트에는 호스트, IP, 프로세스 CI 등을 만드는 API가 포함됩니다. 이러한 API를 사용하면 공통 개체를 만들 수 있고 코드의 가독성을 향상시킬 수 있습니다. 예:

```
ipOSH= modeling.createIpOSH(ip)
host = modeling.createHostOSH(ip_address)
member1 = modeling.createLinkOSH('member', ipOSH, networkOSH)
```

netutils.py

netutils.py 라이브러리는 네트워크 및 TCP 정보를 검색(예: 운영 체제 이름 검색, MAC 주소의 유효성 확인, IP 주소의 유효성 확인 등)하는 데 사용됩니다. 예:

```
dnsName = netutils.getHostName(ip, ip)
isValidIp = netutils.isValidIp(ip_address)
address = netutils.getHostAddress(hostName)
```

shellutils.py

shellutils.py 라이브러리는 셸 명령을 실행하고 실행된 명령의 끝 상태를 검색할 수 있는 API를 제공하고, 해당 끝 상태에 따라 다중 명령을 실행할 수 있도록 합니다. 라이브러리는 셸 클라이언트에서 초기화되고 이 클라이언트를 사용하여 명령을 실행하고 결과를 검색합니다. 예:

```
ttyClient = Framework.createClient(Framework.getTriggerCIData(BaseClient.CREDENTIALS_ID), Props)
clientShUtils = shellutils.ShellUtils(ttyClient)
if (clientShUtils.isWinOs()):
    logger.debug ('discovering Windows..')
```

3장: 오류 메시지

이 장의 내용:

오류 메시지 개요

디스커버리 중에 연결 실패, 하드웨어 문제, 예외, 시간 제한 등의 많은 오류가 발견될 수 있습니다. 이러한 오류는 정규 디스커버리 흐름이 실패할 때마다 Universal Discovery 창에 표시됩니다. 문제의 원인이 된 트리거 CI에서 드릴다운하여 오류 메시지 자체를 볼 수 있습니다.

DFM은 상황에 따라 무시할 수 있는 오류(예:도달할 수 없는 호스트)와 조치를 취해야 할 오류(예: 자격 증명 문제 또는 구성 파일이나 DLL 파일 없음)를 구분합니다. 뿐만 아니라 DFM은 연속 실행 시 같은 오류가 발생할 경우에도 오류를 한 번만 보고하고 오류가 한 번만 발생해도 오류를 보고합니다.

패키지를 만들 때 적절한 메시지를 패키지에 리소스로 추가할 수 있습니다. 패키지를 배포할 때 메시지도 올바른 위치에 배포됩니다. 메시지는 "오류 작성 규칙"(61페이지)에 설명된 대로 규칙을 준수해야 합니다.

DFM은 다국어 오류 메시지를 지원합니다. 작성하는 메시지를 해당 지역의 언어로 표시되도록 지역화할 수 있습니다.

오류 검색에 대한 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 "디스커버리 진행률 및 결과"를 참조하십시오.

통신 로그를 설정하는 방법에 대한 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 "실행 옵션 창"을 참조하십시오.

오류 작성 규칙

- 각 오류는 오류 메시지 코드 및 인수 배열(**int, String[]**)로 식별됩니다. 메시지 코드와 인수 배열의 조합으로 특정 오류를 정의합니다. 매개 변수의 배열은 null일 수 있습니다.
- 각 오류 코드는 고정 문자열인 **간단한 메시지**와 0개 이상의 인수가 포함된 템플릿 문자열인 **세부 메시지**에 매핑됩니다. 템플릿의 인수 개수와 실제 매개 변수 개수가 일치하는 것으로 간주됩니다.

오류 메시지 코드의 예:

10234는 다음과 같이 간단한 메시지로 오류를 나타낼 수 있습니다.

Connection Error

다음과 같이 세부 메시지로 오류를 나타낼 수도 있습니다.

Could not connect via {0} protocol due to timeout of {1} msec

여기서 각 항목은 다음과 같습니다.

{0} = 첫 번째 인수: 프로토콜 이름

{1} = 두 번째 인수: 시간 제한 길이(밀리초)

이 섹션에는 다음 항목도 포함됩니다.

속성 파일 컨텐츠

속성 파일에는 각 오류 메시지 코드의 키 두 개가 포함되어야 합니다. 예를 들어 오류 45의 경우 다음과 같은 키를 포함해야 합니다.

- **DDM_ERROR_MESSAGE_SHORT_45**. 간단한 오류 설명입니다.
- **DDM_ERROR_MESSAGE_LONG_45**. 긴 오류 설명으로, 매개 변수(예: {**0**},{**1**})를 포함할 수 있습니다.

오류 메시지 속성 파일

속성 파일에는 오류 메시지 코드와 두 메시지(간단한 메시지와 세부 메시지) 간의 맵이 포함되어야 합니다.

속성 파일이 배포되고 나면 해당 데이터와 기존 데이터가 병합됩니다. 즉, 새 메시지 코드는 추가되고 이전 메시지 코드는 다시 정의됩니다.

인프라 속성 파일은 **AutoDiscoveryInfra** 패키지에 포함됩니다.

로캘 이름 지정 규칙

- 기본 로캘의 경우: **<파일 이름>.properties.errors**
- 특정 로캘의 경우: **<파일 이름>_xx.properties.errors**

  여기서 **xx**는 로캘(예: **infraerr_fr.properties.errors** 또는 **infraerr_en_us.properties.errors**)입니다.

오류 메시지 코드

다음 오류 코드는 HP Universal CMDB에 기본적으로 포함됩니다. 이 목록에 직접 작성한 오류 메시지를 추가할 수 있습니다.

오류 이름	오류 코드	설명
내부	100-199	대부분 Jython 스크립트를 실행하는 동안 발생한 예외에서 확인됨
연결	200-299	연결 실패, 대상 컴퓨터에 에이전트 없음, 대상에 도달할 수 없음 등
자격 증명 관련	300-399	사용 권한이 거부됨, 자격 증명이 없어 연결 시도가 차단됨
시간 제한	400-499	연결/명령 수행 중 시간 제한 초과
예기치 못한 동작 또는 잘못된 동작	500-599	구성 파일 없음, 예기치 못한 중단 등
정보 검색	600-699	대상 컴퓨터에 정보 없음, 에이전트에서 정보 쿼리 실패 등
리소스 관련	700-799	메모리 부족과 관련된 오류 또는 클라이언트가 제대로 릴리스되지 않음
구문 분석	800-899	텍스트 구문 분석 오류
인코딩	900	입력 오류, 지원되지 않는 인코딩
SQL 관련	901-903, 924	SQL 작업에서 받은 오류
HTTP 관련	904-909	HTTP 연결 중에 생성된 오류, HTTP 오류 코드에서 구문 분석된 오류
특정 응용 프로그램	910-923	응용 프로그램별 문제로 인해 보고된 오류(예: 잘못된 LSOF 버전, 대기열 관리자 없음 등)

분류되지 않은 컨텐츠 오류

오래된 컨텐츠를 퇴행 없이 지원하기 위해 응용 프로그램 및 SDK 관련 메서드에서 메시지 코드 100 오류(즉, 분류되지 않은 스크립트 오류)를 다르게 처리합니다.

이러한 오류는 메시지 코드별로 그룹화되지 않고(즉, 같은 유형의 오류로 간주되지 않음) 메시지 컨텐츠별로 그룹화됩니다. 즉, 스크립트에서 오래되어 사용되지 않는 메서드(메시지 문자열은 있고 오류 코드는 없음)로 오류를 보고하면 모든 메시지가 같은 오류 코드를 받지만 응용 프로그램 또는 SDK 관련 메서드에서는 메시지가 다르면 다른 오류로 표시됩니다.

프레임워크의 변경 내용

(com.hp.ucmdb.discovery.library.execution.BaseFramework)

다음 메서드가 인터페이스에 추가되었습니다.

- void reportError(int msgCode, String[] params);
- void reportWarning(int msgCode, String[] params);
- void reportFatal(int msgCode, String[] params);

다음의 오래된 메서드는 이전 버전과의 호환을 위해 여전히 지원되지만 사용되지 않음으로 표시됩니다.

- void reportError(String message);
- void reportWarning (String message);
- void reportFatal (String message);

오류 심각도 수준

어댑터가 트리거 CI에 대한 실행을 마치면 상태를 반환합니다. 오류 또는 경고가 보고되지 않으면 상태는 성공입니다.

심각도 수준은 가장 좁은 범위에서 넓은 범위 순으로 나열됩니다.

치명적인 오류

이 수준은 다음과 같이 인프라 문제, DLL 파일 없음 또는 예외 등 심각한 오류를 보고합니다.

- **작업 생성 실패(프로브 없음, 변수 없음 등)**
- **스크립트를 실행할 수 없음**
- **서버에서 결과를 처리하지 못하여 CMDB에 데이터를 쓰지 못함**

오류

이 수준은 DFM이 데이터를 검색하지 못하는 문제를 보고합니다. 일반적으로 이러한 오류에는 조치(예: 시간 제한 연장, 범위 변경, 매개 변수 변경, 다른 사용자 자격 증명 추가 등)가 필요하므로 철저히 검토합니다.

- **사용자 간섭이 필요할 수 있는 경우(예: 추가 조사가 필요할 수 있는 네트워크 문제 또는 자격 증명 문제)에는 오류가 보고됩니다. 이러한 오류는 디스커버리의 오류가 아니라 구성의 오류입니다.**
- **보통 디스커버리된 컴퓨터 또는 응용 프로그램의 예기치 못한 동작으로 인한 내부 오류(예: 구성 파일 없음 등)**

경고

실행에 성공했지만 사용자가 알고 있어야 하는 심각하지 않은 문제가 있을 수 있습니다. 이 경우 DFM에서는 심각도를 경고로 표시합니다. 더 자세한 디버깅 세션을 시작하기 전에 이러한 CI를 보고 데이터가

없는지 확인해야 합니다. **경고**에는 원격 호스트에 에이전트가 설치되지 않았거나, 잘못된 데이터로 인해 특성이 잘못 계산되었다는 등의 메시지가 포함될 수 있습니다.

- 연결 에이전트(SNMP, WMI) 없음
- 디스커버리에 성공했지만 사용 가능한 모든 정보가 디스커버리되지 않음

4장: 소비자-공급자 종속 관계 매핑

이 장의 내용:

종속 관계 디스커버리 개요

종속 관계 매핑은 배포 가능 구성 요소 또는 실행 중인 소프트웨어 간의 관계를 디스커버리하는 유연한 메서드를 제공합니다. 이 메서드를 사용하면 Universal Discovery 프로세스에서 종속 관계를 자동으로 디스커버리하는 데 사용하는 사용자 정의 종속 관계 매핑 규칙을 간단한 프로그래밍 구문을 통해 사용할 수 있습니다.

서비스는 비즈니스 또는 IT 서비스일 수 있습니다. 비즈니스 서비스는 비즈니스에서 다른 비즈니스에 제공하거나(B2B) 비즈니스 내의 한 조직이 다른 조직에 제공하는(예: 결제 처리) 서비스입니다. IT 서비스는 IT 조직에서 비즈니스 서비스 또는 IT 자체 운영을 지원하기 위해 제공하는 비즈니스 서비스입니다.

배포 가능 구성 요소는 실행 중인 소프트웨어에 배포되는 소프트웨어 구성 요소(예: 응용 프로그램 서버 또는 웹 서버)입니다. 배포 가능 구성 요소의 예로는 JEE EAR 구성 요소 또는 Oracle 데이터베이스 내의 스키마가 있습니다. 종속 관계 디스커버리에서는 실행 중인 소프트웨어를 배포 가능 구성 요소로 간주합니다.

공급자 배포 가능 구성 요소는 서비스를 제공하고 기타 배포 가능 구성 요소에서 해당 서비스를 사용하는 방법을 선언합니다. **소비자** 배포 가능 구성 요소는 공급자 배포 가능 구성 요소에서 제공하는 서비스를 "사용"합니다. 이러한 배포 가능 구성 요소 간의 종속 관계가 **소비자-공급자** 종속 관계입니다.

참고:

- 소비자-공급자 종속 관계 어댑터를 만들고 종속 관계 매핑 프레임워크를 사용하기 위해서는 설치하는 동안 UCMDB 스키마를 설정할 때 **검색 사용** 옵션을 선택해야 합니다.
- 소비자-공급자 종속 관계 어댑터는 결합 모드의 Data Flow Probe에서만 실행할 수 있습니다.

자세한 내용은 다음을 참조하십시오.

- "공급자 및 소비자"(67페이지)
- "종속 관계 서명"(67페이지)
- "종속 관계 매핑 흐름"(68페이지)

공급자 및 소비자

연결 문자열을 사용하여 공급자에 연결합니다. 예를 들어, Oracle 데이터베이스가 공급자인 경우 해당 서비스에 연결하려면 다음이 필요합니다.

- 시스템의 IP 주소
- SID
- TCP 포트

위 세 가지 정보는 공급자가 제공하는 서비스에 연결해야 하는 소비자에게 필요한 연결 문자열을 구성합니다. 예를 들어 Oracle 연결 문자열에 다음과 같은 정보가 포함될 수 있습니다.

- IP 주소: 1.1.1.1, 2.2.2.2
- 포트: 1521
- SID: abcd

소비자는 공급자의 연결 문자열을 하나 이상 알고 있으며, 이 연결 문자열은 구성 문서, 데이터베이스 테이블, Windows 레지스트리 등과 같이 알려진 위치에 있습니다. 해당 위치를 검색하면 소비자와 공급자 사이의 종속 관계를 디스커버리할 수 있습니다.

공급자의 연결 문자열이 특정 구성 문서에 있는 경우 공급자와 구성 문서의 컨테이너가 소비자-공급자 관계로 연결됩니다.

그러면 소비자-공급자 종속 관계를 디스커버리하는 프로세스가 간단해져 소비자의 구성 문서에서 공급자의 연결 문자열을 검색하고 해당 검색 결과에는 지정된 공급자의 소비자가 보유한 모든 구성 문서가 포함됩니다.

자세한 내용은 "종속 관계 정의"(74페이지)를 참조하십시오.

종속 관계 서명

각 구성 문서 및 공급자 유형에 대해 다른 검색 용어를 사용할 수 있습니다. 이러한 검색 용어는 종속 관계 서명 파일에 정의되어 있습니다.

종속 관계 서명은 공급자의 연결 문자열과 소비자의 구성 문서를 사용하여 지정된 공급자 및 소비자 간의 소비자-공급자 종속 관계가 존재하는지 여부를 정의하는 규칙입니다.

종속 관계 서명은 검색식으로 구성됩니다. 이러한 검색식은 특정 공급자에 대한 실제 값이 아닌, 연결 문자열의 정의에 따라 달라집니다. 또한 검색식은 파일의 실제 내용이 아닌, 소비자의 구성 문서 이름, 위치 및 형식에 따라서도 달라집니다. 공급자의 연결 문자열이 알려진 경우 이러한 연결 문자열이 검색식에 삽입되어 구체 검색식이 생성됩니다. 그런 다음 소비자의 구성 문서를 사용하여 구체 검색식이 평가됩니다. 검색식은 소비자의 구성 문서에 공급자의 연결 문자열이 특정 방식으로 존재하는 경우에만 "True"를 반환합니다.

자세한 내용은 "종속 관계 서명 파일"(68페이지)을 참조하십시오.

종속 관계 매핑 흐름

이 섹션에서는 다음과 같이 종속 관계 매핑 중 발생하는 기본 흐름에 대한 간략한 개요를 제공합니다.

1. 배포 가능 구성 요소 및 해당 연결 문자열을 디스커버리합니다.
2. 각 유형의 공급자 배포 가능 구성 요소에서 특정 종속 관계 매핑 작업을 트리거합니다. 각 작업의 어댑터는 특정 배포 가능 구성 요소 유형의 관련 연결 문자열을 추출하는 방법을 알고 있습니다. 어댑터에서 서비스를 사용하는 기타 배포 가능 구성 요소를 검색합니다.
3. 발견된 각 배포 가능 구성 요소와 해당 트리거(공급자) 사이에 소비자-공급자 관계가 생성됩니다.

종속 관계 서명 파일

이 섹션의 내용:

종속 관계 서명 파일의 구조

종속 관계 서명 파일은 배포 가능 구성 요소 간 종속 관계를 하나 이상 정의합니다.

종속 관계의 소비자 부분은 <Deployable> 요소로 정의됩니다. 각 <Deployable> 요소에는 <Dependency> 요소가 하나 이상 포함될 수 있습니다. 각 <Dependency> 요소에는 소비자(<Deployable> 요소)와 공급자 사이에 종속 관계가 존재하는 조건이 포함됩니다.

<Dependency> 요소는 공급자 CI 및 소비자의 구성 문서를 해당 입력으로 간주하고, 이 두 CI 사이에 소비자에서 공급자로의 종속 관계가 존재하는 경우에만 "True"를 반환하는 부울 함수로 볼 수 있습니다.

<Dependency> 요소에서 공급자는 해당 CI 유형을 사용하는 파일에서만 식별됩니다. 각 종속 관계는 공급자의 단일 CI 유형에 사용될 수도 있습니다. 다음 예에서는 실행 중인 소프트웨어에 해당하는 소비자와 실행 중인 소프트웨어에 해당하는 공급자 간의 종속 관계 함수를 정의합니다.

```
<Deployable name="ApolloOnNod">
  <Descriptor cit="running_software">
  </Descriptor>
  <Dependency name="history_db" providerCiType="running_software" scope="default">
    ...
```

변수 및 개념

변수에는 프로그램이 실행되는 동안 다양한 값이 포함될 수 있습니다. 종속 관계 매핑의 경우 다양한 종속 관계 검색 실행을 위해 변수에 다양한 값이 포함될 수 있습니다. 이러한 변수는 검색식을 정의할 때 구

성 문서에 연결 문자열이 있는지 여부를 결정하는 데 사용됩니다. 연결 문자열은 공급자에 따라 다르므로 연결 문자열의 특정 값과 관계없이 변수를 사용하여 일반 검색식을 정의할 수 있습니다.

Variables

참고: "변수"라는 용어는 변수 및 개념 변수 모두를 가리킵니다.

변수를 사용하기 위한 구문은 ${VARIABLE_NAME}입니다. 각 변수 값은 문자열 값 또는 문자열 목록이어야 합니다. 예를 들어, 연결 문자열의 일부가 공급자의 IP 주소일 수 있습니다. 구성 문서에서 IP 주소를 검색하려면 IP_ADDRESS라는 변수를 정의하여 ${IP_ADDRESS}와 같은 방식으로 식에 사용할 수 있습니다.

먼저 변수를 전체 종속 관계 서명 파일의 범위(글로벌 범위라고도 함) 또는 종속 관계 범위(로컬 범위라고도 함)에 정의해야 합니다. 변수를 정의하면 해당 변수를 사용할 수 있습니다.

변수를 정의하지 않고 사용하면 서명 파일을 배포할 때 오류가 발생합니다.

글로벌 범위

글로벌 범위 변수는 해당 변수가 정의된 파일의 모든 종속 관계에서 사용할 수 있습니다. 글로벌 범위 변수를 정의하려면 다음을 수행합니다.

```
<DependencySignatures xmlns="http://www.hp.com/ucmdb/1-0-0/Dependencies">
  <VariableDeclarations>
    <Variable name="IPADDRESS"/>
    <Variable name="PROTOCOL"/>
</DependencySignatures>
```

위 예에서는 두 개의 글로벌 범위 변수인 IPADDRESS 및 PROTOCOL을 정의합니다. 모든 변수와 마찬가지로 문자열 값이나 문자열 목록을 포함할 수 있습니다.

글로벌 변수 값은 트리거의 대상 데이터로만 설정할 수 있습니다.

로컬 범위

로컬 범위 변수는 해당 변수가 정의된 종속 관계에서만 사용할 수 있습니다. 동일한 배포 가능 구성 요소에 해당하는지 여부에 관계없이 다른 종속 관계에는 표시되지 않습니다.

서로 다른 종속 관계에 이름이 같은 로컬 범위 변수를 여러 개 정의할 수 있습니다. 각 변수는 완전히 별도의 변수이며 해당 값은 변수가 정의된 범위에서만 사용될 수 있습니다.

참고: 글로벌 범위 변수와 이름이 같은 로컬 범위 변수나 동일한 종속 관계 컨텍스트에서 이름이 같은 두 개의 로컬 범위 변수는 정의하지 못할 수 있습니다.

로컬 범위 변수를 정의하려면 다음 구문을 사용합니다.

```
<Deployable name="StrongXmlApplication">
  <Descriptor cit="cluster_software"/>
```

```
  <Dependency name="app_cluster" providerCiType="running_software" scope="default">
    <VariableDeclarations>
      <Variable name="IPADDRESS"/>
      <Variable name="PROTOCOL"/>
    </VariableDeclarations>
    ...
  </Dependency>
</Deployable>
```

로컬 범위 변수는 삽입문을 사용해서만 값을 할당할 수 있습니다. 값이 추출되는 파일 유형에 따라 서로 다른 특정 삽입문이 있습니다. 삽입문은 구성 문서의 두 위치에 표시될 수 있습니다.

- 전용 <Variables> 섹션. <Variables> 섹션의 삽입문은 전체 파일의 검색식이 True로 평가된 경우에만 평가됩니다.
- 검색 조건의 일부인 <Variables> 섹션. 이 옵션에서는 검색 조건이 True로 평가된 경우에만 변수가 삽입됩니다. 대체 식을 사용할 때는 지원되지 않습니다. 자세한 내용은 "변수의 기본값 사용"(76페이지)을 참조하십시오.

변수 할당에 대한 자세한 내용은 다음을 참조하십시오.

- "속성 구성 문서"(83페이지)
- "XML 구성 문서"(87페이지)
- "텍스트 구성 문서"(90페이지)

개념

검색식은 관련된 각 구성 문서에 적용됩니다. 연결 문자열의 일부 값은 서로 밀접하게 결합되어 있으며, 구체 검색식에서 서로와 함께 사용해야만 합니다. 각각의 그러한 결합된 연결 문자열 집합은 고유 이름을 가지며 개념이라고 합니다.

예를 들어, 공급자 배포 가능 구성 요소를 1.1.1.1:8080 또는 2.2.2.2:85에서 액세스할 수 있다고 하겠습니다. 그러면 해당 공급자의 모든 소비자는 구성 파일 중 하나에 1.1.1.1:8080 또는 2.2.2.2:85가 있어야 합니다. 그러나 이 공급자는 1.1.1.1:85에서 수신하지 않으므로 해당 공급자의 소비자 구성 문서에 1.1.1.1:85가 있을 가능성이 낮습니다. 소비자의 구성 문서 중 하나에 1.1.1.1:85가 있더라도 이는 이 공급자와의 종속 관계가 아니라, 공급자와 동일한 노드에서 실행 중이며 1.1.1.1:85에서 수신 중인 일부 다른 배포 가능 구성 요소와의 종속 관계를 나타냅니다.

검색은 정확하게 일치하는 IP 주소 및 포트를 찾아야 합니다. 그러지 않으면 가양성 종속성이 표시될 수 있기 때문입니다.

또한 각 IP 주소에 이름이 여러 개 있을 수 있으므로(예: 하나의 신뢰할 수 있는 DNS 이름 및 여러 개의 별칭 이름) 검색에서는 각 IP 주소 및 포트의 이름이 정확하게 일치하는 이름을 찾아야 합니다.

따라서 해당 공급자의 소비자 구성 문서에 XYZ:8080, FOO:8080 또는 ABC:85(여기서, XYZ, FOO 및 ABC는 공급자 주소의 다른 이름)와 같이 일치하는 항목 또는 쌍이 있을 가능성이 있습니다. 이러한

일치는 공급자와의 종속 관계를 나타내기 때문입니다. 그러나 일치 항목 ABC:8080은 해당 공급자와의 종속 관계를 나타내지 않으므로 그러한 일치 항목이 검색되어서는 안 됩니다.

이를 위해 개념을 사용합니다. 다른 특성과 함께 사용되어야 하는 각 연결 문자열 특성을 동일한 개념에 정의해야 합니다. 각 연결 문자열 특성은 해당 개념의 변수여야 합니다.

개념은 글로벌 범위에만 정의할 수 있습니다. 밀접하게 결합된 연결 문자열 집합을 나타내는 각 개념 인스턴스에는 변수와 유사하게 어댑터에서 설정한 값이 있어야 합니다.

각 개념은 정확히 하나의 키 변수와 추가 변수로 구성됩니다. 이러한 각 변수(키 변수 및 추가 변수)는 밀접하게 결합된 연결 문자열을 저장하는 데 사용됩니다. 키 변수는 동일한 개념의 다양한 인스턴스를 구분하는 데 사용합니다. 따라서 키 변수에 같은 값을 가지는 개념 인스턴스가 동일한 개념에 두 개 있을 수 없습니다. 자세한 내용은 "개념 변수 값 지정"(101페이지)을 참조하십시오.

참고: 키 변수를 기타 다른 개념 변수처럼 사용할 수 있습니다. 중요한 것은 트리거를 실행하는 동안 개념 인스턴스를 인스턴스화하는 방법입니다.

개념 및 해당 변수를 정의하려면 다음 구문을 사용합니다.

```
<Concept name="ConceptName">
  <속성>
    <KeyProperty name="KeyVariableName"/>
    <Property name="VariableName1"/>
    <Property name="VariableName2"/>
  </Properties>
</Concept>
```

개념의 변수를 검색식에 사용하려면 다음 구문을 사용합니다.

```
${ConceptName.VariableName}.
```

검색식에 사용되는 각 개념에 대해 개념의 관련 변수는 모두 포함하고 다른 개념의 변수는 제외하는 논리 연산자가 있어야 합니다.

다음은 개념이 포함된 올바른 검색식의 예입니다.

```
${C.A} = X AND ${X.B} = Y

(${C1.A} = X AND ${C1.B} = Y) OR ${C2.C} = Z
```

다음은 잘못된 검색식의 예입니다.

```
${C1.A} = X AND ${C2.B} = Y AND ${C2.X} = Z
```

기본값

글로벌 변수 및 개념 변수에는 기본값을 선택적으로 사용할 수 있습니다. 어댑터에서 변수에 기본값과 동일한 값을 삽입하는 경우 대체 검색식을 정의할 수 있습니다. 대체 검색식에 대한 자세한 내용은 "검색식 구성"(74페이지)을 참조하십시오.

기본값을 정의하려면 다음 구문을 사용합니다.

```
<Variable name=”PORT” defaultValue=”8080” />
```

IP 주소 변수 유형

IP 주소는 중요한 연결 문자열 유형입니다. 그러나 공급자 주소가 구성 파일에 명시되지 않는 경우도 있습니다.

이에 대한 일반적인 예는 소비자와 공급자가 동일한 호스트 컴퓨터에 있을 때입니다 이 경우 공급자 주소 대신 "localhost" 또는 "127.0.0.1"과 같은 문자열이 표시되거나 주소가 전혀 표시되지 않을 수 있습니다.

변수를 **IP 주소** 유형으로 표시하면 프레임워크에서 IP 주소 변수를 무시하고 연결 문자열의 나머지 부분을 사용하여 동일한 호스트에 있는 공급자를 찾아 자동으로 로컬 종속 관계를 찾습니다.

변수를 IP 주소로 표시

- 글로벌 변수의 경우

```
<DependencySignatures xmlns="http://www.hp.com/ucmdb/1-0-0/Dependencies">
  <VariableDeclarations>
    <Variable name="MY_VARIABLE" type=”IP Address” />
  </VariableDeclarations>
      ...
</DependencySignatures>
```

- 개념 변수의 경우

```
<Concept name="IpEndpoint">
  <속성>
    <KeyProperty name="PORT"/>
    <Property name="MY_VARIABLE" type=”IP Address” />
  </Properties>
</Concept>
```

참고: 로컬 변수는 IP 주소 유형일 수 없습니다.

소비자의 설명자 정의

소비자-공급자 종속 관계를 정의하기 위해 검색 어댑터는 공급자의 연결 문자열이 사용된 구성 문서가 있는 소비자를 찾습니다. 그러나 때로는 소비자의 구성 문서에 연결 문자열이 있거나 배포 가능 구성 요소의 속성에 따라 다양한 출력 변수가 있는 경우에도 특정 서비스의 소비자일 가능성이 있는 배포 가능 구성 요소를 제한해야 할 수 있습니다.

이를 위해 <Descriptor> 요소를 사용하여 CI 유형만이 아니라 보다 자세히 배포 가능 구성 요소를 설명할 수 있습니다. 다음을 사용하여 배포 가능 구성 요소를 설명할 수 있습니다.

- CI 유형. 즉, 종속 관계는 "J2EE 응용 프로그램" 유형의 배포 가능 구성 요소에만 관련이 있습니다(필수).
- 문자열 특성에 대한 조건. 예를 들어 "J2EE 응용 프로그램" 유형의 배포 가능 구성 요소에 대해 응용 프로그램 이름이 "MyShop"이라는 조건입니다. 조건은 등식만을 테스트할 수 있습니다(선택 사항).
- 다른 CI에 대한 필수 Composition 링크. 설명자에 Composition 링크가 지정되어 있는 경우 지정된 유형의 CI에 대한 Composition 링크가 배포 가능 구성 요소에 있어야 합니다(선택 사항).
- 연결된 CI의 문자열 특성에 대한 조건. 이전 옵션에서와 같이 지정된 경우에 해당합니다(선택 사항).
- 배포 가능 구성 요소가 포함된 노드 경로(TQL 쿼리). 쿼리 이름은 **nodeToDeployableQuery** 특성을 사용하여 지정됩니다. 노드가 배포 가능 구성 요소에 Composition 링크로 직접 연결된 경우 TQL 쿼리를 경로로 표시하지 않아도 됩니다. 그러나 배포 가능 구성 요소를 설명하는 데 노드를 계속 사용하는 경우 <ConnectedCICondition> 태그를 사용하여 이를 표시해야 합니다. 배포 가능 구성 요소의 상위 계층 구조에 포함 노드가 없는 경우 hasContainingNode = “false” 특성을 사용하여 이를 표시해야 합니다. 경로를 표시할 때는 hasContainingNode = “true”도 추가해야 합니다. 자세한 내용은 "TQL 쿼리 정의" (95페이지)를 참조하십시오(선택 사항).

참고: 배포 가능 설명자 또는 연결된 CI에서는 STRING 특성 유형만 지원됩니다. 해당 특성은 정적일 수 없으며 계산된 특성이 포함될 수 없습니다.

다음은 CI 유형만 표시하는 설명자 예입니다.

```
<Deployable name="ApolloOnNode_2">
  <Descriptor cit="running_software">
  </Descriptor>
    ...
```

다음은 문자열 특성이 있는 설명자 예입니다.

```
<Deployable name="ApolloOnCluster">
  <Descriptor cit="node">
    <Attribute name="default_gateway_ip_address_type" value="IPv6" />
  </Descriptor>
    ...
```

다음은 대/소문자를 구분하지 않는 등식을 테스트하여 연결된 CI가 있는 설명자의 예입니다.

```
<Deployable name="MyRunningSoftware">
  <Descriptor cit="running_software">
    <ConnectedCiCondition cit=”node” linkType=”composition” isDirectionForward=”true”>
      <Attribute name=”name” value=”MyNode” operator=”equalIgnoreCase” />
    </ConnectedCiCondition>
  </Descriptor>
    ...
```

참고: ConnectedCiCondition의 경우 linkType="composition" 및 isDirectionForward="true"만 사용해야 할 수 있습니다.

종속 관계 정의

모든 소비자에 대해 여러 종속 관계를 정의할 수 있습니다. 각 종속 관계는 특정 공급자 CI 유형과 연관되어 있습니다. 공급자 CI의 종속 관계 서명을 평가하는 동안 해당 공급자의 CI 유형과 연관된 모든 종속 관계가 평가됩니다. 각 종속 관계는 하나 이상의 구성 문서에 있는 검색식을 가집니다. 검색식이 True로 평가되면 소비자와 공급자 간 종속 관계(소비자-공급자 관계)가 존재하는 것입니다. 동일한 소비자와 공급자 CI 유형 사이에 여러 종속 관계가 존재하고 True로 평가되는 경우에도 관계는 하나만 만들어집니다.

다음은 종속 관계 구문의 예입니다.

```
<Deployable name="Websphere J2EE Application">
  <Descriptor cit="j2eeapplication"/>
  <Dependency name="J2EE Application to DB by JNDI" providerCiType="oracle" scope="default">
    ...
```

또한 모든 종속 관계는 범위를 가집니다. 범위는 해당 설명자를 따르는 모든 소비자 중 특정 종속 관계와 관련이 있는 소비자입니다. 자세한 내용은 "소비자의 설명자 정의"(73페이지)를 참조하십시오.

검색식 구성

참고: 배포 가능 설명자 또는 연결된 CI에서는 STRING 특성 유형만 지원됩니다. 해당 특성은 정적일 수 없으며 계산된 특성이 포함될 수 없습니다.

검색식은 논리 연산자와 조건으로 구성됩니다. 지원되는 논리 연산자는 "And" 및 "Or"입니다.

조건은 공급자의 연결 문자열 및 소비자의 구성 문서를 사용하여 평가되는 식입니다. 조건은 파일 유형에 따라 다릅니다. 즉, 속성 구성 문서와 XML 문서의 조건은 다릅니다.

다음은 속성 구성 문서에 대한 검색식의 예로, 공급자의 IP 주소가 소비자의 **MyConfig.properties** 구성 문서의 IPADDRESS 키에 존재하는 경우에만 공급자와 소비자 간 종속성을 만듭니다. 종속 관계 구성 문서에 다음과 같이 작성됩니다.

```
<PropertiesConfigurationDocument name="MyConfig.properties"> (Type and name of the
```

```
configuration document)
  <Condition> (Beginning of the search expression)
    <Operator type="and"> (Operator)
      <KeyCondition key="IPADDRESS"> (Start a condition and state its type)
        <Values>
          <Value>${IP_ADDRESS}</Value> (Use a variable stating the provider's IP address)
        </Values>
      </KeyCondition>
    </Operator>
  </Condition>
</PropertiesConfigurationDocument>
```

참고: <Condition> 요소에는 <Operator> 요소가 하나 이상 있어야 하며 위 예와 같이 검색 조건이 하나뿐이므로 논리적으로 불필요한 경우에도 마찬가지입니다.

더 복잡한 검색식을 살펴보겠습니다. 다음은 공급자의 IP 주소가 소비자의 **MyConfig.properties** 구성 문서의 IPADDRESS 키에 존재하거나, 공급자의 호스트 이름이 동일한 파일의 HOST 키에 있는 경우입니다.

```
<PropertiesConfigurationDocument name="MyConfig.properties"> (Type and Name of the
configuration document)
  <Condition> (Beginning of the search expression)
    <Operator type="or"> (Operator)
      <KeyCondition key="IPADDRESS"> (Start a condition and state its type)
        <Values>
          <Value>${IP_ADDRESS}</Value> (Use a variable stating the provider's IP address)
        </Values>
      </KeyCondition>
      <KeyCondition key="HOST"> (Start another condition, under the same operator)
        <Values>
          <Value>${HOSTNAME}</Value> (Use a variable stating the provider's host name)
        </Values>
      </KeyCondition>
    </Operator>
  </Condition>
</PropertiesConfigurationDocument>
```

논리 연산자를 중첩하여 조건을 만들 수 있으며(예: (C1 AND (C2 OR C3))) 이 경우 XML은 다음과 같습니다.

```
<PropertiesConfigurationDocument name="MyConfig.properties"> (Type and Name of the
configuration document)
  <Condition>
    <Operator type="and">
      C1
      <Operator type=”or”>
```

```
      C2
      C3
    </Operator>
  </Operator>
 </Condition>
</PropertiesConfigurationDocument>
```

여기서 C1, C2, C3 는 필수 검색 조건의 XML 조각입니다.

다음과 같이 세 가지 파일 유형이 지원되며 각각 고유한 유형의 검색식을 가집니다.

- PropertiesConfigurationDocument –각 줄에 key=value 형식이 있는 파일. 자세한 내용은 "속성 구성 문서"(83페이지)를 참조하십시오.

  KeyCondition –지정된 키 값에 대한 조건
- XmlConfigurationDocument –XML 파일. 자세한 내용은 "XML 구성 문서"(87페이지)를 참조하십시오.

  XPathCondition –XPath 쿼리 평가
- TextConfigurationDocument –텍스트 구성 문서. 자세한 내용은 "텍스트 구성 문서"(90페이지)를 참조하십시오.

  RegExp –정규식 평가

기본값에 대한 자세한 내용은 "변수의 기본값 사용"(76페이지)을 참조하십시오.

문제 해결

- 동일한 조건 트리 노드에 서로 다른 개념이 있을 수는 없습니다. 이와 마찬가지로 동일한 연산자에 개념이 다른 두 리프가 있을 수 없습니다.

  예를 들어, 다음 조건 정의는 올바르지 않습니다.

```
<Condition>
  <Operator type="and">
    <XPathCondition>
      <XPath>/Setup/Configuration/Hostname[matches(@Name, '.*?${Concept1.HOSTNAME}.*')]
</XPath>
    </XPathCondition>
    <XPathCondition>
      <XPath>/Setup/Configuration/Port[matches(text(), '${Concept2.PORT}')]</XPath>
    </XPathCondition>
  </Operator>
</Condition>
```

변수의 기본값 사용

특정 연결 문자열 값이 일부 기본값과 동일한 경우 구성 파일에 해당 값이 포함되지 않는 경우가 있습니다. 예를 들어, HTTP URL을 정의할 때 포트 값 80이 URL에 구체적으로 표시되지 않을 수 있습니다. 즉, http://www.hp.com:80/과 http://www.hp.com/은 둘 다 올바르지만 이 구성 파일에는 http://www.hp.com/이 포함될 가능성이 높습니다. 이로 인해 검색 조건을 작성할 때 문제가 발생할 수

있는데, PORT 변수에 값 80이 포함되어 있고 조건에 해당 값이 존재하도록 지정된 경우 해당 값이 지정되어 있지 않으면 URL 테스트가 실패하게 되기 때문입니다.

이를 해결하기 위해 원하는 경우 각 변수에 기본값을 사용할 수 있습니다. 변수 값이 기본값과 동일한 경우 검색식을 변경할 수 있습니다.

검색식을 변경하는 방법에는 다음 두 가지가 있습니다.

- 변수 값이 기본값과 동일한 경우 조건 무시

  검색 조건에서 하나 이상의 변수 값이 기본값과 동일한 경우 조건을 완전히 무시하려면 이 옵션을 선택합니다. 조건을 무시하면 True 또는 False를 반환하지 않고 무시하므로 논리 연산자의 결과가 연산자 및 기타 연산에 따라 달라집니다. 예를 들면 다음과 같습니다.

  - **A AND B** 형식의 조건(여기서 **A** 및 **B**는 다른 식)에서 **A**를 무시하면 결과는 **B**의 결과와 동일합니다.
  - 식 **A AND B AND C**의 경우 **A**를 무시하면 결과는 **B AND C**와 동일합니다.
  - 식 **A OR B OR C**의 경우 **A**를 무시하면 결과는 **B OR C**와 동일합니다.

  드문 경우지만 하위 조건을 모두 무시하면(전체 조건 무시) 구성 문서 조건 결과가 False가 됩니다.

  조건을 무시하려면 변수 목록 다음에 해당 조건을 무시하도록 트리거하는 **ignoreIfDefaultValue** 특성을 조건에 추가합니다. 예를 들면 다음과 같습니다.

  ```
  <KeyCondition key="serverName" ignoreIfDefaultValue="IPADDRESS">
  ```

- 변수 값이 해당 기본값과 동일한 경우 다른 조건 사용

  PORT 80 및 URL을 예로 들어 살펴보겠습니다. 다음 정규식을 사용하여 URL을 테스트합니다.

  ```
  http://${DOMAIN}:${PORT}/
  ```

  콜론(:)이 문자열에 존재하지 않으므로 이 정규식은 http://www.hp.com/과 같은 문자열에 대해 True로 평가되지 않습니다. 따라서 PORT 변수 값이 **80**인 경우 다음 정규식을 테스트해야 합니다.

  ```
  http://${DOMAIN}/
  ```

  이러한 종류의 시나리오에서는 대체 식을 사용하면 됩니다. 대체 식을 정의하려면 동일한 조건에 검색식을 여러 개 정의하고 변수가 기본값을 가질 때 사용할 검색식을 명시합니다. 예를 들면 다음과 같습니다.

  ```
  <KeyCondition key="url">
    <RegExp>http://${DOMAIN}:${PORT}/</RegExp>
    <RegExp alternativeFor=”PORT”>http://${DOMAIN}/</RegExp>
  </KeyCondition>
  ```

  alternativeFor 특성을 사용하여 변수가 기본값을 가지는 경우 원래의 식 대신 대체 식을 평가하도록 명시합니다. 원래 식은 **alternativeFor** 특성이 없거나 **alternativeFor** 특성이 비어 있는 식입니다.

식에 표시되며 기본값을 가지는 모든 변수에 대해 대체 식을 정의하여 해당 변수의 모든 조합에 적용되도록 해야 합니다. 그렇지 않으면 컴파일 오류가 발생합니다. 변수가 하나뿐인 경우에는 위의 예와 같이 대체 식이 하나만 있으면 됩니다. 기본값을 가지는 변수가 여러 개인 경우에는 대체 식도 여러 개 있어야 합니다.

예를 들어, DOMAIN 변수도 기본값을 가지는 경우 **alternativeFor** 문을 사용하면 다음과 같은 대체 조건을 모두 가지게 됩니다.

```
<KeyCondition key="url">
  <RegExp>http://${DOMAIN}:${PORT}/</RegExp>
  <RegExp alternativeFor="PORT">http://${DOMAIN}/</RegExp>
  <RegExp alternativeFor="DOMAIN">http://www.hp.com:${PORT}/</RegExp>
  <RegExp alternativeFor="PORT, DOMAIN">http://www.hp.com/</RegExp>
</KeyCondition>
```

런타임 동안 해당 조합 중 하나만 평가됩니다.

변수의 기본값 지정에 대한 자세한 내용은 "기본값"(72페이지)을 참조하십시오.

구성 문서의 경로 지정

구성 문서 경로는 호스트 파일 시스템의 파일 경로를 참조하지 않고 소비자 배포 가능 구성 요소 및 구성 문서 간 토폴로지 경로를 참조합니다.

경로가 지정되지 않은 경우 기본적으로 구성 문서가 Composition 링크를 통해 배포 가능 구성 요소에 연결되어 있는 것으로 간주됩니다. 즉, 배포 가능 구성 요소 CI가 구성 문서 CI를 소유한 것으로 간주됩니다.

다른 경로를 지정하려면 다음을 수행합니다.

1. "TQL 쿼리 정의"(95페이지)에 지정된 대로 TQL 쿼리를 정의합니다.
2. 다음과 같이 <DocumentCILocation> 요소를 사용하여 구성 문서의 TQL 쿼리를 참조합니다.

```
<TextConfigurationDocument name="cmdb.properties">
  <DocumentCILocation>
    <ReferenceLocation>YourQueryName</ReferenceLocation>
  <DocumentCILocation>
  <Condition>
    ...
  </Condition>
</TextConfigurationDocument>
```

이 예에서 YourQueryName은 <Queries> 섹션의 쿼리 이름에 대한 참조입니다.

TQL 쿼리를 작성하여 경로를 지정할 경우 다음을 수행합니다.

- TQL 쿼리에서 배포 가능 구성 요소와 구성 문서 간의 경로를 지정해야 합니다. 이 경로는 단순해야 합니다(순환 없음).
- TQL 쿼리에 다음과 같이 대/소문자를 구분하는 특정 이름이 있는 두 개의 끝 노드가 포함되어야 합니

다.

- Deployable - 경로의 배포 가능 구성 요소 CI
- Configuration_document –구성 문서를 지정하는 경로의 CI

- Deployable 및 Configuration_document 노드에는 조건을 사용할 수 없습니다.
- 모든 노드 간 카디널리티는 1..1이어야 합니다.
- 정규 링크 유형만 지원됩니다. 복합, 조인 또는 하위 그래프는 사용할 수 없습니다.

구성 문서 다시 정의

상황에 따라 구성 문서로 다른 구성 문서를 다시 정의하거나 다른 구성 문서로 해당 구성 문서를 다시 정의할 수 있습니다. 예를 들면, 호스트에 있는 구성 문서에서 해당 호스트가 속한 클러스터에 있는 구성 문서를 다시 정의할 수 있습니다. 그러한 경우 값을 다시 정의할 수 있는 모든 파일에서 키 값을 검토한 후 우선 순위가 가장 높은 값을 선택해야 합니다. 이 예제에서는 클러스터 문서보다 호스트의 로컬 구성 문서의 우선 순위가 더 높습니다.

종속 관계 서명에서 파일 다시 정의를 정의하려면 다음 구문을 사용하여 구성 문서에 우선 순위가 있는 여러 경로를 추가합니다.

```
<PropertiesConfigurationDocument name="resources.xml">
  <DocumentCILocation>
    <ReferenceLocation priority="2">websphereas_resource_configfiles</ReferenceLocation>
    <ReferenceLocation priority="3">j2ee_cluster_configfiles</ReferenceLocation>
    <ReferenceLocation priority="1">j2eeapplication_configfiles</ReferenceLocation>
  </DocumentCILocation>
  ...
</PropertiesConfigurationDocument
```

파일 이름(위 예의 경우 **resources.xml**)은 모든 참조 위치에서 동일해야 합니다.

참조 위치는 소비자 배포 가능 구성 요소에서 구성 문서로의 토폴로지 경로를 지정하는 TQL 쿼리에 대한 참조입니다. 따라서 해당 경로는 배포 가능 구성 요소와 다른 모든 위치 사이에 있어야 합니다. <DocumentCILocation>에 대한 자세한 내용은 "구성 문서의 경로 지정"(78페이지)을 참조하십시오.

우선 순위가 있는 여러 개의 경로를 추가할 때 조건 자체는 변경되지 않습니다. 런타임 시(검색식이 평가될 때) 우선 순위에 따라 올바른 값이 사용됩니다. 예를 들어, 속성 파일에서 지정된 키 K가 우선 순위 1 및 우선 순위 2에 모두 존재하는 경우(해당 값에 대한 조건 있음) 우선 순위가 1인 문서에 대해서만 조건이 평가됩니다. 키 K가 우선 순위 2에만 있는 경우에는 우선 순위가 2인 문서에 대해서만 조건이 평가됩니다.

속성 조건에 대한 자세한 내용은 "속성 구성 문서"(83페이지)를 참조하십시오.

XML 문서에서 각 XPath는 키로 간주됩니다. 예를 들어, \Root\Element\@Attribute는 해당 경로의 특성 "Attribute"가 키이며, 해당 값을 다른 파일에서 다시 정의할 수 있음을 의미합니다.

XPath에 일부 상수 조건도 포함되어 있는 경우(변수를 포함하지 않는 조건)에는 해당 조건이 키의 일부가 됩니다. 예를 들면 다음과 같습니다. \Root\Element[@name = 'name']\@Attribute

그러나 XPath에 상수 이외의 조건이 있는 경우 해당 조건이 키에서 제외되며 값의 일부로 간주됩니다. 따라서 경로 \Root\Element[@name = ${NAME}]\@Attribute의 경우 키는 경로 \Root\Element\@Attribute가 되고, 조건 Element[@name = 'name']은 해당 키가 있으며 우선 순위가 가장 높은 문서에서만 평가됩니다.

XML 문서에 우선 순위가 사용되는 경우 잘못되거나 예기치 않은 동작을 방지할 수 있도록 위에 설명된 대로 XPath 조건을 작성해야 합니다.

XPath 조건에 대한 자세한 내용은 "XML 구성 문서"(87페이지)를 참조하십시오.

참고: 속성 및 XML 구성 문서에는 우선 순위가 있는 참조 위치를 여러 개 사용할 수 있지만 텍스트 문서에서는 사용할 수 없습니다.

우선 순위가 같은 여러 구성 문서

다양한 <ReferenceLocation> 요소에 동일한 우선 순위를 부여할 수 있습니다. 이 경우 조건을 평가할 때 프레임워크에서 우선 순위가 같은 파일을 모두 확인하고 모두 일치시킵니다.

설정된 파일 수가 하나 이상 True로 평가되면 조건이 True로 평가됩니다. 파일 수는 **samePriorityMatchAtLeast** 특성으로 정의됩니다.

예:

```
<PropertiesConfigurationDocument name="resources.xml">
  <DocumentCILocation samePriorityMatchAtLeast=”1”>
    <ReferenceLocation priority="1">websphereas_resource_configfiles</ReferenceLocation>
    <ReferenceLocation priority="1">j2ee_cluster_configfiles</ReferenceLocation>
    <ReferenceLocation priority="1">j2eeapplication_configfiles</ReferenceLocation>
  </DocumentCILocation>
...
</PropertiesConfigurationDocument>
```

이 예는 조건을 평가할 때 참조된 파일 위치 중 하나 이상이 해당 조건이 True로 평가되는 파일에 대한 것임을 나타냅니다.

참고: 현재 **samePriorityMatchAtLeast** 특성에는 값 **1**만 지원됩니다.

여러 문서에 정의된 종속 관계

대부분의 경우 소비자가 서비스를 제공하는지 여부를 결정하려면 여러 구성 문서를 검색하여 모든 공급자의 연결 문자열을 찾아야 합니다.

관련이 없는 여러 구성 문서 검색

구성 문서는 서로 무관할 수 있으며 일부 연결 문자열이 어떤 문서에는 있고 다른 문서에는 없을 수 있습니다. 예를 들어 소비자에게 **A.conf** 및 **B.conf**의 두 구성 문서가 있을 수 있습니다. 공급자의 연결 문자열은 **A.conf**에 정의된 C1 및 **B.conf**에 정의된 C2입니다. C1 및 C2 모두에서 소비자와 공급자 사이에 종속 관계가 실제로 존재하는지 확인해야 합니다. 이 경우 각 문서에 하나씩, 모두 두 개의 필수 검색식이 있습니

다. 따라서 <Dependency> 태그의 구조는 다음과 같습니다.

```
<Dependency name="dependency_name" providerCiType="webmodule" scope="my_scope">
  <PropertiesConfigurationDocument name="A.conf">
    <Condition>
      <Operator type="and">
        <KeyCondition key="C1">
          <Values>
            <Value>${VAR_1}</Value>
          </Values>
        </KeyCondition>
      </Operator>
    </Condition>
  </PropertiesConfigurationDocument>
  <TextConfigurationDocument name="B.conf">
    <Condition>
      <Operator type="and">
        <RegExpCondition>
          <RegExp>C2?${VAR_2}</RegExp>
        </RegExpCondition>
      </Operator>
    </Condition>
  </TextConfigurationDocument>
</Dependency>
```

참고:

- <Dependency> 요소에 표시될 수 있는 파일 수에는 제한이 없습니다.
- 각 파일의 유형(속성, XML 또는 텍스트)은 다를 수 있습니다.
- <Dependency> 요소의 파일 순서는 무시됩니다. 파일은 임의의 순서로 테스트됩니다.
- 종속성이 존재하려면 모든 파일의 검색식에서 True를 반환해야 합니다.
- 연결 문자열이 여러 파일에 분산되어 있으므로 해당 범위에서 필수 파일 중 하나 이상이 반환될 수 있도록 범위 검색식이 충분히 광범위해야 합니다. 자세한 내용은 "검색 범위 지정"(92페이지)을 참조하십시오.

여러 파일을 검색할 때의 흐름을 요약하면 다음과 같습니다.

1. "검색 범위 지정"(92페이지)에 설명된 대로 범위 필터링이 실행됩니다. 반환되는 필수 파일이 없으면 소비자와 공급자 사이에 종속 관계가 존재하지 않습니다.
2. 필터에서 파일이 하나 이상 반환되면 해당 파일에 연결된 소비자 및 나머지 필수 구성이 Data Flow Probe로 다운로드됩니다.
3. 각 파일의 조건이 임의의 순서로 평가됩니다.
4. 모든 조건에서 True를 반환하면 종속성이 존재하고, 그러지 않으면 존재하지 않습니다.

종속 관계가 있는 여러 구성 문서 검색

다른 경우에는, 소비자의 구성 문서는 서로 관련되어 있습니다. 예를 들어, **DBConnections.conf** 파일은 여러 데이터베이스 및 스키마에 대해 여러 연결 문자열을 정의하고 이러한 각 연결 문자열에는 이름이 지정됩니다. **MyApp.conf** 파일에서 이러한 연결 이름 중 하나가 사용되면 MyApp에서 해당 특정 데이터베이스 및 스키마를 사용하도록 정의됩니다.

로컬 범위 변수는 종속 관계가 있는 구성 문서에 사용됩니다. 변수 값(예: 연결 문자열의 이름)이 해당 문서에 존재하도록 삽입문을 사용하여 로컬 범위 변수에 대한 값을 구성 문서 중 하나에 삽입합니다. 그러면 해당 변수를 다른 구성 문서의 조건에 사용할 수 있습니다(예: **MyApp.conf** 파일에 연결 이름을 사용하여 MyApp에서 해당 공급자(DB)를 사용하도록 함). 자세한 내용은 "로컬 범위"(69페이지)를 참조하십시오.

변수 삽입문을 사용하여 변수에 값을 삽입합니다. 문서 유형에 따라 문이 다릅니다.

- 속성 파일 - KeyVariable 및 KeyRegExpVariable. 자세한 내용은 "속성 구성 문서"(83페이지)를 참조하십시오.
- XML 파일 - XPathVariable. 자세한 내용은 "XML 구성 문서"(87페이지)를 참조하십시오.
- 텍스트 파일 - RegExpVariable. 자세한 내용은 "텍스트 구성 문서"(90페이지)를 참조하십시오.

이 경우 각 파일에 하나씩 두 개의 필수 검색식이 있습니다. 따라서 <Dependency> 태그의 구조는 다음과 같습니다.

```
<Dependency name="dependency_name" providerCiType="webmodule" scope="my_scope">
    <VariableDeclarations>
      <Variable name=”REFERENCE_NAME” />
    </VariableDeclarations>
  <PropertiesConfigurationDocument name="DBConnections.conf">
    <Condition>
      <Operator type="and">
        <KeyCondition key="C1">
          <Values>
            <Value>${VAR_1}</Value>
          </Values>
        </KeyCondition>
      </Operator>
    </Condition>
    <Variables>
      <KeyVariable variable=”REFERENCE_NAME” key=”reference_name” />
    </Variables>
  </PropertiesConfigurationDocument>
  <TextConfigurationDocument name="MyApp.conf">
    <Condition>
      <Operator type="and">
        <RegExpCondition>
          <RegExp>C2?${REFERENCE_NAME}</RegExp>
        </RegExpCondition>
```

```
    </Operator>
   </Condition>
  </TextConfigurationDocument>
</Dependency>
```

참고:

- 종속 관계에 사용될 하나 이상의 로컬 변수를 정의하려면 <VariableDeclarations> 요소를 사용합니다.
- <Variables> 요소는 값을 로컬 변수에 삽입하는 데 사용됩니다. 섹션은 <Condition> 요소에서 True를 반환하는 경우에만 실행됩니다.
- 이 예에서 <KeyVariable>은 "reference_name" 키 값을 REFERENCE_NAME 변수에 삽입하는 데 사용됩니다.
- ${REFERENCE_NAME} 변수는 **MyApp.conf** 조건에서 사용됩니다. 이 변수는 **DBConnections.conf** 파일에 따라 다르며 **DBConnections.conf** 이후 및 검색식이 True로 평가되는 경우에만 평가됩니다.
- 파일 간 순환 종속 관계는 허용되지 않습니다.
- **MyApp.conf** 문서를 평가할 때 ${REFERENCE_NAME} 변수에는 **DBConnections.conf**의 값이 이미 포함되어 있습니다.
- 연결 문자열이 여러 파일에 분산되어 있으므로 해당 범위에서 종속 관계가 없는 파일을 반환할 수 있도록 범위 검색식이 광범위해야 합니다. 자세한 내용은 "검색 범위 지정"(92페이지)을 참조하십시오.

서로 종속적인 여러 파일을 검색할 때의 흐름을 요약하면 다음과 같습니다.

1. "검색 범위 지정"(92페이지)에 설명된 대로 범위 필터링이 실행됩니다. 반환되는 필수 파일이 없거나 다른 파일에 종속적인 파일이 없으면 소비자와 공급자 사이에 종속 관계가 존재하지 않습니다.
2. 필터에서 종속 관계가 없는 파일을 하나 이상 반환합니다.
3. 해당 파일에 연결된 소비자 및 나머지 필수 구성이 Data Flow Probe로 다운로드됩니다.

예를 들어, 참조하는 키가 존재하지 않는 경우 삽입문에서 값을 반환하지 않을 수 있습니다. 이 경우 변수는 빈 값을 가져옵니다. 그러나 이는 원하는 동작이 아닐 수 있습니다. 필요한 값이 없는 경우 이는 종속 관계가 없으며 계속해서 후속 파일을 평가할 필요가 없음을 나타낼 수 있습니다. 이러한 시나리오의 경우 삽입문에 allowNull="false"를 사용합니다. 예를 들면 다음과 같습니다.

```
<KeyVariable variable="REFERENCE_NAME" key="reference_name" allowNull="false" />
```

속성 구성 문서

속성 구성 문서는 구성을 Key=value 형식으로 저장합니다. 예를 들어, 속성 구성 문서는 다음과 같습니다.

```
# My configuration document
Hostname=MyNode
```

여기서 Hostname은 키이며 MyNode는 연관된 값입니다.

"#"는 주석 줄을 나타냅니다. 주석 줄은 검색식을 테스트할 때 무시됩니다.

구성 문서가 속성 구성 문서임을 명시하려면 <PropertiesConfigurationDocument> 요소를 사용합니다. 예를 들면 다음과 같습니다.

```
<Dependency name="history_db" providerCiType="cmdb" scope="default">
  <PropertiesConfigurationDocument name="cmdb.conf">
    ...
```

조건 정의

속성 파일에서 연결 문자열 변수의 존재 여부를 테스트하는 방법에는 여러 가지가 있습니다. 모든 조건 유형에서 **key** 특성에 지정된 키 값을 일부 값에 대해 테스트합니다. 키는 항상 상수 값입니다(변수를 사용할 수 없음).

- 상수 값이 키 값으로 존재하는지 테스트합니다.

  ```
  <KeyCondition key="dal.datamodel.name">
    <Values>
      <Value>ConstantValue1</Value>
      <Value>ConstantValue2</Value>
    </Values>
  </KeyCondition>
  ```

  위 조건은 키 값이 상수 값 중 하나와 동일한 경우에만(대/소문자 구분하지 않음) True를 반환합니다.

- 변수 값이 키 값으로 존재하는지 테스트합니다.

  ```
  <KeyCondition key="dal.datamodel.name">
    <Values>
      <Value>${VARIABLE1}</Value>
      <Value>${VARIABLE2}</Value>
    </Values>
  </KeyCondition>
  ```

  위 조건은 키 값이 변수 값 중 하나와 동일한 경우에만(대/소문자 구분하지 않음) True를 반환합니다. 변수 값이 값 목록인 경우 모든 값을 테스트하고 키와 목록에 있는 값 중 하나만 일치해도 True를 반환합니다.

  참고: 변수와 상수 값을 모두 동일한 조건으로 지정할 수 있습니다.

- 키 값이 일부 정규식을 준수하는지 테스트합니다.

```
<KeyCondition key="dal.datamodel.name">
  <RegExp>
    ^SomeText${VARIABLE}MoreText$
  </RegExp>
</KeyCondition>
```

정규식에는 조건의 일부로 하나 이상의 변수가 포함될 수도 있습니다. 런타임 시에는 구체 검색식을 생성할 때 변수가 실제 값으로 대체됩니다.

변수에 값 목록이 포함되어 있는 경우 다음과 비슷한 식이 생성됩니다.

```
^SomeText(Value1 | Value2  | Value3)MoreText$
```

이 경우 각 값에서 조건에 True를 반환할 수 있습니다.

속성 구성 문서에 있는 조건의 예

```
<PropertiesConfigurationDocument name="MyConfig.properties">
  <Condition>
    <Operator type="and">
      <KeyCondition key="TYPE">
        <Values>
          <Value>HTTP</Value>
          <Value>HTTPS</Value>
        </Values>
      </KeyCondition>
      <KeyCondition key="IPADDRESS">
        <Values>
          <Value>${IP_ADDRESS}</Value>
          <Value>${HOSTNAME}</Value>
        </Values>
      </KeyCondition>
      <KeyCondition key="SITE">
        <RegExp>
          //${SITE_NAME}//.*
        </RegExo>
      </KeyCondition>
    </Operator>
  </Condition>
</PropertiesConfigurationDocument>
```

즉, **MyConfig.properties**라는 속성 구성 문서에 다음 조건이 모두 있어야 합니다.

- TYPE이라는 키가 HTTP 또는 HTTPS 값 중 하나를 포함합니다.
- IPADDRESS라는 키가 공급자의 IP 주소(또는 IP 주소 중 하나) 또는 호스트 이름을 포함합니다.
- SITE라는 키가 “/”로 시작하고, 그 뒤에 공급자의 SITE_NAME 및 또 다른 “/”가 있으며, 그 뒤에 임의의 문자열이 있습니다.

변수 값 삽입

다음은 키 값 또는 값의 일부를 변수로 추출하는 두 가지 방법입니다.

- 키 값을 삽입하는 방법
  - 전용 <Variables> 섹션에서 다음 구문을 사용합니다.

```
<KeyVariable variable="VARIABLE_NAME" key="KEY_NAME" />
```

  - 검색 조건의 일부로 다음 구문을 사용합니다.

```
<KeyCondition key="KEY_NAME">
  <Values>
    <Value>REQUIRED_VALUE</Value>
  </Values>
  <Variables>
    <KeyVariable variable="VARIABLE_NAME" />
  </Variables>
</KeyCondition>
```

  VARIABLE_NAME 변수의 키가 <KeyCondition> 요소에 정의됩니다.

  이 구문은 <Values> 대신 <RegExp>를 사용해도 동일합니다. 두 경우 모두 변수에 전체 키 값이 삽입됩니다.

- 키 값의 일부를 삽입하는 방법
  - 전용 <Variables> 섹션에서 다음 구문을 사용합니다.

```
<KeyRegExpVariable variable="VARIABLE_NAME" key="KEY_NAME" expression="REGULAR_
EXPRESSION" group="GROUP_NUMBER" />
```

  이 경우 GROUP_NUMBER는 REGULAR_EXPRESSION에서 정의하는 정규식의 그룹 인덱스(0으로 시작)입니다. 예를 들어 다음 라인 및

```
myKey = 123Value123
```

  다음 명령문이 포함된 속성 문서에서

```
<KeyRegExpVariable variable="VAR" key="myKey" expression="[0-9]*([a-zA-Z]*)[0-9]*"
group="0" />
```

  변수 "VAR"에 값 "Value"를 삽입합니다.

  같은 파일 또는 다른 파일에 정의된 변수를 사용할 수 있습니다. 예를 들면 다음과 같습니다.

```
<KeyRegExpVariable variable="VAR" key="myKey" expression="${PREV_VAR}([a-zA-Z]*)${PREV_
```

```
VAR}" group="0" />
```

PREV_VAR에 값 "123"이 있다고 가정하면 키의 전체 값을 삽입하는 것과 같은 방법으로 VAR은 값 "Value"를 가집니다. 두 옵션 모두 변수에 항상 단일 값이 포함됩니다.

- 검색 조건의 일부로 다음 구문을 사용합니다.

```
<KeyCondition key="KEY_NAME">
  <RegExp>REGULAR_EXPRESSION</RegExp>
  <Variables>
    <KeyRegExpVariable variable="VARIABLE_NAME" group="GROUP_NUMBER" />
  </Variables>
</KeyCondition>
```

VARIABLE_NAME 변수의 키가 <KeyCondition> 요소에 정의됩니다.

이 구문은 <RegExp>를 사용할 때만 지원되며, 변수의 "그룹" 특성이 참조하는 정규식은 <RegExp>에 있는 정규식과 동일합니다.

XML 구성 문서

XML 구성 문서를 사용하면 XPath 쿼리를 사용하여 XML 표준에 따라 구성 문서에 대한 검색식을 쉽게 작성할 수 있습니다.

구성 문서가 XML 구성 문서임을 명시하려면 <XmlConfigurationDocument> 요소를 사용합니다. 예를 들면 다음과 같습니다.

```
<Dependency name="some_reference" providerCiType="webmodule" scope="default">
  <XmlConfigurationDocument name="web.xml">
    ...
```

조건 정의

XML 구성 문서에 허용되는 검색 조건은 유효한 XPath 2.0 쿼리뿐입니다. 노드를 XPath에서 선택한 경우 조건에서 True를 반환하고 그렇지 않은 경우 False를 반환합니다. 쿼리에는 다음 위치에 있는 변수가 포함될 수 있습니다.

- 요소 이름

```
/Root/${SITE_NAME}
```

변수에 값이 여러 개 있으면 프레임워크에서 XPath 문을 여러 개 실행합니다. 예를 들어, SITE_NAME에 값 SITE1 및 SITE2가 있으면 이 예의 문에서 다음과 같은 두 개의 구체 검색을 생성합니다.

```
\Root\SITE1
```

```
\Root\SITE2
```

- 등식 테스트

```
/Element[@att = ${VAR}]
또는
/Element/text() = ${VAR}
```

위 예에 표시된 등식은 단일 값을 포함하는 변수에 대해서만 작동합니다. 변수에 값이 두 개 이상 포함되어 있는 경우에는 대신 다음 구문을 사용합니다.

```
/Element[${equals(@att, VAR)]
또는
/Element[${equals-ignore-case(text(), VAR}]
```

참고:

- 대/소문자를 구분하는 등식은 ${equals()}를, 대/소문자를 구분하지 않는 등식에는 ${equals-ignore-case()}를 사용하여 테스트합니다.
- 첫 번째 매개 변수는 XPath 함수 또는 특성 이름이어야 합니다.
- 두 번째 매개 변수는 항상 변수 이름이어야 합니다. 변수 이름은 ${} 없이 표시되어야 합니다.
- 이 구문은 단일 값 또는 여러 값이 있는 변수에 대해 작동하며, 모든 경우에 사용할 수 있습니다.
- 여러 개의 값이 있고 첫 번째 매개 변수의 값이 해당 변수의 값 하나와만 동일한 경우 함수에서 True를 반환합니다.

- 정규식

```
/datasources/mbean[matches(@name, '.*?,ip=${IPADDRESS}.*')]
```

변수에 값 목록이 있는 경우 다음과 비슷한 식이 생성됩니다.

```
/datasources/mbean[matches(@name, '.*?,ip=(Value1 | Value2 | Value3).*')]
```

각 값에서 조건에 대해 True를 반환합니다.

XML 구성 문서에 있는 조건의 예

```
<XmlConfigurationDocument name="MyConfig.xml">
  <Condition>
```

```
    <Operator type="and">
      <XPathCondition>
        <XPath>\URL\Protocol[@name='HTTP' or @name='HTTPS']</XPath>
      </XPathCondition>
      <XPathCondition>
        <XPath>\URL\Host[${equals-ignore-case(@name, HOSTNAME)} or ${equals(@name, IP_
ADDRESS)}]</XPath>
      </XPathCondition >
      <XPathCondition>
        <XPath>\URL\Site[matches(text(), '//${SITE_NAME}//.*')]</XPath>
      </XPathCondition>
    </Operator>
  </Condition>
</XmlConfigurationDocument>
```

이 예는 **MyConfig.xml** XML 구성 문서에 다음 조건이 모두 있어야 함을 보여 줍니다.

- \URL\Protocol\@name 값이 HTTP 또는 HTTPS여야 합니다.
- \URL\Host\@name에 공급자의 IP 주소(또는 IP 주소 중 하나) 또는 호스트 이름이 포함되어야 합니다.
- \URL\Site\text()는 /로 시작하고 그 뒤에 공급자의 SITE_NAME 및 다른 /가 있으며, 그 뒤에 임의의 문자열이 있습니다.

변수 값 삽입

XPath를 사용하여 텍스트 값을 쿼리하고 변수에 삽입할 수 있습니다. 하나의 변수에 전체 요소를 삽입하는 것은 불가능합니다. 전용 <Variables> 섹션에서 이 작업을 수행하려면 다음 구문을 사용합니다.

```
<XPathVariable variable="VARIABLE_NAME" xpath="XPATH_QUERY" />
```

다음은 몇 가지 예입니다.

- 특성 값 선택

  ```
  <XPathVariable variable="VAR" xpath="\Root\Element\@Att" />
  ```

  XML 문서에서 모든 \Root\Element 요소의 **Att** 특성에서 값을 선택합니다.
- 요소 텍스트 선택

  ```
  <XPathVariable variable="VAR" xpath="\Root\Element\text()" />
  ```

  \Root\Element와 일치하는 모든 요소의 텍스트를 선택합니다.
- 조건을 사용하여 특성 값 선택(같거나 다른 구성 문서에 있는 변수 사용)

```
<XPathVariable variable=”VAR” xpath=”\Root\Element[${equals(@Att, VAR)}]\@AnotherAtt” />
```

@Att=${VAR}인 모든 \Root\Element에서 @AnotherAtt 값을 선택합니다.

XPath 변수를 XPath 조건의 일부로 사용할 수 있습니다. 이 모드에서는 결과가 있었다고(즉, 조건이 True로 평가되었음) 가정하고 XPath에 조건에 대한 결과 노드의 상대 경로도 포함될 수 있습니다. 다음 구문을 사용합니다.

```
<XPathCondition>
  <XPath>XPATH_QUERY</XPath>
  <Variables>
    <XPathVariable variable="VARIABLE_NAME" relativePath="RELATIVE_XPATH_QUERY" />
  </Variables>
</XPathCondition>
```

예:

```
<XPathCondition>
  <XPath>/${VAR_1}/chcpCodeToCharsetName[@name='test1' and matches(text(), '.*?${OUTPUT_
VAR2}.*')]</XPath>
  <Variables>
    <XPathVariable variable="OUTPUT_VAR1" relativePath="./@type" />
  </Variables>
</XPathCondition>
```

XPathCondition이 True로 평가되었고 <XPath>로 일부 XML 노드(특히 이 경우에는 요소)를 선택했다고 가정하면 변수는 해당 노드의 "유형" 특성 값을 포함하게 됩니다.

"/"로 시작하는 문서의 루트에서 XPath를 지정하면 해당 변수와 무관한 변수에 대해 XPath를 정의할 수 있습니다.

XPath를 삽입한 후에는 변수에 값 목록이 포함될 수 있습니다.

텍스트 구성 문서

텍스트 구성 문서는 컨텐츠 개발자에게 알려진 형식을 가지는 텍스트 문서로, 속성 구성 문서나 XML 구성 문서가 아닙니다. 정규식을 사용하여 텍스트 구성 문서에 검색식을 작성할 수 있습니다.

구성 문서가 텍스트 파일 유형임을 명시하려면 <TextConfigurationDocument> 요소를 사용합니다. 예를 들면 다음과 같습니다.

```
<Dependency name="some_reference" providerCiType="oracle" scope="default">
  <TextConfigurationDocument name="tnsnames.ora">
    ...
```

조건 정의

텍스트 구성 문서에 허용되는 검색 조건만 정규식입니다. 패턴이 일치하는 경우 조건에서 True를 반환합니다.

정규식에는 조건의 일부로 하나 이상의 변수가 포함될 수 있습니다. 런타임 시 구체 검색식을 생성할 때 변수는 실제 값으로 대체됩니다.

변수에 값 목록이 있는 경우 다음과 비슷한 식이 생성됩니다.

```
^SomeText(Value1 | Value2 | Value3)MoreText$
```

각 값에서 조건에 대해 True를 반환합니다.

텍스트 구성 문서에 있는 조건의 예

```
<TextConfigurationDocument name="MyConfig.txt">
  <Condition>
    <Operator type="or">
      <RegExpCondition>
        <RegExp>^HTTPS?://(${HOSTNAME)/${SITE_NAME}/.*$</RegExp>
      </RegExpCondition>
      <RegExpCondition>
        <RegExp>^HTTPS?://(${IP_ADDRESS)/${SITE_NAME}/.*$</RegExp>
      </RegExpCondition>
    </Operator>
  </Condition>
</TextConfigurationDocument>
```

이 예는 MyConfig.txt 텍스트 구성 문서에 다음 조건이 모두 있어야 함을 보여 줍니다.

- **\URL\Protocol\@name 값이 HTTP 또는 HTTPS여야 합니다.**
- **\URL\Host\@name에 공급자의 IP 주소(또는 IP 주소 중 하나) 또는 호스트 이름이 포함되어야 합니다.**
- **\URL\Site\text()는 /로 시작하고 그 뒤에 공급자의 SITE_NAME 및 다른 /가 있으며, 그 뒤에 임의의 문자열이 있습니다.**

변수 값 삽입

정규식을 그룹과 함께 사용하여 텍스트 문서의 변수에 값을 삽입할 수 있습니다. 그룹은 삽입해야 할 텍스트를 표시합니다. 전용 <Variables> 섹션에서 다음 구문을 사용합니다.

```
<RegExpVariable variable=”VARIABLE_NAME” expression=”REGULAR_EXPRESSION”
group=”GROUP_NUMBER” />
```

이 경우 GROUP_NUMBER는 REGULAR_EXPRESSION에서 정의하는 정규식의 그룹 인덱스(0으로 시작)입니다.

예를 들면, 다음 라인 및

```
This is part of a configuration document
```

다음 명령문이 포함된 파일의 경우

```
<RegExpVariable variable=”VAR” expression=”(.*)configuration (.*)” group=”1” />
```

변수 VAR에 값 "document"를 넣습니다. 서로 다른 문서에서 정의된 변수를 같은 문서에서 함께 사용할 수도 있습니다. 예를 들면 다음과 같습니다.

```
<RegExpVariable variable=”VAR” expression=”.*${PREV_VAR} (.*)” group=”0” />
```

PREV_VAR의 값이 "configuration"인 경우 VAR에 문자열 "document"가 삽입됩니다.

다음 구문을 사용하여 <RegExpCondition> 태그의 일부로 <RegExpVariable>을 사용할 수 있습니다.

```
<RegExpCondition>
  <RegExp>REGULAR_EXPRESSION</RegExp>
  <Variables>
    <RegExpVariable variable="VARIABLE_NAME" group="GROUP_NUMBER" />
  </Variables>
</RegExpCondition>
```

변수의 "그룹" 특성이 참조하는 정규식은 <RegExp>에 있는 것과 동일합니다.

삽입되는 변수는 항상 단일 값으로 삽입됩니다.

검색 범위 지정

검색 범위의 목적은 잠재적으로 특정 공급자 유형과의 종속 관계가 있을 수 있는 모든 소비자를 찾는 것입니다. 범위는 다음 두 가지 용도로 사용됩니다.

- 검색 결과를 줄일 수 있도록 무관한 배포 가능 구성 요소를 UCMDB에서 필터링 (필수)

  이를 위해 모든 구성 문서에서 공급자의 연결 문자열 하위 집합을 찾고, 해당 구성 문서와 연결된 배포 가능 구성 요소로만 검색을 제한합니다.

  이 검색식은 모든 잠재 소비자를 최대한 빨리 반환하지만 소비자-공급자 종속 관계가 존재하는지 여부에 대한 정확한 결과는 반환하지 않으므로 공급자별 종속 관계 검색과 다릅니다.

  검색식 구문은 구성 문서 검색식을 작성하는 방법과 유사하게 구성됩니다. 그러나 파일 유형별 특수 조건이 없으며 파일 이름을 지정하지 않아도 됩니다. 자세한 내용은 "검색식 구성"(74페이지)을 참조하십시오.

  이 예는 식 (x | ((y & z & w) & (h | g)))를 나타냅니다.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ConfigurationDocumentSearchCondition xmlns="http://www.hp.com/ucmdb/1-0-
0/DependenciesDefaultSearch">
  <Operator type="Or">
    <Operator type="And">
      <Operand value="y"/>
      <Operand value="z"/>
      <Operand value="w"/>
      <Operator type="Or">
        <Operand value="h"/>
        <Operand value="g"/>
      </Operator>
    </Operator>
    <Operand value="x"/>
  </Operator>
</ConfigurationDocumentSearchCondition>
```

x, y, z, w, h, g는 상수 값, 변수 또는 개념 변수가 될 수 있습니다.

값 또는 변수 값은 UCMDB에 있는 모든 관련 소비자 구성 문서에서 이러한 식을 사용하여 검색합니다.

해당 파일이 있는 소비자만 공급자별 검색이 진행되며 정확한 검색식을 사용하여 평가됩니다.

- 배포 가능 구성 요소를 공급자의 연결 구성 요소 하위 집합으로 제한(예: 공급자 J2EE 도메인의 일부인 종속 관계 검색으로만 제한) (선택 사항)

  이를 위해 TQL 쿼리를 사용하여 공급자 유형에 특정 방식으로 연결된 모든 배포 가능 구성 요소를 찾습니다. 자세한 내용은 "TQL 쿼리 정의"(95페이지)를 참조하십시오.

  특정 공급자 CI가 지정된 TQL 쿼리 결과는 해당 공급자에 대한 종속 관계가 있을 수 있는 모든 가능한 배포 가능 구성 요소를 정의합니다. TQL 쿼리는 다음 규칙을 따라야 합니다.

  - 배포 가능 구성 요소에 대한 쿼리 노드를 "Deployable"이라고 해야 합니다(대/소문자 구분). 이 쿼리 노드의 결과는 해당 범위에 대해 가능한 배포 가능 구성 요소입니다.
  - "배포 가능" 구성 요소는 동일한 범위에 있을 수 있는 소비자 구성 요소와 공급자 구성 요소를 나타냅니다. 따라서 쿼리 노드 CI 유형은 소비자와 공급자의 배포 가능 CI 유형에 대한 상위 항목이어야 합니다.
  - TQL 쿼리에는 "Deployable" 이외의 기타 쿼리 노드가 하나만 있어야 합니다. 기타 쿼리 노드의 이름은 제한되지 않습니다. 이 노드를 Scope 쿼리 노드라고 합니다.
  - Deployable 및 Scope 쿼리 노드는 두 노드 사이에서 사용할 수 있는 모든 경로가 포함된 복합 링크로 연결되어 있습니다.

- 모든 범위는 공급자와 동일한 라우팅 도메인에 없는 소비자 배포 가능 구성 요소를 암시적으로 필터링합니다(공급자를 특정 라우팅 도메인에서 발견했다고 가정). 라우팅 도메인으로 필터링하려면 배포 가능 설명자에 노드 경로가 있어야 합니다. 자세한 내용은 "소비자의 설명자 정의"(73페이지)를 참조하십시오.

예를 들어, 공급자 배포 가능 구성 요소가 실행 중인 소프트웨어의 특정 유형인 경우 실행 중인 이 소프트웨어의 라우팅 도메인은 포함된 노드와 동일합니다. 노드의 라우팅 도메인은 IP 주소의 라우팅 도메인에 따라 결정됩니다. IP 주소의 라우팅 도메인이 서로 다른 경우 노드가 둘 이상의 라우팅 도메인과 관련될 수 있습니다. 한편, 공급자 배포 가능 구성 요소가 노드 또는 IP 주소와 관련이 없는 CI인 경우(예: 비즈니스 서비스) 해당 라우팅 도메인과 관계없이 소비자 배포 가능 구성 요소에 연결될 수 있습니다.

예를 들어, J2EE 도메인 범위를 정의하려면 다음을 수행합니다.

1. "J2EE 배포 개체" 유형의 배포 가능 쿼리 노드를 만듭니다. 이는 J2EE 도메인에 있는 배포 가능 구성 요소를 나타내는 모든 CI 유형 중 가장 낮은 공통 분모 클래스입니다. 예를 들어, 이 범위의 배포 가능 구성 요소는 웹 모듈 또는 J2EE 응용 프로그램이 될 수 있습니다.
2. "J2EE 도메인" 유형의 범위 쿼리 노드를 만듭니다.
3. J2EE 배포 개체 및 J2EE 도메인 간 모든 경로를 사용하여 종속 관계 및 범위 쿼리 노드 간 복합 링크를 만듭니다. 예를 들어, 복합 링크에는 다음이 포함될 수 있습니다.
   - J2EE 응용 프로그램 -Composition -J2EE 도메인
   - 웹 모듈 - Composition - J2EE 응용 프로그램

TQL 쿼리 정의는 범위 XML에 포함되어 있어야 합니다.

범위 검색의 변수 기본값

다음 예를 사용합니다.

- 기본값이 80인 PORT 변수를 사용하고 있습니다. 이 기본값은 일부 구성 파일에서 사실상 표시되지 않을 수 있습니다.
- 범위 검색식으로 IP_ADDRESS AND PORT를 사용 중이며 특정 공급자의 PORT 변수 값이 80입니다. 이 범위 검색에서는 기본 포트 값 80이 포함되지 않은 구성 문서에 연결된 배포 가능 구성 요소를 반환해야 하지만, 검색식에 포트가 구성 파일에 있어야 한다고 명시하므로 해당 구성 요소를 반환하지 않습니다.

따라서 PORT 변수에 기본값이 있으면 종속 관계 검색식에서 무시되는 것과 마찬가지로 범위 검색식에서 제거됩니다. 자세한 내용은 "변수의 기본값 사용"(76페이지)을 참조하십시오.

변수에 기본값 이외의 값이 포함되어 있는 경우 이 값은 일반적인 방법으로 검색식에 삽입됩니다.

문제 해결

범위에서 예상 소비자를 반환하는지 여부를 테스트하려면 UCMDB 서버에서 디스커버리 관리자 서비스의 **searchConfigurationDocuments** JMX 메서드를 사용합니다. "searchString" 매개 변수는 "검색 범위 지정"(92페이지)에 설명된 대로 검색식 XML이어야 합니다. XML에는 변수를 제외한 상수 값만 포함되어야 합니다.

이 XML은 검색 어댑터 중 하나에 있는 공급자의 통신 로그에서도 검색할 수 있습니다. 자세한 내용은 "종속 관계 검색 어댑터"(97페이지)를 참조하십시오.

TQL 쿼리 정의

1. 다음과 같이 <Queries> 섹션의 종속 관계 서명에 TQL 쿼리 정의를 추가합니다.

```
...
<Queries>
  <Query name=”YourQueryName”>
    [TQL XML]
  </Query>
</Queries>
```

2. a. 텍스트 편집기를 사용하여 빈 문서를 만듭니다.
   b. 다음 컨텐츠를 추가합니다.

```
<tql:query xmlns:ns4="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns3=
"http://www.hp.com/ucmdb/1-0-0/PolicyRuleDefinition" xmlns:tql=
"http://www.hp.com/ucmdb/1-0-0/TopologyQueryLanguage" name="<Query Name>">

</tql:query>
```

   c. 모델링 스튜디오에서 TQL 쿼리를 작성합니다. 자세한 내용은 *HP Universal CMDB 모델링 안내서*에서 "모델링 스튜디오"를 참조하십시오.
   d. TQL 쿼리를 XML로 내보냅니다.
   e. 내보낸 XML 문서를 열고 <resource> 요소에 있는 모든 텍스트를 복사합니다.
   f. 이 텍스트를 1단계에서 만든 문서의 <tql:query> 요소에 붙여 넣습니다.
   g. 텍스트 문서의 전체 컨텐츠를 복사하여 종속 관계 서명 파일의 <Query> 요소에 삽입합니다.

참고: TQL 쿼리는 종속 관계 서명 XML 파일에 포함되며, 유효성이 검사되지 않습니다. 그러나 쿼리 XML 파일 자체가 잘못된 경우 배포 중 예외가 발생합니다. 자세한 내용은 "컴파일 오류"(96페이지)를 참조하십시오.

여러 종속 관계 서명 파일 패키지화 및 배포

UCMDB에서는 여러 개의 종속 관계 서명 파일을 배포할 수 있습니다. 소비자-공급자 링크를 검색하는 경우 배포된 파일이 모두 사용됩니다.

종속 관계 서명 파일은 접두사가 **dependencies/**이고 접미사가 **.xml**인 디스커버리 구성 파일입니다(예: **dependencies/JEE.xml**).

각 종속 관계 서명 파일은 완전히 독립적입니다. 다음 항목을 참고하십시오.

- 글로벌 변수 정의는 정의된 파일에서만 사용할 수 있습니다. 가능하면 유사한 변수에는 동일한 이름을 사용하는 것이 좋습니다. 예를 들어, 변수에 IP 주소가 포함되어 있고 두 개의 종속 관계 파일이 있는 경우 두 파일 모두에 IP_ADDRESS라는 변수를 정의해야 합니다. 이렇게 하면 서로 다른 검색 어댑

터에서 이름이 IP_ADDRESS인 단일 대상 데이터를 사용할 수 있습니다. 자세한 내용은 "변수 값 지정"(101페이지)을 참조하십시오.

- 개념 정의는 개념 정의가 정의된 파일에서만 사용할 수 있습니다. 가능하면 유사한 용도의 개념에 대해서는 동일한 개념 이름 및 개념 변수를 사용하는 것이 좋습니다. 자세한 내용은 "개념 변수 값 지정"(101페이지)을 참조하십시오.
- 배포 가능 구성 요소가 여러 파일에 정의되어 있는 경우 서로 이름이 같을 수 있습니다. 이 경우에도 서로 다른 배포 가능 구성 요소로 처리됩니다.
- TQL 쿼리가 여러 파일에 정의되어 있는 경우 이름이 같아도 서로 다른 TQL 쿼리로 처리됩니다. 종속 관계 서명 파일에서 TQL 쿼리를 이름으로 참조하는 경우 해당 이름의 TQL 쿼리가 동일한 파일에 있어야 합니다.
- 범위가 여러 파일에 정의되어 있는 경우 서로 이름이 같을 수 있습니다. 이 경우에도 서로 다른 배포 범위로 처리됩니다. 종속 관계 서명 파일에서 범위를 이름으로 참조하는 경우 해당 이름의 범위가 동일한 파일에 있어야 합니다.
- 조건 함수가 여러 파일에 정의되어 있는 경우 서로 이름이 같을 수 있습니다. 종속 관계 서명 파일에서 범위를 이름으로 언급하는 경우 해당 이름의 함수가 동일한 파일에 있어야 합니다.
- 변수 및 개념 변수의 기본값은 해당 변수가 정의된 파일에서 고유합니다. 이름이 같은 두 개의 변수 또는 개념 변수는 다른 파일에서 기본값이 서로 다를 수 있습니다.
- 다양한 종속 관계 서명 파일에서 동일한 개념에 서로 다른 키 속성을 설정하는 것은 바람직하지 않습니다. 그렇게 하면 유지 관리가 어려울 뿐만 아니라 대상 데이터 변수에서 여러 어댑터의 값을 올바르게 할당하기 어렵습니다. 자세한 내용은 "개념 변수 값 지정"(101페이지)을 참조하십시오.

컴파일 오류

컴파일 유효성 검사

- 동일한 파일에 이름이 같은 배포 가능 구성 요소가 두 개 이상 있습니다.
- 동일한 파일에 이름이 같은 범위가 두 개 이상 있습니다.
- 동일한 파일에 이름이 같은 TQL 쿼리가 두 개 이상 있습니다.
- 동일한 파일에 이름이 같은 조건 함수가 두 개 이상 있습니다.
- 동일한 배포 가능 구성 요소에 이름이 같은 종속 관계가 두 개 있습니다.
- 구성 파일 조건에 동일한 파일의 글로벌 변수 또는 구성 파일이 속하는 종속 관계의 로컬 변수로 선언되지 않은 변수 이름을 사용합니다.
- 조건 함수에 글로벌 변수 또는 함수의 매개 변수로 선언되지 않은 변수 이름을 사용합니다.
- 변수 간 순환 종속성이 있습니다(예: 변수 VAR_A에 대한 삽입문에서 VAR_B의 값을 사용하고, VAR_B에 대한 삽입 값에서 변수 VAR_A의 값을 사용).
- 동일한 파일에 존재하지 않는 TQL 쿼리를 참조합니다.
- 동일한 파일에 존재하지 않는 범위를 참조합니다.
- 동일한 파일에 존재하지 않는 조건 함수를 참조합니다.

- 무시문에 기본값을 가지지 않는 변수를 사용하거나 대체 식에서 컴파일 오류가 발생합니다. 자세한 내용은 "변수의 기본값 사용"(76페이지)을 참조하십시오.
- 검색식에 기본값을 가지는 변수가 포함된 경우 해당 변수의 모든 조합에서 기본값을 사용하는 모든 사례를 처리해야 합니다. 이는 일부 변수에 대한 식을 완전히 무시하거나 대체 식을 사용하여 수행할 수 있습니다. 무시문에 표시되지 않는 모든 변수 조합에 대해 대체 식이 있어야 합니다. 그렇지 않은 경우 컴파일 오류가 발생합니다. 자세한 내용은 "변수의 기본값 사용"(76페이지)을 참조하십시오.
- <DocumentCILocation> 태그에 TQL 쿼리가 사용되지만 다음 규칙을 따르지 않는 경우
  - TQL 쿼리에서 배포 가능 구성 요소와 구성 문서 간의 경로를 지정해야 합니다. 이 경로는 단순해야 합니다(순환 없음).
  - TQL 쿼리에 다음과 같이 대/소문자를 구분하는 특정 이름이 있는 두 개의 끝 노드가 포함되어야 합니다.
    - Deployable - 경로의 배포 가능 구성 요소 CI
    - Configuration_document –구성 문서를 지정하는 경로의 CI
  - Deployable 및 Configuration_document 노드에는 조건을 사용할 수 없습니다.
  - 모든 노드 간 카디널리티는 1..1이어야 합니다.
  - 정규 링크 유형만 지원됩니다. 복합, 조인 또는 하위 그래프는 사용할 수 없습니다.
- 범위에서 TQL 쿼리가 사용되지만 다음 규칙을 따르지 않는 경우
  - 배포 가능 구성 요소에 대한 쿼리 노드를 "Deployable"이라고 해야 합니다(대/소문자 구분). 이 쿼리 노드의 결과는 해당 범위에 대해 가능한 배포 가능 구성 요소입니다.
  - TQL 쿼리에는 "Deployable" 이외의 기타 쿼리 노드가 하나만 있어야 합니다. 기타 쿼리 노드의 이름은 제한되지 않습니다. 이 노드를 Scope 쿼리 노드라고 합니다.
  - Deployable 및 Scope 쿼리 노드는 두 노드 사이에서 사용할 수 있는 모든 경로가 포함된 복합 링크로 연결되어 있습니다.
  - "배포 가능" 구성 요소는 동일한 범위에 있을 수 있는 소비자 구성 요소와 공급자 구성 요소를 나타냅니다. 따라서 쿼리 노드 CI 유형은 소비자와 공급자의 배포 가능 CI 유형에 대한 상위 항목이어야 합니다.
- 텍스트 구성 문서에 우선 순위가 다른 <ReferenceLocation> 태그가 여러 개 포함되는 경우. 자세한 내용은 "구성 문서 다시 정의"(79페이지)를 참조하십시오.

종속 관계 검색 어댑터

종속 관계 검색 프레임워크는 전용 어댑터로 실행해야 합니다. 어댑터는 다음을 수행합니다.

- 공급자의 관련 연결 문자열을 모두 수집합니다.
- 종속 관계 검색 프레임워크를 실행합니다.

- UCMDB에 검색 결과(종속 관계)를 보고합니다.

각 어댑터는 하나의 공급자 유형을 처리합니다. 예를 들어, 하나의 어댑터는 여러 JAR 공급자를 처리할 수 있습니다.

여러 어댑터에서 동일한 공급자 유형을 처리할 수도 있습니다. 그러나 이 경우에는 일관적인 결과를 얻을 수 있도록 어댑터별로 각 공급자를 한 번만 실행해야 합니다.

기타 어댑터와 마찬가지로 종속 관계 검색 어댑터가 준비되면 어댑터 논리를 실행할 디스커버리 작업을 만들어야 합니다.

이 섹션의 내용은 다음과 같습니다.

종속 관계 검색 어댑터 만들기

1. **데이터 흐름 관리 > 어댑터 관리**를 선택합니다.
2. **새로 만들기**  를 클릭하고 **새 어댑터**를 선택합니다.
3. 어댑터 세부 사항을 입력하고 **확인**을 클릭합니다.
4. 리소스 창에서 방금 만든 어댑터를 마우스 오른쪽 버튼으로 클릭하고 **어댑터 원본 편집**을 선택합니다.
5. 다음 줄을

   ```
   <taskInfo className="com.hp.ucmdb.discovery.probe.services.dynamic.core.DynamicService">
   ```

   다음 줄로 바꿉니다.

   ```
   <taskInfo
   className="com.hp.ucmdb.discovery.probe.services.dynamic.core.WorkflowService">
   ```

6. 다음 줄을

   ```
   <params
   className="com.hp.ucmdb.discovery.probe.services.dynamic.core.DynamicServiceParams"
   ignoreMissingReconciliationRules="false" enableRecording="false" enableAging="true"
   useDefaultValueForAging="false" autoDeleteOnErrors="success" recordResult="false" />
   ```

   다음 줄로 바꿉니다.

   ```
   <params
   className="com.hp.ucmdb.discovery.probe.services.dynamic.core.WorkflowServiceParams"
   patternType="workflow_adapter" enableAging="true"
   ```

```
ignoreMissingReconciliationRules="false" enableRecording="false"
autoDeleteOnErrors="success" recordResult="false" maxThreadRuntime="86400000"
useDefaultValueForAging="false">
  <workflow>
    <steps>
      <step name="Dependencies Discvoery" failure-policy="mandatory">
        <module
type="java">com.hp.ucmdb.discovery.probe.agents.probemgr.accuratedependencies.processi
ng.DependenciesDiscoveryWorkflowStep</module>
        <timeoutParking>
          <initialTimeout>180000</initialTimeout>
          <retriesThreshold>12</retriesThreshold>
          <multipleBy>2</multipleBy>
          <maxRetry>10</maxRetry>
          <timeoutThreshold>10800000</timeoutThreshold>
        </timeoutParking>
      </step>
    </steps>
    <finalStep />
    <libraryScripts />
  </workflow>
</params>
```

7. 종속 관계 서명 검색 결과를 처리할 스크립트를 준비합니다. 자세한 내용은 "Jython 스크립트 작성" (104페이지)을 참조하십시오.
8. 워크플로 어댑터에 다음 단계를 추가합니다.

```
<step name="Default Search Result Awaiting" failure-policy="mandatory">
  <module type="jython">[Your Jython Script Name]</module>
  <noParking />
</step>
```

9. 입력 창에서 **CI 유형 선택**  을 클릭하고 이 어댑터의 공급자 CI 유형을 선택합니다.
10. **입력 쿼리 편집**  을 클릭하고 입력 TQL 쿼리를 준비합니다. 자세한 내용은 "입력 TQL 쿼리 및 대상 데이터 정의"(100페이지)를 참조하십시오.
11. 디스커버리된 CIT 창에서  를 클릭하고 공급자 유형, 가능한 모든 소비자 유형 및 공급자-소비자 링크 유형을 추가합니다.
12. 마쳤으면 **저장**을 클릭합니다.

소비자-공급자 어댑터 정의

검색 기능은 디스커버리 어댑터에서 수행됩니다. 각 어댑터는 공급자 유형의 특정 연결 문자열 집합에 대해 검색을 처리합니다. 어댑터의 입력 TQL 쿼리는 해당 연결 문자열이 있는 위치에서 모든 CI를 가져오

고, 트리거의 대상 데이터를 사용하여 연결 문자열 변수 및 개념에 값을 삽입합니다. 자세한 내용은 "Jython 어댑터 개발"(36페이지)을 참조하십시오.

이러한 어댑터는 "워크플로 어댑터" 유형으로, 다음과 같이 검색을 수행하는 여러 단계를 실행하여 소비자-공급자 관계를 디스커버리합니다.

1. 어댑터에서 연결 문자열 값을 종속 관계 서명 글로벌 변수에 삽입하고 개념을 인스턴스화하여 범위의 구성 문서 필터에 대해 구체 검색식을 구성하는 논리를 실행합니다.
2. 어댑터에서 범위의 구체 검색식을 UCMDB 서버에서 실행 중인 UCMDB의 Solr 엔진에 제출합니다.
3. 종속 관계 서명 검색이 실행될 수 있도록 어댑터에서 UCMDB에 TQL 쿼리를 실행하여 필수 정보(예: 구성 문서, 배포 가능 설명자 등)를 모두 가져옵니다.
4. 어댑터에서 필수 구성 문서의 내용을 Data Flow Probe에 다운로드하고 저장합니다.
5. 그런 다음 2단계 및 3단계의 범위에서 발견한 공급자 및 소비자와 관련이 있는 종속 관계 검색을 실행합니다.
6. 그러면 Jython 스크립트에서 종속 관계 검색 결과를 보고합니다.

입력 TQL 쿼리 및 대상 데이터 정의

종속성 서명 범위 및 조건에서 검색식을 구성하려면 검색식에서 언급된 각 변수 및 개념을 구체적인 연결 문자열(TQL 쿼리에서 반환되는)로 대체해야 합니다. 이러한 대체 작업은 정의한 어댑터별 매핑에 따라 수행됩니다.

종속 관계 매핑 어댑터에 대한 입력 TQL 쿼리에서는 다음을 정의해야 합니다.

- 공급자(SOURCE 쿼리 노드)에 대한 소비자를 찾는 트리거 CI
- 특정 공급자에 필요한 모든 연결 문자열

이러한 TQL 쿼리에서는 CI 토폴로지 전반에 분산된 모든 연결 문자열을 반환하고 서비스를 제공하고 있는 공급자를 인식해야 합니다. 예를 들어, 공급자가 Oracle RAC인 경우 TQL 쿼리는 소비자가 RAC에 연결하여 해당 서비스를 사용하는 데 필요한 연결 문자열이 포함된 모든 CI를 반환해야 합니다. 따라서 TQL 쿼리의 레이아웃 정의에는 이러한 연결 문자열을 저장하는 모든 특성이 언급되어야 합니다. 예를 들어, RAC에 대한 TQL 쿼리는 RAC의 각 인스턴스에 액세스할 수 있는 IP 주소 및 포트와 함께 RAC 서비스 이름을 반환해야 합니다.

변수 및 개념은 어댑터의 대상 데이터 정의를 사용하여 입력 TQL 쿼리의 쿼리 노드 특성으로 매핑되어야 합니다. 각 패턴 요소는 이름이 고유해야 올바르게 매핑할 수 있습니다.

어댑터의 경우 검색 어댑터의 입력 TQL 쿼리는 이 어댑터와 연관된 작업의 트리거 TQL 쿼리와 일치하는 모든 트리거에 대해 실행됩니다.

어댑터를 서비스 디스커버리에 사용하는 경우 트리거를 서버에서 디스커버리된 트리거(자동으로 발생) 및 특정 경로로 비즈니스 서비스 CI에 연결되어 있는 트리거로 제한할 수 있습니다. 이를 위해 트리거와 비즈니스 서비스 CI 간의 경로를 포함하도록 입력 TQL 쿼리를 정의하고 비즈니스 서비스 CI "SERVICE"(대/소문자 구분)의 쿼리 노드 이름을 지정합니다. 그러면 프레임워크에서 "SERVICE" 쿼리 노드가 확인될 때 자동으로 해당 트리거 CI가 속하는 서비스로만 결과가 제한됩니다.

변수 값 지정

각 검색 어댑터는 어댑터에서 디스커버리해야 하는 종속 관계에 대해 종속 관계 서명의 검색 조건에 나타나는 모든 글로벌 변수 및 개념 값을 설정해야 합니다. 해당 값은 공급자의 연결 문자열을 나타내며(자세한 내용은 "변수 및 개념"(68페이지) 참조), 공급자(트리거)의 소비자를 찾는 데 사용됩니다.

참고: 매핑 특성 중 하나가 비어 있는 경우 다음 예에 표시된 대로 그대로 둡니다.

CONTEXT_ROOT = ${WEB_MODULE.j2eemanagedobject_contextroot:}

이 예에서 CONTEXT_ROOT는 종속 관계 서명에 빈 값을 수신하여 무시될 특성입니다.

변수 값을 설정하려면 다음을 수행합니다.

1. 정확한 변수 이름이 있는 대상 데이터 값을 추가합니다(대/소문자 구분).
2. 대상 데이터 값을 하드 코드된 값 또는 변수로 설정합니다.

자세한 내용은 "Jython 어댑터 개발"(36페이지)을 참조하십시오.

개념 변수 값 지정

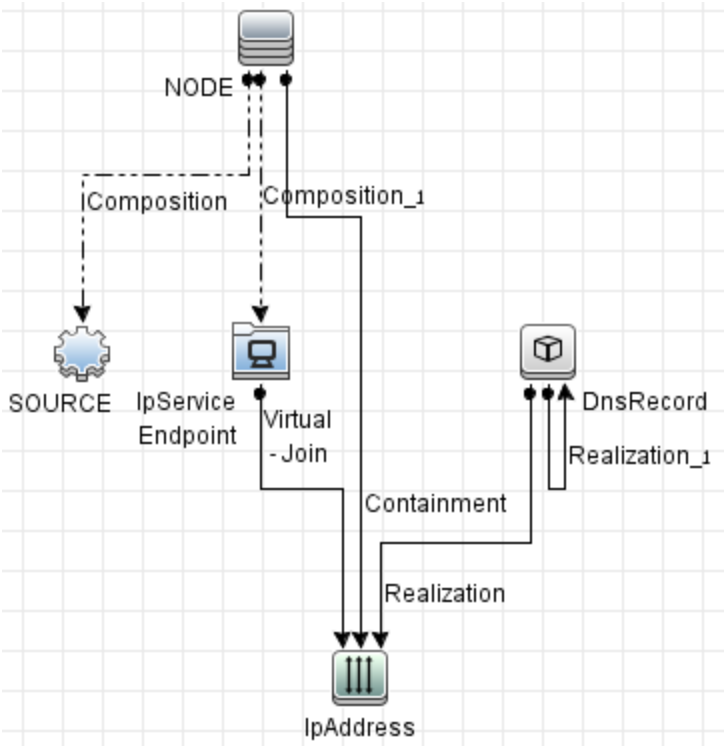
긴밀하게 연결되어 분리할 수 없는 연결 문자열 집합을 나타내도록 개념을 인스턴스화해야 합니다. 입력 TQL 쿼리에서 분리할 수 없는 연결 문자열 집합이 여러 개 반환될 수 있습니다. 예를 들어, 실행 중인 소프트웨어 인스턴스가 여러 개의 IP:포트 쌍에서 수신할 수 있습니다. 이러한 각 집합의 경우 별도의 개념 인스턴스를 인스턴스화해야 합니다. 개념의 키 특성은 개념 인스턴스를 분리하는 데 사용됩니다. 해당 키는 TQL 쿼리의 패턴 요소를 참조합니다. 키 패턴 요소에 반환된 각 CI 인스턴스에 대해 새 개념 인스턴스가 생성됩니다. 이러한 각 CI 인스턴스를 **키 CI 인스턴스**라고 합니다.

참고: TQL 쿼리에는 동일한 개념에 대한 매핑 정의가 여러 개 포함될 수 없습니다.

각 개념 인스턴스의 연결 문자열은 일치하는 키 CI 인스턴스 및 해당 키 CI에 연결된 CI에서 가져와야 합니다. 다음은 이러한 개념 정의가 포함된 종속 관계 서명 파일의 예입니다.

```
<Concept name="IpEndpoint">
  <Properties>
    <KeyProperty name="PORT"/>
    <Property name="IPADDRESS"/>
    <Property name="DNS"/>
  </Properties>
</Concept>
```

어댑터의 입력 TQL 쿼리는 다음과 같습니다.



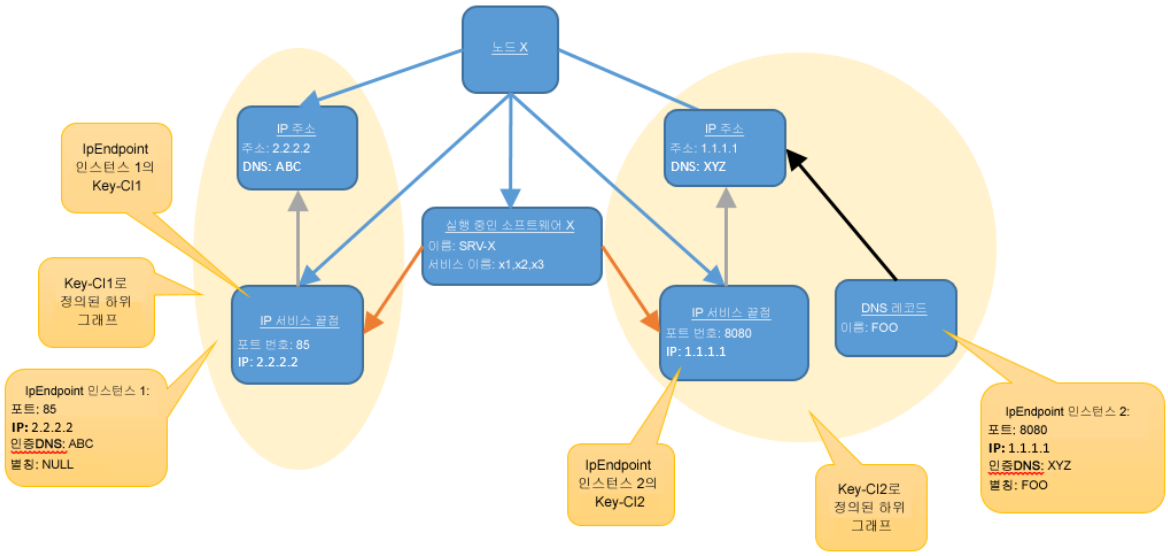
어댑터에서 개념을 인스턴스화하려면 변수와 유사하게 각 개념 변수를 대상 데이터 변수에 매핑해야 합니다. 자세한 내용은 "변수 값 지정"(101페이지)을 참조하십시오.

어댑터의 대상 데이터는 다음과 같습니다.

```
IpEndpoint.PORT = SOURCE.NODE.IpServiceEndpoint.name
IpEndpoint.IPADDRESS = SOURCE.NODE.IpServiceEndpoint.IpAddress.name
IpEndpoint.DNS = SOURCE.NODE.IpServiceEndpoint.IpAddress.DnsRecord.name
```

따라서 **IpServiceEndpoint** 쿼리 요소는 이 어댑터의 **IpEndpoint** 개념에 대한 키 CI입니다.

TQL 쿼리 그래픽(실행 중인 소프트웨어 X에 대한 TQL 쿼리의 결과를 표시) 예에서 두 개의 키 CI를 인식할 수 있습니다. 각 **IpEndpoint** 개념 인스턴스의 특성을 채울 연결 문자열은 각 키 CI의 하위 그래프에서 가져옵니다. 이러한 방식으로 각 개념 인스턴스는 분리할 수 없는 연결 문자열의 서로 다른 집합을 나타냅니다.



변수와 달리 개념 변수를 매핑할 때는 트리거의 쿼리 노드(이름이 항상 SOURCE로 지정됨)에서 변수를 바인딩할 쿼리 노드로의 경로를 입력 TQL 쿼리에 언급해야 합니다. 또한 키 변수가 설정되면 기타 모든 개념 변수 앞에 키 변수의 경로를 붙여야 합니다.

이 예제에서는 다음과 같습니다.

- 대상 데이터 변수의 이름은 개념 이름과 개념 변수의 이름이 차례로 표시된 이름입니다.
- 각 개념 이름을 대상 데이터 변수에 지정하여 동일한 어댑터에 여러 개념을 매핑할 수 있습니다.
- 경로는 항상 SOURCE로 시작합니다.
- 키 변수가 아닌 개념 변수 경로에는 키 변수에 대한 경로가 앞에 붙습니다.
- 경로는 링크 없이 쿼리 노드 이름으로만 구성됩니다. 그러나 경로에 지정된 두 쿼리 노드 이름 사이에는 링크가 하나 이상 있어야 합니다.

다음 항목 중 하나 이상이 경로에 있으면 해당 발송 단계에서 트리거할 수 없습니다.

- 입력 TQL 쿼리에 없는 쿼리 노드 이름
- 입력 TQL 쿼리에 링크되지 않은 두 개의 인접 쿼리 노드 이름
- 결과 CI 유형의 일부가 아닌 특성 이름

키 쿼리 노드가 아닌 쿼리 노드의 결과 CI가 여러 개 있는 경우 대상 데이터 변수는 이러한 CI의 모든 속성 값 목록이 됩니다. 목록은 동일한 개념의 기타 대상 데이터 변수에는 포함되지 않는 경로가 포함된 대상 데이터 변수에 대해서만 만들 수 있습니다. 그렇지 않으면 해당 발송 단계에서 트리거할 수 없습니다. 위에 표시된 예에서 IpEndpoint.IPADDRESS 대상 데이터 변수에는 목록이 포함될 수 없는데, 해당 경로가 IpEndpoint.DNS 대상 데이터 변수에 포함되기 때문입니다. IpEndpoint.DNS에는 목록이 포함될 수 있는데, 해당 경로가 다른 변수에 포함되지 않기 때문입니다. 즉, **IpServiceEndpoint** 쿼리 노드의 결과와 연결

되는 **IpAddress** 쿼리 노드의 결과 CI는 하나만 있을 수 있습니다. TQL 쿼리 및 경로를 계획할 때에는 이 점을 고려해야 합니다.

TQL 쿼리에 자체 링크가 있으면 키 쿼리 노드가 아닌 쿼리 노드에 결과 CI가 여러 개 있는 것과 유사하게 목록이 생성됩니다. 따라서 자체 링크가 있는 쿼리 노드에 대해서도 동일한 제한이 존재합니다.

자세한 내용은 "변수 및 개념"(68페이지)을 참조하십시오.

Jython 스크립트 작성

종속 관계 매핑 어댑터를 만드는 마지막 단계는 Jython 스크립트를 작성하여 결과를 보고하는 것입니다.

종속 관계 검색 결과에 액세스하려면 다음 코드 줄을 사용하여 워크플로 상태에서 검색해야 합니다.

```
workflowState = Framework.getWorkflowState()
searchResult = workflowState.getProperty(DependenciesDiscoveryConsts.DEPENDENCIES_
DISCOVERY_RESULT)
```

searchResult 변수는 검색 단계에 성공한 경우 null이 아닙니다. 워크플로에 검색 단계를 필수로 정의하여 검색 단계가 실패할 경우 Jython 스크립트 단계가 실행되지 않도록 하는 것이 좋습니다. 변수에는 **com.hp.ucmdb.discovery.probe.agents.probemgr.accuratedependencies.search.ConsumerDeployableSearchResult** 유형의 개체가 포함됩니다.

이 개체를 사용하여 다음을 수행합니다.

1. 종속 관계 검색으로 찾은 소비자 배포 가능 구성 요소를 모두 검토합니다. 주의할 점은 검색에서 공급자의 연결 문자열을 입력으로 사용하고 공급자에 종속적인 소비자 배포 가능 구성 요소를 모두 반환한다는 점입니다.
2. 각 소비자에 대해 종속 관계 서명이 두 개 이상 있을 수 있으며, 이에 대해서도 반복해야 합니다. 각 종속 관계 서명의 출력 변수 값은 서로 다를 수 있습니다. 사실상 종속 관계 서명에 사용된 모든 변수(글로벌 변수, 로컬 변수, 개념 변수) 값을 검색 결과의 Jython 스크립트에 사용할 수 있습니다.
3. 소비자와 공급자 간 링크가 포함된 개체 상태 소유자(OSH)를 만듭니다. 검색 결과 개체에서 공급자 세부 사항을 가져올 수도 있습니다.
4. OSH를 결과 벡터(OSHV)에 추가하고 결과를 UCMDB로 보냅니다.

다음은 위에서 언급한 흐름을 수행하는 Jython 스크립트 조각입니다.

```
# Get the search results object from the Workflow State
workflowState = Framework.getWorkflowState()
searchResult = workflowState.getProperty(DependenciesDiscoveryConsts.DEPENDENCIES_
DISCOVERY_RESULT)

# Prepare the OSHV that will contain the dependencies
oshv = ObjectStateHolderVector()
dependencyCount = 0

# Retrieve the provider OSH
```

```
providerServiceOsh = searchResult.getProviderDeployable().getDeployable()

# Loop through all the consumer deployable components
for index in range(0, searchResult.size()):
  deployable = searchResult.get(index)

  # Get the consumer deployable component's OSH
  deployableOsh = deployable.getDeployable()

  # Create an OSH for the dependency relationship between the consumer and provider
  consumerProviderLink = modeling.createLinkOSH('consumer_provider', deployableOsh,
providerServiceOsh)

  references = []

  # Iterate through all of the dependencies that were found between the consumer and the
provider in deployable.getDependencies():

    # Extract the name of the dependency as it appears in the dependency signature file
    dependencyName = dependency.getDependencyName()

    # Get the values of all the variables that were used by the dependency
    variables = dependency.getExportVariables()

    # Aggregate the values of the variable named REFERENCE
    # Note: The value of a variable can be a list if the variable contains multiple values
    dependencyNames.append(dependencyName)
    for var in variables:
      varName = var.getName()
      values = var.getValues()
      if varName.lower() == REFERENCES:
        references += list(values)

  reference = references and ','.join(references)
  if reference:
    consumerProviderLink.setAttribute(REFERENCES, reference)

  # Add the link to the results OSHV
  oshv.add(consumerProviderLink)

# Send the result to the UCMDB
Framework.sendObjects(oshv)
```

참고: 서비스 테두리 규칙(디스커버리 규칙 엔진의 규칙 집합)은 스크립트에서 보내는 정보를 기반으로 실행됩니다. 이러한 이유로 관계, 소비자 및 공급자에 대해 가능한 많은 정보를 보내는 것이 중요합니다. 위에 표시된 바와 같이 검색 결과 개체에서 검색한 소비자 및 공급자 OSH와의 관계를 만드는 것이 좋습니다. 이러한 OSH에는 서비스 테두리 규칙 및 기타 규칙 엔진 규칙이 제대로 작동되

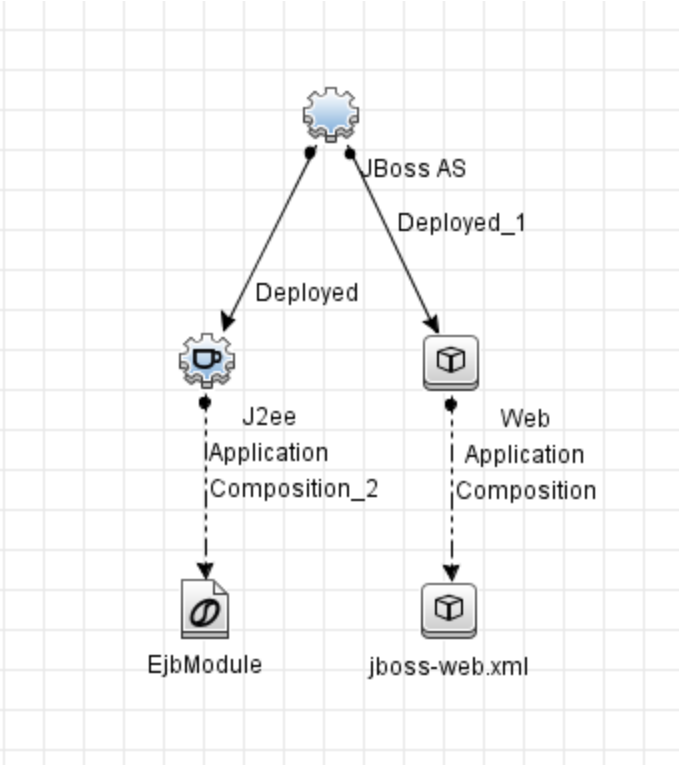
는 데 필요한 모든 정보가 포함됩니다.

어댑터 제한 사항

- 종속 관계 매핑 어댑터는 별도의 모드로 설치된 Data Flow Probe에 지원되지 않습니다.
- UCMDB 서버를 다시 시작할 때 시간 제한으로 인해 종속 관계 매핑 어댑터 작업을 트리거하지 못할 수 있습니다. 예약된 다음 실행 기간 동안 트리거에 성공해야 합니다.

전체 예

이 섹션에는 동일한 JBoss AS에 배포된 J2ee 응용 프로그램과 웹 응용 프로그램 간의 종속 관계를 찾는 종속 관계 서명 및 어댑터를 개발하는 방법에 대한 전체 예가 나와 있습니다.



이 섹션에는 다음 항목이 포함됩니다.

개발 워크플로

종속 관계 서명 및 어댑터를 개발하는 일반적인 흐름은 다음과 같습니다.

1. 개발할 서명의 공급자 CI 유형을 파악합니다. 이 예제에서는 공급자 유형이 J2ee 응용 프로그램입니다.

2. 포함할 공급자와 관련된 소비자 배포 가능 구성 요소를 결정합니다. 서명은 소비자의 구성 문서에 따라 다르므로 각 소비자는 고유의 종속성 서명을 보유하게 됩니다.

   이 예제에서는 소비자 배포 가능 구성 요소 중 하나인 JBoss의 웹 응용 프로그램을 선택합니다.

3. 이러한 서명을 기존의 종속성 서명 파일에 추가할지 또는 종속성 서명 파일을 새로 만들지 여부를 결정합니다.

   두 방법에 기능적 차이는 없습니다. 종속 관계 서명을 쉽게 유지 관리할 수 있는 옵션을 선택합니다. 예를 들어, 종속 관계 서명 파일 중 하나에 소비자 배포 가능 구성 요소가 이미 있는 경우 이러한 새 종속 관계를 기존의 배포 가능 구성 요소에 추가하면 더 쉽게 유지 관리할 수 있습니다. 이 예제에서는 새 종속성 서명 파일을 만들어 새 종속성 서명을 포함합니다.

4. 새 어댑터가 필요한지 또는 공급자에 기존 어댑터가 이미 적용되고 있는지 여부를 확인합니다. 검색 어댑터의 트리거 CI는 공급자 CI 유형임에 유의하고 다음을 확인해야 합니다.

   a. 이 CI 유형이 트리거 CI인 기존 검색 어댑터가 있는가?

      없는 경우 그러한 어댑터를 만들어야 합니다. 이 예제에서는 이를 수행합니다.

   b. 검색 어댑터가 있는 경우 필수 연결 문자열 변수 및 개념을 모두 대상 데이터로 매핑해야 하는가?

      그렇지 않은 경우 기존 어댑터를 업데이트하고 해당 변수를 대상 데이터에 추가해야 합니다. 대상 데이터는 입력 TQL 쿼리에서 추출됩니다. 따라서 새로운 정보를 가져오기 위해 입력 TQL 쿼리를 변경해야 할 수 있습니다.

5. 어댑터에 대한 작업이 없는 경우 작업을 준비합니다.
6. 트리거 TQL 쿼리에서 공급자도 트리거로 가져오는지 확인합니다.

종속 관계 서명 개발

JBoss J2EE 응용 프로그램 및 웹 응용 프로그램 간의 종속 관계 서명을 개발합니다.

1. 소비자와 공급자 사이의 종속 관계를 만드는 데 필요한 연결 문자열을 파악합니다. 이 예에서 필요한 연결 문자열은 공급자 J2EE 응용 프로그램에 포함된 EJB 모듈에 있는 EJB의 JNDI 이름입니다.
2. 모든 연결 문자열을 글로벌 변수 또는 개념으로 정의해야 합니다. 이러한 변수 값을 어댑터로 삽입합니다. 그러나 종속성 서명 파일(신규 또는 기존)에 해당 변수를 정의하기 전에 모든 파일에서 유사한 변수 및 개념이 있는지 확인합니다. 있는 경우 변수, 개념 및 개념 변수에 기존 항목과 동일한 이름을 지정합니다. 이렇게 하면 삽입해야 하는 글로벌 변수 수가 늘어나지 않으므로 더욱 쉽게 어댑터를 개발할 수 있습니다. 이 예제에서는 새 파일에 이러한 정의를 추가하므로 파일이 다음과 같습니다.

   ```
   <VariableDeclarations>
     <Variable name="EJB_JNDI_NAME"/>
   </VariableDeclarations>
   ```

3. 구성 파일을 분석하고 다음을 식별합니다.

- 각 구성 파일에 있는 각 연결 문자열의 위치
- 각 파일의 유형: 속성, XML 또는 기타 텍스트 파일
- 각 파일 간 관계(파일이 여러 개 있는 경우)

이 예제에서는 다음과 같이 JBoss 웹 응용 프로그램에서 **jboss-web.xml** 구성 파일에 EJB 참조를 정의합니다.

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
  <!-- A reference to an EJB in the same server with a custom JNDI binding -->
  <ejb-ref>
    <ejb-ref-name>ejb/BHome</ejb-ref-name>
    <jndi-name>someapp/ejbs/beanB</jndi-name>
  </ejb-ref>
  <!-- A reference to an EJB in an external server -->
  <ejb-ref>
    <ejb-ref-name>ejb/RemoteBHome</ejb-ref-name>
    <jndi-name>jnp://otherserver/application/beanB</jndi-name>
  </ejb-ref>
</jboss-web>
```

<ejb-ref-name> 섹션의 값은 연결 문자열의 참조에 해당합니다. 이는 XML 파일이므로 다음 XPath 식을 사용하여 EJB의 JNDI 이름을 일치시킬 수 있습니다.

```
//jboss-web/ ejb-ref/ ejb-ref-name[matches(., '^${EJB_JNDI_NAME}$')]
```

4. 기본 검색식이 포함된 범위를 정의해야 합니다. 이는 종속 관계 매핑에서 필요한 구성 파일을 찾는 데 도움이 됩니다. 이 예제에서는 JNDI 이름 자체가 키워드입니다. 다음은 범위 정의입니다.

```
<ScopeDefinitions>
  <Scope name="JBoss_EJB_Same_Cell">
    <ConfigurationDocumentContentFilter>
      <Operator type="and">
        <Operand value="${EJB_JNDI_NAME}"/>
      </Operator>
    </ConfigurationDocumentContentFilter>
  </Scope>
</ScopeDefinitions>
```

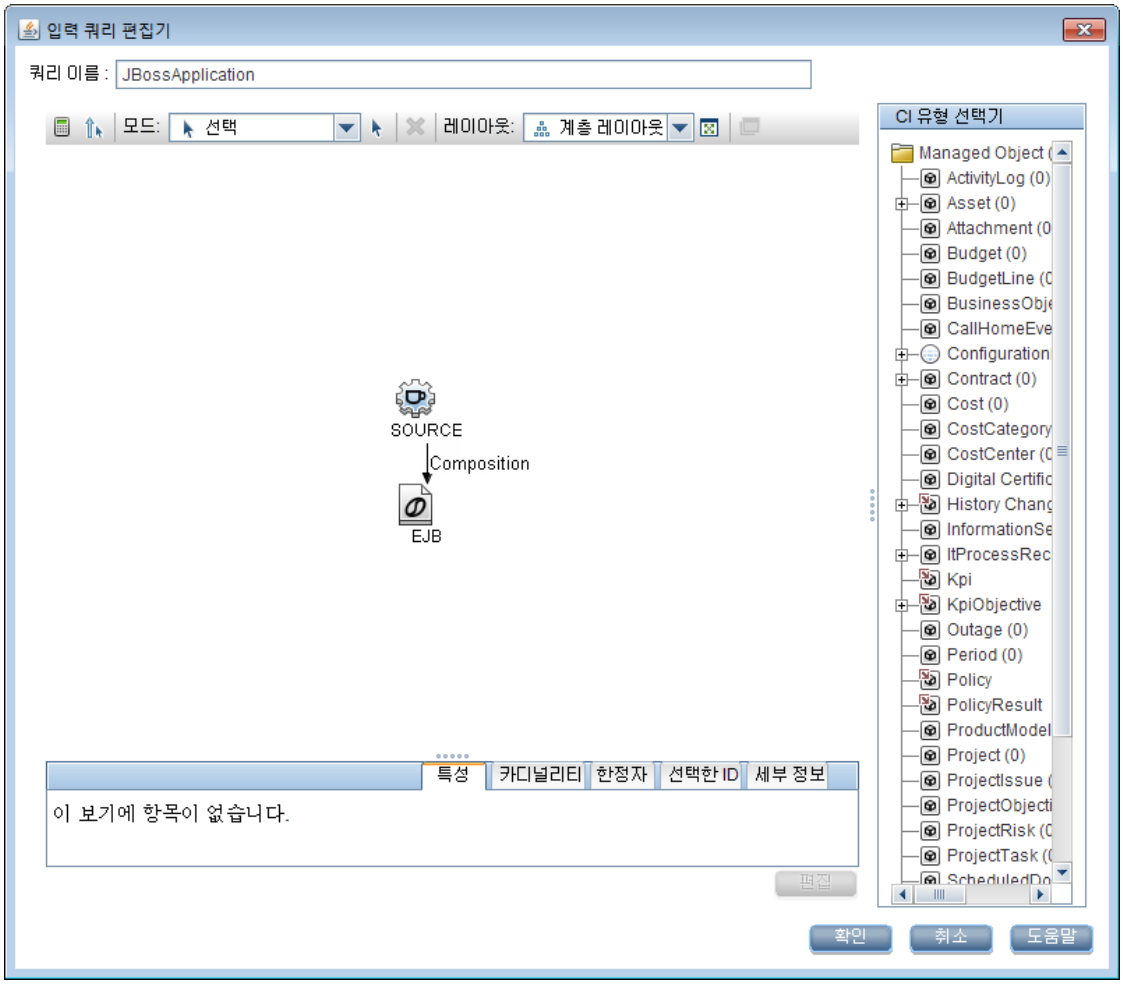
전체 종속 관계 서명 예는 다음과 같습니다.

```
<?xml version="1.0"?>
<DependencySignatures xmlns="http://www.hp.com/ucmdb/1-0-0/Dependencies">
  <VariableDeclarations>
    <Variable name="EJB_JNDI_NAME"/>
```

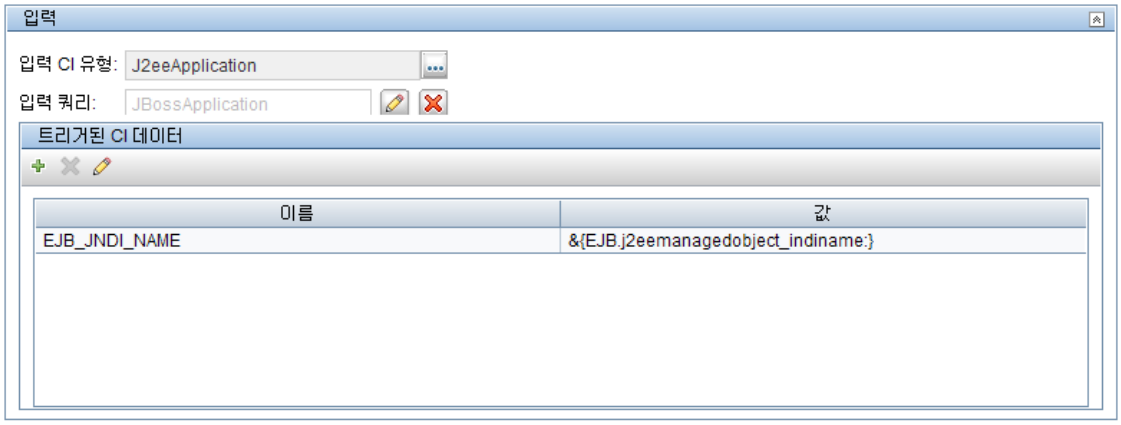
```
  </VariableDeclarations>
  <Deployable name="JBoss J2EE Application to Web Application by JNDI" >
    <Descriptor cit="webapplication"/>
    <Dependency name="J2EE Application with Internal EJB" providerCiType="j2eeapplication"
scope="JBoss_EJB_Same">
      <XmlConfigurationDocument name="jboss-web.xml">
        <조건>
          <Operator type="or">
            <XPathCondition>
              <XPath>//ejb-ref/jndi-name[matches(., '^${EJB_JNDI_NAME}$', 'i')]</XPath>
            </XPathCondition>
          </Operator>
        </Condition>
      </XmlConfigurationDocument>
    </Dependency>
  </Deployable>
  <ScopeDefinitions>
    <Scope name="JBoss_EJB_Same_Cell">
      <ConfigurationDocumentContentFilter>
        <Operator type="and">
          <Operand value="${EJB_JNDI_NAME}"/>
        </Operator>
      </ConfigurationDocumentContentFilter>
    </Scope>
  </ScopeDefinitions>
</DependencySignatures>
```

어댑터 개발

1. "종속 관계 검색 어댑터 만들기"(98페이지)에 설명된 대로 새 워크플로 어댑터를 만듭니다.
2. 트리거 CI 유형을 공급자 유형으로 설정합니다. 이 예제에서는 J2EE 응용 프로그램이 공급자입니다.
3. 어댑터에 입력 TQL 쿼리를 정의하여 필요한 CI 및 해당 특성을 가져옵니다. 이 예제에서는 트리거 J2EEApplication CI와 함께, 공급자 J2EE 응용 프로그램에 속하는 EJBModule CI가 필요합니다.



4. 필요한 연결 문자열 변수 및 개념에 대한 서명에 정의된 글로벌 변수를 트리거 CI 데이터를 사용하여 매핑합니다. 이 예제에서는 j2eemanagedobject_jndiname 특성을 EJBModule CI에서 EJB_JNDI_NAME 대상 데이터로 매핑합니다.



5. 워크플로 단계 섹션에서 다음의 워크플로 정의를 붙여 넣습니다.

```
<workflow>
  <steps>
    <step name="Accurate Dependency Search" failure-policy="mandatory">
      <module type="jython">DependenciesDiscovery.py</module>
      <timeoutParking>
        <initialTimeout>60000</initialTimeout>
        <retriesThreshold>1</retriesThreshold>
        <multipleBy>1</multipleBy>
        <maxRetry>20</maxRetry>
        <timeoutThreshold>60000</timeoutThreshold>
      </timeoutParking>
    </step>
  </steps>
  <finalStep>
    <module type="jython">AccurateDependencyMapping.py</module>
  </finalStep>
  <libraryScripts />
</workflow>
```

timeoutParking 매개 변수를 조정하여 시간 제한 기간을 변경할 수 있습니다.

6. 새 어댑터에 대한 트리거 TQL 쿼리를 만듭니다.
7. 다음과 같이 서비스 디스커버리 활동 유형에 작업 정의를 추가합니다.

```
<ServiceDiscoveryActivityType id="top-down" displayName="Top-down">
  <JobsDefinitions>
    ...
     <job id=" JBoss Application to Web Application " displayName=" JBoss Application to Web
Application ">
       <patternId>JBossApplication2WebApplication</patternId>
       <triggers>
         <trigger>jboss_application_trigger</trigger>
       </triggers>
       <parameters/>
     </job>
    ...
  </JobsDefinitions>
</ServiceDiscoveryActivityType>
```

5장: 일반 데이터베이스 어댑터 개발

이 장의 내용:

일반 데이터베이스 어댑터 개요

일반 데이터베이스 어댑터 플랫폼의 목적은 RDBMS(관계형 데이터베이스 관리 시스템)와 통합하고 데이터베이스에 대해 TQL 쿼리 및 채우기 작업을 실행할 수 있는 어댑터를 만드는 것입니다. 일반 데이터베이스 어댑터에서 지원하는 RDBMS로는 Oracle, Microsoft SQL Server, MySQL이 있습니다.

이 버전의 데이터베이스 어댑터 구현은 지속성 공급자로 Hibernate ORM 라이브러리를 사용하는 JPA (Java Persistence API) 표준에 기반합니다.

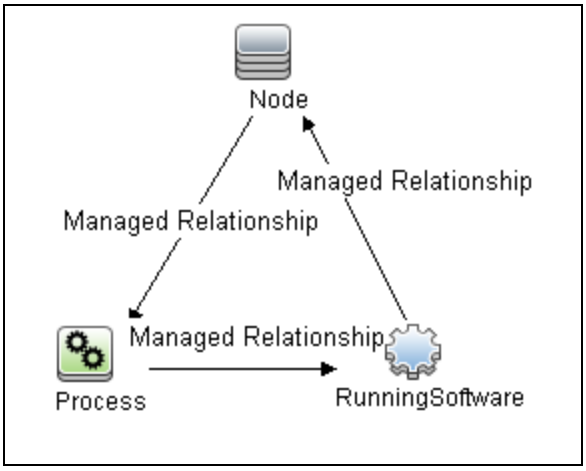
일반 데이터베이스 어댑터의 TQL 쿼리

채우기 작업의 경우, 모델링 스튜디오의 레이아웃 설정 대화 상자에서 필요한 모든 CI 레이아웃을 선택해야 합니다. 자세한 내용은 *HP Universal CMDB 모델링 안내서*에서 쿼리 노드 속성/관계 속성 대화 상자를 참조하십시오. CI에서 특성을 식별해야 하며 이러한 특성이 없으면 CI를 UCMDB에 추가할 수 없습니다.

다음 제한은 일반 데이터베이스 어댑터에서 계산한 TQL 쿼리에만 해당합니다.

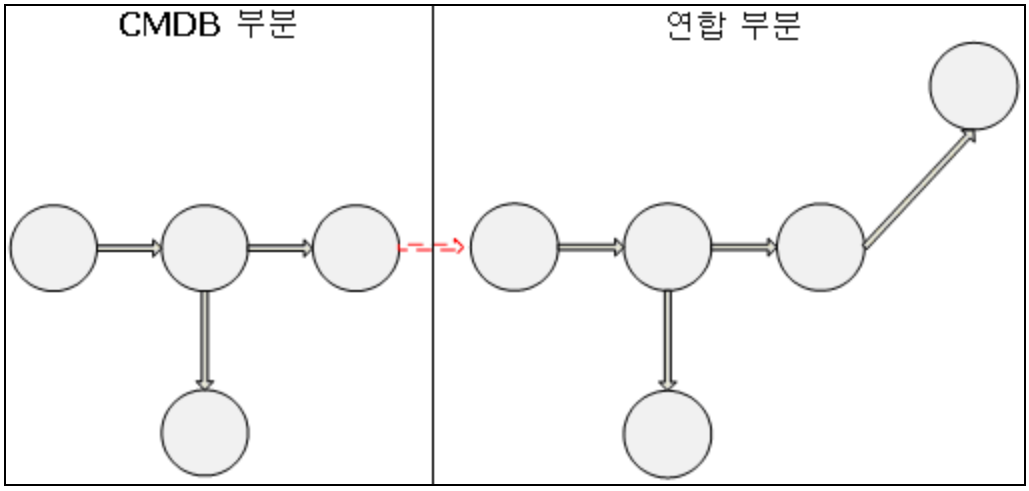
- 하위 그래프는 지원되지 않습니다.
- 복합 관계는 지원되지 않습니다.
- 주기 또는 주기 일부분은 지원되지 않습니다.

  다음 TQL 쿼리는 주기의 예입니다.



- 함수 레이아웃은 지원되지 않습니다.
- 0..0 카디널리티는 지원되지 않습니다.
- 조인 관계는 지원되지 않습니다.
- 한정자 조건은 지원되지 않습니다.

- 두 CI 사이를 연결하려면 외부 데이터베이스 원본에 테이블 형태의 관계나 외래 키가 있어야 합니다.



조정

조정은 어댑터 쪽의 TQL 계산 중에 수행됩니다. 조정이 발생할 수 있도록 CMDB 쪽이 조정 CIT라는 연합 엔터티에 매핑됩니다.

매핑. CMDB의 각 특성이 데이터 원본의 열에 매핑됩니다.

매핑이 바로 완료되더라도 매핑 데이터에 대한 변환 함수도 지원됩니다. 새 함수는 Java 코드(예: lowercase, uppercase)를 통해 추가할 수 있습니다. 이러한 함수는 값(CMDB와 연합 데이터베이스에 서로 다른 형식으로 저장되어 있는 값)을 변환하기 위해 사용됩니다.

참고:

- CMDB와 외부 데이터베이스 원본을 연결하려면 데이터베이스에 적절한 연관이 있어야 합니다. 자세한 내용은 "선행 조건"(117페이지)을 참조하십시오.
- CMDB ID와의 조정도 지원됩니다.
- 글로벌 ID와의 조정도 지원됩니다.

Hibernate를 JPA 공급자로 사용

Hibernate는 몇 가지 종류의 관계형 데이터베이스(예: Oracle 및 Microsoft SQL Server)를 통해 Java 클래스를 테이블에 매핑할 수 있는 OR(개체-관계) 매핑 도구입니다. 자세한 내용은 "기능 제한"(180페이지)을 참조하십시오.

기본 매핑에서 각 Java 클래스는 단일 테이블에 매핑됩니다. 한층 더 고급 단계의 매핑을 사용하면 상속 매핑(CMDB 데이터베이스에서 발생 가능)이 가능합니다.

기타 지원되는 기능에는 여러 테이블에 클래스 하나를 매핑하는 기능, 컬렉션 지원 기능 및 일대일, 일대다, 다대일 유형의 연관 기능이 있습니다. 자세한 내용은 아래의 "연관"(116페이지)을 참조하십시오.

여기서 설명하려는 작업에서는 Java 클래스를 만들 필요가 없습니다. 매핑은 CMDB 클래스 모델 CIT에서 데이터베이스 테이블 방향으로 정의됩니다.

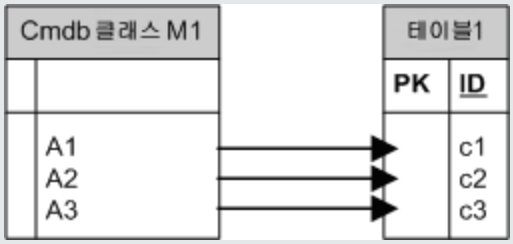
이 섹션에는 다음 항목도 포함됩니다.

개체-관계 매핑의 예

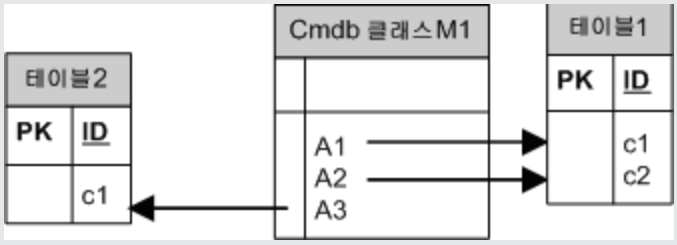
다음 예는 개체-관계 매핑을 설명합니다.

하나의 데이터베이스 테이블에 매핑된 하나의 CMDB 클래스의 예:

A1, A2, A3 특성이 있는 클래스 M1이 c1, c2, c3 열이 있는 테이블 1에 매핑되었습니다. 이는 임의의 M1 인스턴스가 테이블 1에 일치하는 행을 가지고 있음을 의미합니다.

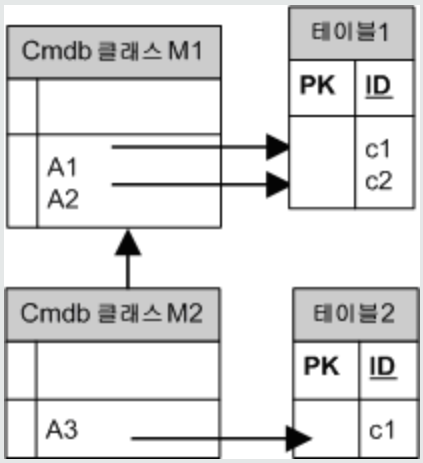


두 개의 데이터베이스 테이블에 매핑된 하나의 CMDB 클래스의 예:



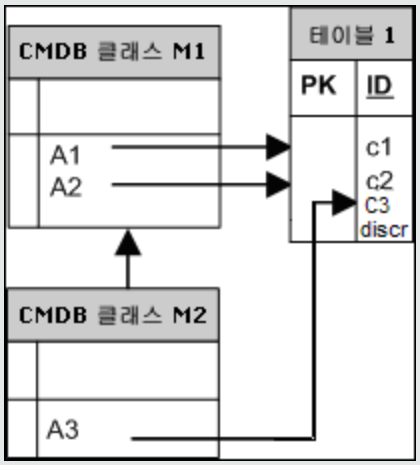
상속의 예:

이 사례는 각 클래스가 고유한 데이터베이스 테이블을 가지고 있는 CMDB에 사용됩니다.



판별자가 있는 단일 테이블 상속의 예:

클래스의 전체 계층 구조가 단일 데이터베이스 테이블에 매핑됩니다. 이 테이블의 열은 매핑된 클래스의 전체 특성으로 이루어진 상위 집합을 구성합니다. 해당 테이블에는 추가 열(판별자)도 포함되는데, 이 열의 값은 이 항목에 매핑되어야 하는 특정 클래스를 나타냅니다.



연관

연관에는 일대다, 다대일, 다대다 등 세 가지 유형이 있습니다. 서로 다른 데이터베이스 개체 사이를 연결하려면 외래 키 열(일대다의 경우)이나 매핑 테이블(다대다의 경우)을 사용하여 이러한 연관 중 하나를 정의해야 합니다.

사용성

JPA 스키마는 매우 광범위하므로 쉽게 연결을 정의할 수 있도록 간소화된 XML 파일이 제공됩니다.

이 XML 파일을 사용할 수 있는 사용 사례는 다음과 같습니다. 연합 데이터가 하나의 연합 클래스로 모델링되었습니다. 이 클래스에는 연합 클래스가 아닌 CMDB 클래스에 대한 다대일 관계가 있습니다. 그리고 연합 클래스와 연합 클래스가 아닌 클래스 간에 가능한 관계 유형이 하나뿐입니다.

어댑터 만들기 준비

이 작업에서는 어댑터를 만드는 데 필요한 준비 사항을 설명합니다.

참고: UCMDB API에서 일반 DB 어댑터의 샘플을 볼 수 있습니다. 특히, DDMi 어댑터 샘플에는 복잡한 **orm.xml** 파일과 일부 플러그인 인터페이스에 대한 구현이 포함되어 있습니다.

이 작업에는 다음 단계가 포함됩니다.

- "선행 조건"(117페이지)
- "CI 유형 만들기"(119페이지)
- "관계 만들기"(119페이지)

1. 선행 조건

   데이터베이스에 데이터베이스 어댑터를 사용할 수 있는지 유효성을 검사하려면 다음 사항을 확인하십시오.

   - 데이터베이스에 조정 클래스 및 그 특성(multinode라고도 함)이 있습니다. 예를 들어 노드 이름에 따라 조정을 실행하는 경우 노드 이름이 있는 열이 포함된 테이블이 있는지 확인합니다. cmdb_id 노드에 따라 조정을 실행하는 경우에는 CMDB에 있는 노드의 CMDB ID와 일치하는 CMDB ID가 있는 열이 있는지 확인합니다. 조정에 대한 자세한 내용은 "조정"(114페이지)을 참조하십시오.

ID	NAME	IP_ADDRESS
31	BABA	16.59.33.60
33	ext3.devlab.ad	16.59.59.116
46	LABM1MAM15	16.59.58.188
72	cert-3-j2ee	16.59.57.100
102	labm1sun03.devlab.ad	16.59.58.45
114	LABM2PCOE73	16.59.66.79
116	CUT	16.59.41.214
117	labm1hp4.devlab.ad	16.59.60.182

   - 관계가 있는 두 CIT를 상관 관계로 연결하려면 CIT 테이블 간에 상관 관계 데이터가 있어야 합니다. 상관 관계는 외래 키 열이나 매핑 테이블을 기준으로 지정할 수 있습니다. 예를 들어 노드와

티켓 간에 상관 관계를 만들려면 티켓 테이블에 노드 ID가 포함된 열이 있거나, 노드 테이블에 티켓 ID가 연결된 열이 있거나, end1이 노드 ID이고 end2가 티켓 ID인 매핑 테이블이 있어야 합니다. 상관 관계 데이터에 대한 자세한 내용은 "Hibernate를 JPA 공급자로 사용"(114페이지)을 참조하십시오.

다음 표는 외래 키 NODE_ID 열을 보여 줍니다.

NODE_ID	CARD_ID	CARD_TYPE	CARD_NAME
2015	1	직렬 버스 컨트롤러	Intel 82801EB USB Universal Host Controller
3581	2	시스템	Intel 631xESB/6321ESB/3100 칩셋 LPC
3581	3	디스플레이	ATI ES1000
3581	4	기본 시스템 주변 장치	HP ProLiant iLO 2 레거시 지원 기능

- 각 CIT는 하나 이상의 테이블에 매핑될 수 있습니다. CIT 하나를 둘 이상의 테이블에 매핑하려면 기본 키가 다른 테이블에 있는 기본 테이블과 고유한 값 열이 있는지 확인하십시오.

  예를 들어 티켓은 ticket1과 ticket2 등의 두 테이블에 매핑됩니다. 첫 번째 테이블에는 c1 및 c2 열이 있고 두 번째 테이블에는 c3 및 c4 열이 있습니다. 이 둘을 하나의 테이블로 간주할 수 있도록 하려면 두 테이블에 동일한 기본 키가 있어야 합니다. 아니면, 첫 번째 테이블 기본 키가 두 번째 테이블의 열일 수 있습니다.

  다음 예에서는 테이블 간에 CARD_ID라는 같은 기본 키를 공유합니다.

CARD_ID	CARD_TYPE	CARD_NAME
1	직렬 버스 컨트롤러	Intel 82801EB USB Universal Host Controller
2	시스템	Intel 631xESB/6321ESB/3100 칩셋 LPC
3	표시	ATI ES1000
4	기본 시스템 주변 장치	HP ProLiant iLO 2 레거시 지원 기능

CARD_ID	CARD_VENDOR
1	Hewlett-Packard Company
2	(표준 USB 호스트 컨트롤러)
3	Hewlett-Packard Company

CARD_ID	CARD_VENDOR
4	(표준 시스템 장치)
5	Hewlett-Packard Company

2. CI 유형 만들기

   이 단계에서는 RDBMS(외부 데이터 원본)의 데이터를 나타낸 CIT를 만듭니다.

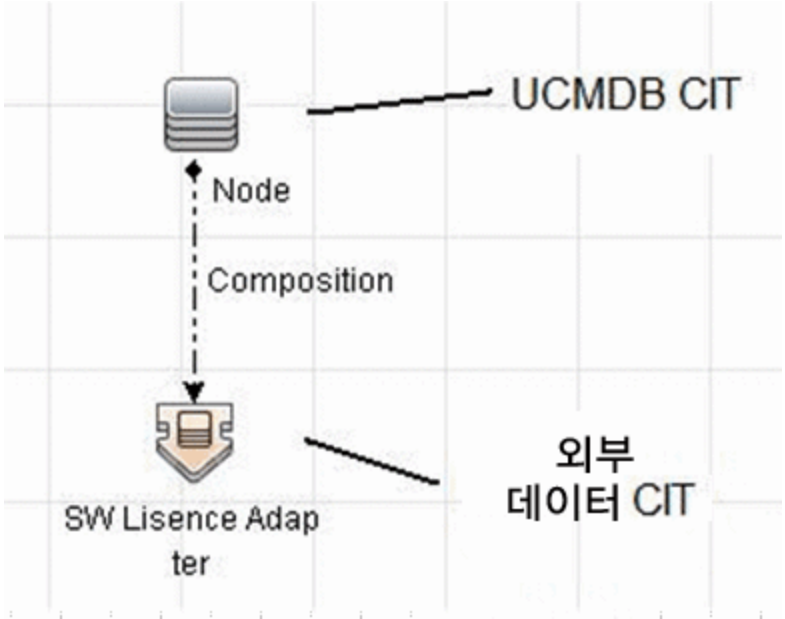
   a. UCMDB에서 CI 유형 관리자에 액세스하고 새 CI 유형을 만듭니다. 자세한 내용은 *HP Universal CMDB 모델링 안내서*에서 CI 유형을 만드는 방법을 참조하십시오.

   b. CIT에 필요한 특성(예: 마지막 액세스 시간, 벤더 등)을 추가합니다. 이러한 특성은 어댑터가 외부 데이터 원본에서 검색하여 CMDB 보기로 가져올 특성입니다.

3. 관계 만들기

   이 단계에서는 UCMDB CIT와 외부 데이터 원본에서 데이터를 나타내는 새 CIT 간에 관계를 추가합니다.

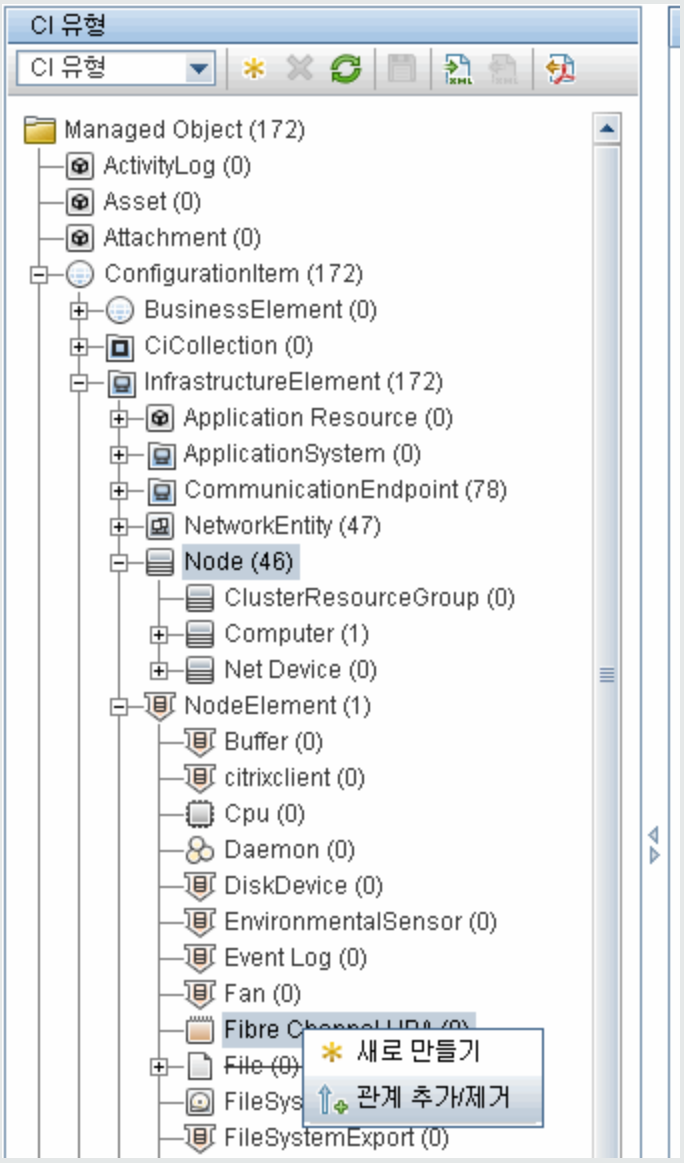
   새 CIT에 적절하고 유효한 관계를 추가합니다. 자세한 내용은 *HP Universal CMDB 모델링 안내서*에서 관계 추가/제거 대화 상자를 참조하십시오.



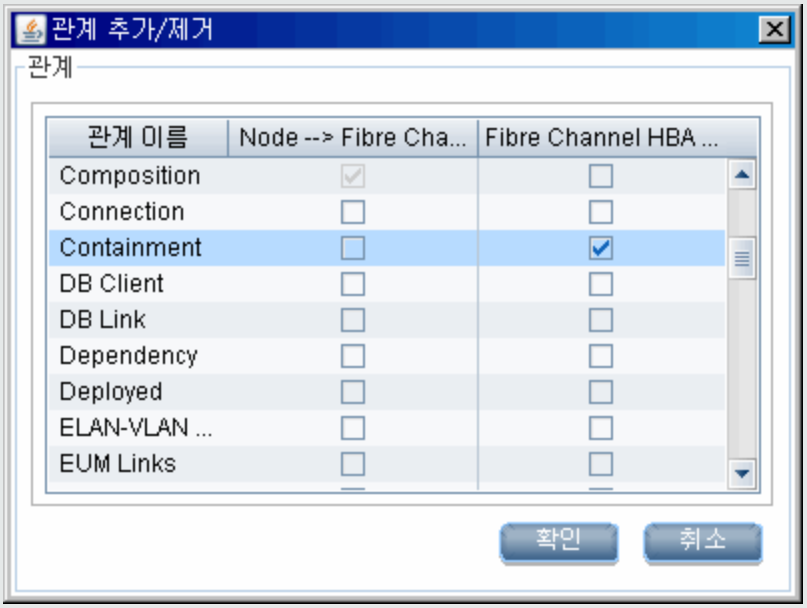
참고: 이 단계에서는 데이터에 가져올 메서드를 아직 정의하지 않았기 때문에 아직 연합 데이터를 보거나 외부 데이터를 채울 수 없습니다.

Containment 관계를 만드는 작업의 예:

a. CIT 관리자에서 다음 두 CIT를 선택합니다.



b. 두 CIT 간에 **Containment** 관계를 만듭니다.



어댑터 패키지 준비

이 단계에서는 일반 DB 어댑터 패키지를 적절한 위치에 넣어 구성합니다.

1. **C:\hp\UCMDB\UCMDBServer\content\adapters** 폴더에서 **db-adapter.zip** 패키지를 찾습니다.
2. 로컬 임시 디렉터리에 패키지의 압축을 풉니다.
3. 다음과 같이 어댑터 XML 파일을 편집합니다.
   - 텍스트 편집기에서 **discoveryPatterns\db_adapter.xml** 파일을 엽니다.
   - 다음과 같이 **adapter id** 특성을 찾아 이름을 바꿉니다.

```
<pattern id="MyAdapter" displayLabel="My Adapter"
xsi:noNamespaceSchemaLocation="../../Patterns.xsd" description="Discovery Pattern
Description"
      schemaVersion="9.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
displayName="UCMDB API Population">
```

어댑터에서 채우기를 지원하는 경우 **<adapter-capabilities>** 요소에 다음 기능을 추가해야 합니다.

```
      <support-replicatioin-data>
        <source>
```

```
        <changes-source>
      </source>
    </support-replicatioin-data>
```

HP Universal CMDB의 통합 포인트 창에 있는 어댑터 목록에 표시 레이블 또는 ID가 나타납니다.

일반 DB 어댑터를 만들 때 **support-replicatioin-data** 태그에서 **changes-source** 태그를 편집할 필요가 없습니다. **FcmdbPluginForSyncGetChangesTopology** 플러그인을 구현하면 마지막 실행 후로 변경된 토폴로지가 반환됩니다. 플러그인이 구현되지 않은 경우에는 전체 토폴로지가 반환되고 반환된 CI에 따라 자동 삭제가 수행됩니다.

CMDB을 데이터로 채우는 방법에 대한 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 "통합 스튜디오 페이지"를 참조하십시오.

- 어댑터가 버전 8.x의 매핑 엔진을 사용하는 경우(즉, 새 조정 매핑 엔진을 사용하지 않는 경우) 다음 요소를

  <default-mapping-engine>

  다음 요소로 바꿉니다.

  <default-mapping-engine>com.hp.ucmdb.federation.
  mappingEngine.AdapterMappingEngine</default-mapping-engine>

  새 매핑 엔진으로 되돌리려면 요소를 다음 값으로 되돌립니다.

  <default-mapping-engine>

- **범주** 정의를 찾습니다.

  <category>Generic</category>

  Generic 범주 이름을 선택한 범주로 변경합니다.

  참고: 범주가 **Generic**으로 지정된 어댑터는 새 통합 포인트를 만들 때 통합 스튜디오에 나열되지 않습니다.

- 데이터베이스와의 연결은 사용자 이름(스키마), 비밀번호, 데이터베이스 유형, 데이터베이스 호스트 시스템 이름 및 데이터베이스 이름 또는 SID를 사용하여 설명할 수 있습니다.

  이러한 연결 유형에 대해 매개 변수는 어댑터 XML 파일의 **parameter** 섹션에 다음과 같은 요소를 갖습니다.

```
<parameters>
  <!--The description attribute may be written in simple text or HTML.-->
  <!--The host attribute is treated as a special case by UCMDB-->
  <!--and will automatically select the probe name (if possible)-->
  <!--according to this attribute's value.-->
  <!--Display name and description may be overwritten by I18N values-->
```

```
    <parameter name="host" display-name="Hostname/IP" type="string" description="The
host name or IP address of the remote machine" mandatory="false" order-index="10" />
    <parameter name="port" display-name="Port" type="integer" description="The
remote machine's connection port" mandatory="false" order-index="11" />
    <parameter name="dbtype" display-name="DB Type" type="string" description="The
type of database" valid-values="Oracle;SQLServer;MySQL;BO" mandatory="false" order-
index="13">Oracle</parameter>
    <parameter name="dbname" display-name="DB Name/SID" type="string"
description="The name of the database or its SID (in case of Oracle)"  mandatory="false"
order-index="13" />
    <parameter name="credentialsId" display-name="Credentials ID" type="integer"
description="The credentials to be used" mandatory="true" order-index="12" />
</parameters>
```

참고: 이것이 기본 구성입니다. 따라서 **db_adapter.xml** 파일에 이미 이 정의가 포함되어 있습니다.

이 방법으로 데이터베이스에 대한 연결을 구성할 수 없는 경우가 있습니다. 예를 들어, Oracle RAC에 연결하거나 CMDB에 제공된 것과 다른 데이터베이스 드라이버를 사용하여 연결하는 경우입니다.

이러한 경우 사용자 이름(스키마), 비밀번호 및 연결 URL 문자열을 사용하여 연결을 설명할 수 있습니다.

이를 정의하려면 다음과 같이 어댑터의 XM 매개 변수 섹션을 편집합니다.

```
<parameters>
  <!--The description attribute may be written in simple text or HTML.-->
  <!--The host attribute is treated as a special case by CMDBRTSM-->
  <!--and will automatically select the probe name (if possible)-->
  <!--according to this attribute's value.-->
  <!--Display name and description may be overwritten by I18N values-->
    <parameter name="url" display-name="Connection String" type="string" description="The
connection string to connect to the database" mandatory="true" order-index="10" />
    <parameter name="credentialsId" display-name="Credentials ID" type="integer"
description="The credentials to be used" mandatory="true" order-index="12" />
</parameters>
```

다음은 기본 Data Direct 드라이버를 사용하여 Oracle RAC에 연결하는 URL의 예는 **jdbc:mercury:oracle://labm3amdb17:1521;ServiceName=RACQA;AlternateServers=(labm3amdb18:1521);LoadBalancing=true**입니다.

4. 임시 디렉터리에서 **adapterCode** 폴더를 열고 **GenericDBAdapter**의 이름을 이전 단계에서 사용한 **adapter id**의 값으로 바꿉니다.

   이 폴더에는 어댑터의 구성(예: CMDB의 어댑터 이름, 쿼리 및 클래스)과 어댑터가 지원하는 RDBMS의 필드가 포함되어 있습니다.

5. 필요에 따라 어댑터를 구성합니다. 자세한 내용은 "어댑터 구성 -간단한 방법"(124페이지)을 참조하십시오.

6. "다음과 같이 어댑터 XML 파일을 편집합니다."(121페이지) 단계에 설명된 대로 **adapter id** 특성에 지정한 이름과 동일한 이름을 사용하여 *.zip 파일을 만듭니다.

   참고: descriptor.xml 파일은 모든 패키지에 있는 기본 파일입니다.

7. 이전 단계에서 만든 새 패키지를 저장합니다. 어댑터의 기본 디렉터리는 **C:\hp\UCMDB\UCMDBServer\content\adapters**입니다.

어댑터 구성 - 간단한 방법

간단한 방법은 어댑터가 사용하는 **simplifiedConfiguration.xml** 매핑 파일을 만들기 위한 방법입니다. 이 방법을 사용하면 단일 CIT의 기본 채우기 또는 연합을 수행할 수 있습니다.

이 섹션에 제공된 지침에서는 CMDB의 특정 CI 유형에 대한 클래스 모델을 RDBMS에 매핑하는 방법을 설명합니다.

이 섹션에 설명된 모든 구성 파일은 "어댑터 패키지 준비"(121페이지)에서 압축을 푼 **C:\hp\UCMDB\UCMDBServer\content\adapters** 폴더의 **db-adapter.zip** 패키지에 있습니다.

참고: 이 방법을 실행하면 자동으로 생성되는 **orm.xml** 파일은 고급 방법으로 작업할 때 사용할 수 있는 좋은 예입니다.

이 간단한 방법은 다음 작업이 필요한 경우에 사용합니다.

- 노드 특성과 같은 단일 노드 연합/채우기
- 일반 데이터베이스 어댑터 기능 시연

이 메서드가 지원하는 사항은 다음과 같습니다.

- 단일 노드 연합/채우기만 지원
- 다대일 가상 관계만 지원

adapter.conf 파일 구성

어댑터가 간단한 구성 방법을 사용하도록 adapter.conf 파일의 설정을 변경하려면 다음을 수행합니다.

1. 텍스트 편집기에서 **adapter.conf** 파일을 엽니다.
2. **use.simplified.xml.config=<true/false>** 줄을 찾습니다.
3. **use.simplified.xml.config=true**로 변경합니다.

예: 간단한 방법을 사용하여 노드 및 IP 주소 채우기

이 예는 **IP Address**에 Containment 링크로 연관된 **Node**를 UCMDB에 채우는 과정을 보여줍니다. RDBMS에는 컴퓨터 이름, 컴퓨터 노드, 컴퓨터의 IP 주소 데이터가 포함된 **simpleNode**라는 테이블이 있습니다.

simpleNode 테이블의 컨텐츠는 다음과 같습니다.

	host_id	host_name	note	ip_address
	1	Comp1	Test Computer 1	12.33.211.52
	2	Comp2	Test Computer 2	12.33.211.53
	3	Comp3	Test Computer 3	12.33.211.54
	4	Comp4	Test Computer 4	12.33.211.55

채우기는 다음과 같이 세 단계로 수행됩니다.

1. "simplifiedConfiguration.xml 만들기"(125페이지)
2. "TQL 만들기"(126페이지)
3. "통합 포인트 만들기"(127페이지)

simplifiedConfiguration.xml 만들기

다음과 같이 **simplifiedConfiguration.xml**을 만듭니다.

1. 다음과 같이 **cmdb-class** 엔터티를 만듭니다.

   <cmdb-class cmdb-class-name="node" default-table-name="simpleNode">

   CI 유형은 **node**이고 RDBMS 테이블 이름은 **simpleNode**입니다.
2. 다음과 같이 테이블의 primary-key를 설정합니다.

   <primary-key column-name="host_id"/>

   이 기본 키는 **orm.xml** 파일에 있는 엔터티 ID와 동일합니다.
3. 다음과 같이 **reconciliation-by-two-nodes** 규칙을 설정합니다.

   <reconciliation-by-two-nodes connected-node-cmdb-class-name="ip_address" cmdb-link-type="containment">

   이 태그는 **Node**와 **IpAddress** CI 유형 사이의 관계를 정의합니다. 관계 유형은 Containment 링크입니다. 연결된 두 CI 유형에서 조정을 수행합니다. 연결된 노드의 특성 매핑(이 경우 IpAddress)은 **connected-node** 특성에 정의됩니다.
4. 다음과 같이 조정 특성 사이에 **OR** 조건을 추가합니다.

   <or is-ordered="true">

   이 태그는 조정 특성 간에 OR 관계를 정의하므로 첫 번째 조정 특성이 **true**이면 조정 규칙 전체가 **true**로 설정됩니다.
5. 다음 특성을 추가합니다.

   <attribute cmdb-attribute-name="name" column-name="host_name" ignore-case="true"/>

   이 태그는 UCMDB에 있는 **node.name**과 **simpleNode** 테이블에 있는 **host_name** 열 사이의 매핑을 설정합니다.

   data_note 특성에 대해서도 같은 과정을 수행합니다.

   <attribute cmdb-attribute-name="data_note" column-name="note" ignore-case="true"/>

연결된 노드 특성을 추가합니다.

<connected-node-attribute cmdb-attribute-name="name" column-name="ip_address"/>

이 태그는 **ip_address.name**과 **simpleNode** 테이블에 있는 **ip_address** 열 사이의 매핑을 설정합니다.

6. 열린 태그를 순서대로 닫습니다.

</or>

</reconciliation-by-two-nodes>

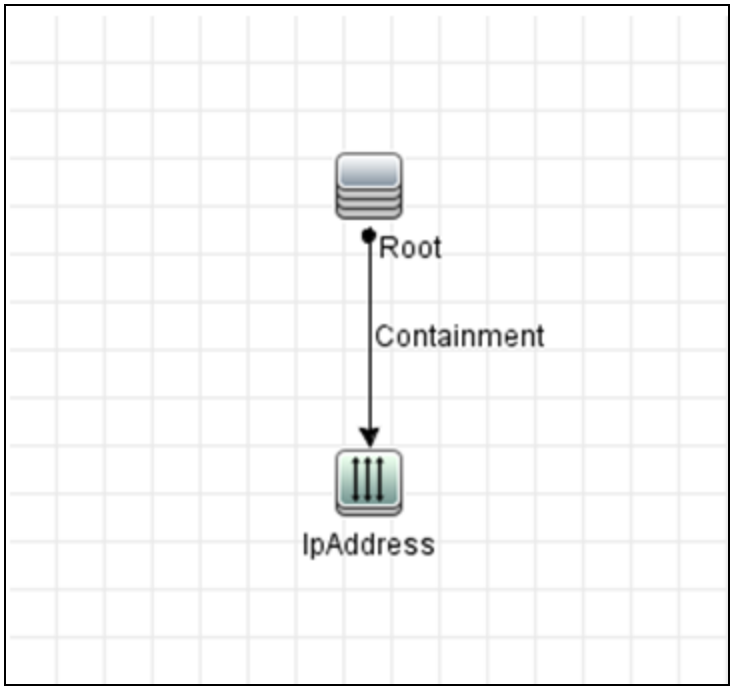
</cmdb-class>

simplifiedConfiguration.xml 파일의 내용은 이제 다음과 같이 표시됩니다.

```
<?xml version="1.0" encoding="UTF-8"?>
<generic-db-adapter-config xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:noNamespaceSchemaLocation="../META-CONF/simplifiedConfiguration.xsd">
  <cmdb-class cmdb-class-name="node" default-table-name="simpleNode">
    <primary-key column-name="host_id"/>
  <reconciliation-by-two-nodes connected-node-cmdb-class-name="ip_address" cmdb-link-
type="containment">
    <or is-ordered="true">
    <attribute cmdb-attribute-name="name" column-name="host_name" ignore-case="true"/>
    <attribute cmdb-attribute-name="data_note" column-name="note" ignore-case="true"/>
      <connected-node-attribute cmdb-attribute-name="name" column-name="ip_address"/>
    </or>
    </reconciliation-by-two-nodes>
  </cmdb-class>
</generic-db-adapter-config>
```

TQL 만들기

TQL은 **ip_address**에 대한 Containment 링크로 연결된 **node**입니다. 노드는 다음과 같이 **root**로 표시되어야 합니다.



TQL을 만들려면 다음을 수행합니다.

1. **모델링 > 모델링 스튜디오**로 이동합니다.
2. **새로 만들기** 버튼을 클릭하고 새 쿼리를 만듭니다.
3. CI 유형 탭으로 이동하여 Node CI 유형과 IpAddress CI 유형을 TQL 화면으로 끕니다.
4. **Node**와 **IpAddress**를 Containment 관계로 연결합니다.
5. **Node** 요소를 마우스 오른쪽 버튼으로 클릭하고 쿼리 노드 속성을 선택합니다.
6. **요소 이름**을 **Root**로 변경합니다.
7. **요소 레이아웃** 탭으로 이동합니다. 특성 조건으로 **특정 특성**을 선택합니다. 사용 가능한 특성 창에서 **Name** 및 **Note**를 선택하여 특정 특성 창으로 이동합니다.
8. **IpAddress** 요소를 마우스 오른쪽 버튼으로 클릭하고 쿼리 노드 속성을 선택합니다.
9. **요소 레이아웃** 탭으로 이동합니다. 특성 조건으로 **특정 특성**을 선택합니다. 사용 가능한 특성 창에서 **Name**을 선택하여 특정 특성 창으로 이동합니다.
10. TQL을 저장합니다.

통합 포인트 만들기

다음과 같이 통합 포인트를 만듭니다.

1. **데이터 흐름 관리 > 통합 스튜디오**로 이동한 후 **새 통합 포인트** 버튼을 클릭합니다.
2. 통합 포인트의 세부 정보를 삽입하고 **확인**을 클릭합니다.
3. 채우기 탭에서 **새 통합 작업** 버튼을 선택하고 이전에 만든 TQL을 추가합니다.
4. 통합 포인트를 저장하고 **전체 동기화 실행** 버튼을 클릭합니다.

어댑터 구성 –고급 방법

이러한 구성 파일은 어댑터 패키지를 준비할 때 압축을 푼 **C:\hp\UCMDB\UCMDBServer\content\adapters** 폴더의 **db-adapter.zip** 패키지에 있습니다. 자세한 내용은 "어댑터 패키지 준비"(121페이지)를 참조하십시오.

이 작업에는 다음 단계가 포함됩니다.

- "orm.xml 파일 구성"(128페이지)
- "reconciliation_rules.txt 파일 구성 "(131페이지)

orm.xml 파일 구성

이 단계에서는 CMDB의 CIT와 관계를 RDBMS의 테이블에 매핑합니다.

1. 텍스트 편집기에서 **orm.xml** 파일을 엽니다.

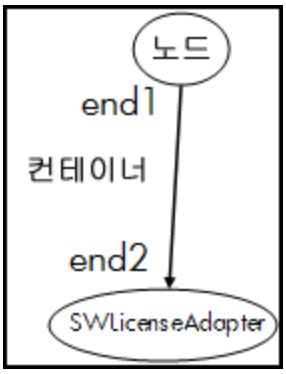
   이 파일에는 기본적으로 필요한 만큼의 여러 CIT와 관계를 매핑하는 데 사용하는 템플릿이 포함되어 있습니다.

   > **참고:** Microsoft Corporation의 메모장에서는 버전에 상관없이 **orm.xml** 파일을 편집하지 마십시오. Notepad++, UltraEdit 또는 기타 타사 텍스트 편집기를 사용합니다.

2. 매핑할 데이터 엔터티에 따라 파일을 변경합니다. 자세한 내용은 다음 예를 참조하십시오.

   다음 유형의 관계는 **orm.xml** 파일에서 매핑할 수 있습니다.

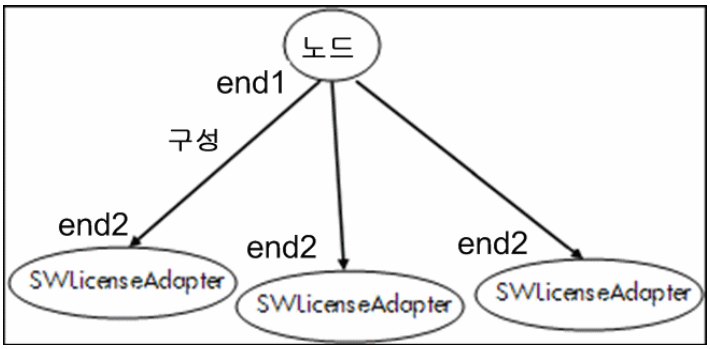
   - 일대일:

   이 관계 유형의 코드는 다음과 같습니다.

```
<one-to-one name="end1" target-entity="node">
      <join-column name="Device_ID" >
</one-to-one>
<one-to-one name="end2" target-entity="sw_sub_component">
      <join-column name="Device_ID" >
      <join-column name="Version_ID" >
</one-to-one>
```

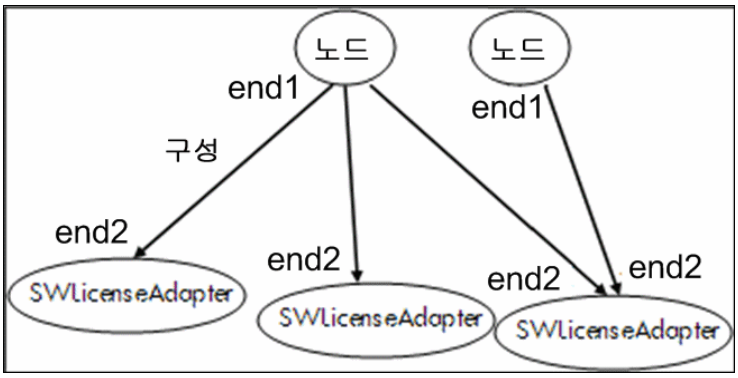
- **다대일:**



이 관계 유형의 코드는 다음과 같습니다.

```
<many-to-one name="end1" target-entity="node">
      <join-column name="Device_ID" >
</many-to-one>
<one-to-one name="end2" target-entity="sw_sub_component">
      <join-column name="Device_ID" >
      <join-column name="Version_ID" >
</one-to-one>
```

- **다대다:**



이 관계 유형의 코드는 다음과 같습니다.

```
<many-to-one name="end1" target-entity="node">
      <join-column name="Device_ID" >
</many-to-one>
<many-to-one name="end2" target-entity="sw_sub_component">
      <join-column name="Device_ID" >
      <join-column name="Version_ID" >
</many-to-one>
```

이름 지정 규칙에 대한 자세한 내용은 "이름 지정 규칙"(150페이지)을 참조하십시오.

데이터 모델과 RDBMS 간 엔터티 매핑의 예:

참고: 구성할 필요가 없는 특성은 다음 예에서 생략되었습니다.

- CMDB CIT의 클래스:

  <entity class="generic_db_adapter.node">

- RDBMS의 테이블 이름:

  <table name="Device"/>

- RDBMS 테이블의 고유한 식별자 열 이름:

  <column name="Device ID"/>

- CMDB CIT의 특성 이름:

  <basic name="name">

- 외부 데이터 원본의 테이블 필드 이름:

  <column name="Device_Name"/>

- "CI 유형 만들기"(119페이지)에서 만든 새 CIT의 이름:

  <entity class="generic_db_adapter.MyAdapter">

- RDBMS의 해당 테이블 이름:

  <table name="SW_License"/>

- RDBMS의 고유 ID:

- CMDB CIT의 특성 이름 및 RDBMS의 해당 특성 이름:

데이터 모델과 RDBMS 간 관계 매핑의 예:

- CMDB 관계의 클래스:

  <entity **class="generic_db_adapter.node_containment_MyAdapter"**>

- 관계가 수행되는 RDBMS 테이블의 이름:

  <table **name="MyAdapter"**/>

- RDBMS의 고유 ID:

```
<id name="id1">
    <column updatable="false" insertable="false"
name="Device_ID">
    <generated-value strategy="TABLE"/>
</id>
<id name="id2">
    <column updatable="false" insertable="false"
name="Version_ID">
    <generated-value strategy="TABLE"/>
</id>
```

- 관계 유형 및 CMDB CIT:

```
<many-to-one target-entity="node" name="end1">
```

- RDBMS의 기본 키 및 외래 키 필드:

```
<join-column updatable="false" insertable="false"
referenced-column-name="[column_name]" name="Device_ID" />
```

reconciliation_rules.txt 파일 구성

이 단계에서는 어댑터에서 CMDB 및 RDBMS를 조정할 때 적용할 규칙을 정의합니다(매핑 엔진을 사용하는 경우에만 버전 8.x와의 호환성을 위해).

1. 텍스트 편집기에서 **META-INF\reconciliation_rules.txt** 파일을 엽니다.
2. 매핑하는 CIT에 따라 파일을 변경합니다. 예를 들어 노드 CIT를 매핑하려면 다음 식을 사용합니다.

   multinode[node] ordered expression[^name]

   참고:
   - 데이터베이스의 데이터가 대/소문자를 구분하는 경우 제어 문자(^)를 삭제하지 마십시오.
   - 여는 각괄호 각각에 해당하는 닫는 괄호가 있는지 확인하십시오.

자세한 내용은 "reconciliation_rules.txt 파일(이전 버전과의 호환용)"(158페이지)을 참조하십시오.

플러그인 구현

이 작업에서는 플러그인을 사용하여 일반 DB 어댑터를 구현하고 배포하는 방법을 설명합니다.

참고: 어댑터용 플러그인을 작성하기 전에 "어댑터 패키지 준비"(121페이지)에 설명된 필요한 단계를 모두 완료해야 합니다.

1. 옵션 1 –Java 기반 플러그인 작성
   a. UCMDB 서버 설치 디렉터리에서 다음 jar 파일을 개발 클래스 경로로 복사합니다.
      - **tools\adapter-dev-kit\db-adapter-framework** 폴더에서 **db-interfaces.jar** 파일 및 **db-interfaces-javadoc.jar** 파일을 복사합니다.
      - **\tools\adapter-dev-kit\SampleAdapters\production-lib** 폴더의 **federation-api.jar** 파일 및 **federation-api-javadoc.jar** 파일을 복사합니다.

        참고: 플러그인을 개발하는 방법에 대한 자세한 내용은 **db-interfaces-javadoc.jar** 및 **federation-api-javadoc.jar** 파일과 다음 위치의 온라인 문서를 참조하십시오.
        - **C:\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\DBAdapterFramework_JavaAPI\index.html**
        - **C:\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\Federation_JavaAPI\index.html**

   b. 플러그인의 Java 인터페이스를 구현하는 Java 클래스를 작성합니다. 인터페이스는 **db-interfaces.jar** 파일에 정의되어 있습니다. 아래 표는 각 플러그인에 대해 구현해야 하는 인터페이스를 나타냅니다.

플러그인 유형	인터페이스 이름	메서드
전체 토폴로지 동기화	FcmdbPluginForSyncGetFullTopology	getFullTopology
변경 내용 동기화	FcmdbPluginForSyncGetChangesTopology	getChangesTopology
레이아웃 동기화	FcmdbPluginForSyncGetLayout	getLayout
지원되는 쿼리 검색	FcmdbPluginForSyncGetSupportedQueries	getSupportedQueries
TQL 쿼리 정의 및 결과 변경	FcmdbPluginGetTopologyCmdbFormat	getTopologyCmdbFormat
CI에 대한 레이아웃 요청 변경	FcmdbPluginGetCIsLayout	getCisLayout
링크에 대한 레이아웃 요청 변경	FcmdbPluginGetRelationsLayout	getRelationsLayout
ID 다시 밀어넣기	FcmdbPluginPushBackIds	getPushBackIdsSQL

플러그인의 클래스에는 공용 기본 생성자가 있어야 합니다. 또한 모든 인터페이스는 initPlugin이라는 메서드를 노출합니다. 이 메서드는 다른 메서드보다 먼저 호출되고 포함하는 어댑터의 환경 개체로 어댑터를 초기화하는 데 사용됩니다.

FcmdbPluginForSyncGetChangesTopology가 구현되면 두 가지 방법으로 변경을 보고합니다.

- **항상 전체 루트 토폴로지를 보고합니다.** 이 토폴로지에 따라 자동 삭제 기능이 제거해야 할 CI를 찾습니다. 이 경우 다음을 사용하여 자동 삭제 기능을 사용 가능하도록 설정합니다.

```
<autoDeleteCITs isEnabled="true">
        <CIT>link</CIT>
        <CIT>object</CIT>
      </autoDeleteCITs>
```

- **제거되었거나 업데이트된 각 CI 인스턴스를 보고합니다.** 이 경우 다음을 사용하여 자동 삭제 메커니즘을 사용되지 않도록 설정합니다.

```
<autoDeleteCITs isEnabled="false">
        <CIT>link</CIT>
        <CIT>object</CIT>
      </autoDeleteCITs>
```

c. Java 코드를 컴파일하기 전에 연합 SDK JAR 및 일반 DB 어댑터 JAR을 클래스 경로에 포함해야 합니다. 연합 SDK는 **C:\hp\UCMDB\UCMDBServer\lib** 디렉터리에 있는 **federation_api.jar** 파일입니다.

d. 배포하기 전에 클래스를 jar 파일로 압축하고 어댑터 패키지의 adapterCode\<어댑터 이름> 폴더 아래에 배치합니다.

2. 옵션 2 –Groovy 기반 플러그인 작성

a. 어댑터 관리 메뉴에서 어댑터 패키지 구성 파일 아래에 Groovy 코드 파일(MyPlugin.groovy)을 만듭니다.

b. Groovy 클래스에서 적절한 인터페이스를 구현합니다. 인터페이스는 db-interfaces.jar 파일에 정의되어 있습니다. 위 표를 참조하십시오.

3. 플러그인은 어댑터의 **\META-INF** 폴더에 있는 **plugins.txt** 파일을 사용하여 구성됩니다.

다음은 DDMi 어댑터의 파일을 보여 주는 예입니다.

```
# mandatory plugin to sync full topology
[getFullTopology]
com.hp.ucmdb.adapters.ed.plugins.replication.EDReplicationPlugin
# mandatory plugin to sync changes in topology
[getChangesTopology]
com.hp.ucmdb.adapters.ed.plugins.replication.EDReplicationPlugin
# mandatory plugin to sync layout
[getLayout]
com.hp.ucmdb.adapters.ed.plugins.replication.EDReplicationPlugin
# plugin to get supported queries in sync. If not defined return all tqls names
[getSupportedQueries]
```

```
# internal not mandatory plugin to change tql definition and tql result
[getTopologyCmdbFormat]
# internal not mandatory plugin to change layout request and CIs result
[getCisLayout]
# internal not mandatory plugin to change layout request and relations result
[getRelationsLayout]
# internal not mandatory plugin to change action on pushBackIds
[pushBackIds]
```

범례:

 # - 주석 줄.

[<어댑터 유형>] -특정 어댑터 유형의 정의가 시작되는 섹션입니다.

각 [<어댑터 유형>] 아래에 빈 줄을 삽입해 연관된 플러그인 클래스가 없음을 나타내거나 플러그인 클래스의 정규화된 이름을 나열할 수 있습니다.

4. **어댑터를 새 jar 파일 및 업데이트된 plugins.xml파일로 압축합니다. 패키지의 나머지 파일은 일반 DB 어댑터에 기반한 어댑터의 파일과 같아야 합니다.**

어댑터 배포

1. UCMDB에서 패키지 관리자에 액세스합니다. 자세한 내용은 *HP Universal CMDB 관리 안내서*에서" 패키지 관리자 페이지"를 참조하십시오.
2. **서버에 패키지 배포(로컬 디스크로부터)** 아이콘  을 클릭하여 어댑터 패키지로 이동합니다. 패키지를 선택하고 **열기**를 클릭한 다음 **배포**를 클릭하여 패키지 관리자의 패키지를 표시합니다.
3. 목록에서 패키지를 선택하고 **패키지 리소스 보기** 아이콘  을 클릭하여 패키지 관리자가 패키지 컨텐츠를 인식하는지 확인합니다.

어댑터 편집

어댑터를 만들고 배포하고 나면 UCMDB 내에서 편집할 수 있습니다. 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 "어댑터 관리"를 참조하십시오.

통합 포인트 만들기

이 단계에서는 연합이 작동하는지 확인합니다. 즉, 연결과 XML 파일이 유효한지 확인합니다. 그러나 이 확인 작업으로는 XML이 RDBMS의 올바른 필드에 매핑되는지는 확인할 수 없습니다.

1. UCMDB에서 통합 스튜디오(**데이터 흐름 관리 > 통합 스튜디오**)에 액세스합니다.
2. 통합 포인트를 만듭니다. 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 새 통합 포인트/통합 포인트 편집 대화 상자를 참조하십시오.

이 통합 포인트를 사용하여 연합할 수 있는 모든 CIT가 연합 탭에 표시됩니다. 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 연합 탭을 참조하십시오.

보기 만들기

이 단계에서는 CIT의 인스턴스를 볼 수 있는 보기를 만듭니다.

1. UCMDB에서 모델링 스튜디오(**모델링 > 모델링 스튜디오**)에 액세스합니다.
2. 보기를 만듭니다. 자세한 내용은 *HP Universal CMDB 모델링 안내서*에서 패턴 보기를 만드는 방법을 참조하십시오.

결과 계산

이 단계에서는 결과를 확인합니다.

1. UCMDB에서 모델링 스튜디오(**모델링 > 모델링 스튜디오**)에 액세스합니다.
2. 보기를 엽니다.
3. **쿼리 결과 개수 계산**  버튼을 클릭하여 결과를 계산합니다.
4. **미리 보기** 버튼을 클릭하여 보기에서 CI를 봅니다.

결과 보기

이 단계에서는 결과를 보고 절차의 문제를 디버깅합니다. 예를 들어 보기에 아무것도 표시되지 않으면 **orm.xml** 파일에서 정의를 확인하고 관계 특성을 제거한 다음 어댑터를 다시 로드합니다.

1. UCMDB에서 IT 유니버스 관리자에 액세스합니다(**모델링 > IT 유니버스 관리자**).
2. CI를 선택합니다. 속성 탭에 연합의 결과가 표시됩니다.

보고서 보기

이 단계에서는 토폴로지 결과를 봅니다. 자세한 내용은 *HP Universal CMDB 모델링 안내서*에서 토폴로지 보고서 개요를 참조하십시오.

로그 파일 사용

계산 흐름, 어댑터 수명 주기를 이해하고 디버그 정보를 보려면 로그 파일을 참조하면 됩니다. 자세한 내용은 "어댑터 로그 파일"(178페이지)을 참조하십시오.

Eclipse를 사용하여 CIT 특성과 데이터베이스 테이블 매핑

주의: 이 절차는 컨텐츠 개발에 관한 고급 지식을 갖춘 사용자를 위해 작성되었습니다. 궁금한 사항은 HP 소프트웨어 지원에 문의하십시오.

이 작업에서는 다음 기능을 위해 Eclipse J2EE 에디션과 함께 제공되는 JPA 플러그인을 설치하고 사용하는 방법을 설명합니다.

- CMDB 클래스 특성과 데이터베이스 테이블 열 간에 그래픽 매핑을 사용할 수 있습니다.
- 정확성을 확인하는 동시에 매핑 파일(**orm.xml**)을 수동으로 편집할 수 있습니다. 정확성 확인에는 구문 확인뿐 아니라 클래스 특성 및 매핑된 데이터베이스 테이블 열이 올바르게 표시되었는지 확인하는 기능이 포함됩니다.
- 상세한 정확성 확인 기능으로, 매핑 파일을 CMDB 서버에 배포하고 오류를 볼 수 있습니다.
- CMDB 서버에서 샘플 쿼리를 정의하고 Eclipse에서 직접 실행하여 매핑 파일을 테스트합니다.

플러그인의 버전 1.1은 UCMDB 버전 9.01 이상과 Eclipse IDE for Java EE Developers 버전 1.2.2.20100217-2310 이상과 호환됩니다.

이 작업에는 다음 단계가 포함됩니다.

- "CMDB에 ORM 파일 배포"(141페이지)
- "샘플 TQL 쿼리 실행"(141페이지)

1. 선행 조건

   다음 사이트에서 Eclipse를 실행할 컴퓨터에 **Java Runtime Environment(JRE) 6**의 최신 업데이트를 설치합니다.
   해당 제품은 **http://java.sun.com/javase/downloads/index.jsp** 사이트에서 다운로드할 수 있습니다.

2. 설치

   a. **Eclipse IDE for Java EE Developers**를 **http://www.eclipse.org/downloads**에서 로컬 폴더(예: **C:\Program Files\eclipse**)로 다운로드하고 압축을 풉니다.

   b. **com.hp.plugin.import_cmdb_model_1.0.jar**을 **C:\hp\UCMDB\UCMDBServer\tools\db-adapter-eclipse-plugin\bin**에서 **C:\Program Files\Eclipse\plugins**로 복사합니다.

   c. **C:\Program Files\Eclipse\eclipse.exe**를 시작합니다. Java 가상 컴퓨터가 없다는 메시지가 표시되면 다음 명령줄을 사용하여 **eclipse.exe**를 시작합니다.

   "C:\Program Files\eclipse\eclipse.exe" -vm "<JRE 설치 폴더>\bin"

3. 작업 환경 준비

   이 단계에서는 작업 영역, 데이터베이스, 연결 및 드라이버 속성을 설정합니다.

   a. **workspaces_gdb.zip** 파일을 **C:\hp\UCMDB\UCMDBServer\tools\db-adapter-eclipse-plugin\workspace**에서 **C:\Documents and Settings\All Users**로 압축을 풉니다.

   > **참고:** 정확한 폴더 경로를 사용해야 합니다. 파일의 압축을 잘못된 경로에 풀거나, 압축을 풀지 않은 상태로 두는 경우 절차가 제대로 수행되지 않습니다.

   b. Eclipse에서 다음과 같이 **File > Switch Workspace > Other**를 선택합니다.

   사용하는 제품에 따라 다음과 같이 선택합니다.

   - SQL Server의 경우 **C:\Documents and Settings\All Users\workspace_gdb_sqlserver** 폴더를 선택합니다.
   - MySQL의 경우 **C:\Documents and Settings\All Users\workspace_gdb_mysql** 폴더를 선택합니다.
   - Oracle의 경우 **C:\Documents and Settings\All Users\workspace_gdb_oracle** 폴더를 선택합니다.

   c. **OK**를 클릭합니다.

   d. Eclipse에서 프로젝트 탐색기 보기를 표시하고 **<활성 프로젝트> > JPA 컨텐츠 > persistence.xml > <활성 프로젝트 이름> > orm.xml**을 선택합니다.

   e. Data Source Explorer 보기(왼쪽 아래 창)에서 데이터베이스 연결을 마우스 오른쪽 버튼으로 클

릭하고 **Properties** 메뉴를 선택합니다.

f. **Properties for <연결 이름>** 대화 상자에서 **Common**을 선택하고 **Connect every time the workbench is started** 확인란을 선택합니다. **Driver Properties**를 선택하고 연결 속성을 입력합니다. **Test Connection**을 클릭하고 연결이 잘 되었는지 확인합니다. **OK**를 클릭합니다.

g. Data Source Explorer 보기에서 데이터베이스 연결을 마우스 오른쪽 버튼으로 클릭하고 **Connect**를 클릭합니다. 데이터베이스 연결 아이콘 아래에 데이터베이스 스키마 및 테이블이 포함된 트리가 표시됩니다.

4. 어댑터 만들기

"1단계: 어댑터 만들기"(27페이지)에 설명된 지침에 따라 어댑터를 만듭니다.

5. CMDB 플러그인 구성

a. Eclipse에서 **UCMDB > Settings**를 클릭하여 **CMDB Settings** 대화 상자를 엽니다.

b. 활성 프로젝트로 새로 만든 JPA 프로젝트를 아직 선택하지 않은 경우 선택합니다.

c. CMDB 호스트 이름(예 **localhost** 또는 **labm1.itdep1**)을 입력합니다. 포트 번호나 http:// 접두사는 주소에 포함할 필요가 없습니다.

d. CMDB API에 액세스할 수 있는 사용자 이름과 비밀번호(보통 **admin/admin**)를 입력합니다.

e. CMDB 서버의 **C:\hp** 폴더가 네트워크 드라이브로 매핑되었는지 확인합니다.

f. **C:\hp** 아래에서 관련 어댑터의 기본 폴더를 선택합니다. 기본 폴더는 **dbAdapter.jar** 파일 및 **META-INF** 하위 폴더가 포함된 폴더입니다. 해당 경로는 **C:\hp\UCMDB\UCMDBServer\runtime\fcmdb\CodeBase\<adapter name>**이어야 합니다. 끝에 백슬래시(****)가 없어야 합니다.

6. UCMDB 클래스 모델 가져오기

이 단계에서는 JPA 엔터티로 매핑할 CIT를 선택합니다.

a. **UCMDB > CMDB 클래스 모델 가져오기**를 클릭하여 **CI 유형 선택** 대화 상자를 엽니다.

b. JPA 엔터티로 매핑할 CI 유형을 선택합니다. **확인**을 클릭합니다. CI 유형은 Java 클래스로 가져옵니다. 가져온 내용이 활성 프로젝트의 **src** 폴더 아래에 나타나는지 확인합니다.

7. ORM 파일 생성 -UCMDB 클래스를 데이터베이스 테이블에 매핑

이 단계에서는 이전 단계에서 가져온 Java 클래스를 데이터베이스 테이블에 매핑합니다.

a. DB 연결이 연결되었는지 확인합니다. 프로젝트 탐색기에서 활성 프로젝트(기본값: myProject)를 마우스 오른쪽 버튼으로 클릭합니다. JPA 보기를 선택하고 **연결에서 기본 스키마 다시 정의** 확인란을 선택한 다음 관련 데이터베이스 스키마를 선택합니다. **확인**을 클릭합니다.

b. CIT를 매핑합니다. JPA 구조 보기에서 **엔터티 매핑** 분기를 마우스 오른쪽 버튼으로 클릭하고 **클래스 추가**를 선택합니다. **영구 클래스 추가** 대화 상자가 열립니다. **매핑 형식** 필드(**엔터티**)는 변경하지 마십시오.

c. **찾아보기**를 클릭하고 매핑할 UCMDB 클래스를 선택합니다(모든 UCMDB 클래스는 **generic_db_adapter** 패키지에 속함).

d. 두 대화 상자에서 모두 **확인**을 클릭합니다. 선택한 클래스가 JPA 구조 보기의 **엔터티 매핑** 분기에 표시됩니다.

> **참고:** 엔터티가 특성 트리 없이 표시되면 프로젝트 탐색기 보기에서 활성 프로젝트를 마우스 오른쪽 버튼으로 클릭합니다. **닫기**, **열기**를 차례로 선택합니다.

e. JPA 세부 정보 보기에서 UCMDB 클래스를 매핑할 기본 데이터베이스 테이블을 선택합니다. 다른 모든 필드는 변경하지 마십시오.

8. ID 매핑

JPA 표준에 따라 각 영구 클래스에 하나 이상의 ID 특성이 있어야 합니다. UCMDB 클래스의 경우 최대 3개의 특성을 ID로 매핑할 수 있습니다. 잠재적 ID 특성을 **id1**, **id2**, **id3**이라고 지칭합니다.

ID 특성을 매핑하려면 다음을 수행합니다.

a. JPA 구조 보기의 **엔터티 매핑** 분기 아래에서 해당 클래스를 확장하고 관련 특성(예: **id1**)을 마우스 오른쪽 버튼으로 클릭한 다음 **XML 및 맵에 특성 추가...**를 선택합니다.

b. **영구 특성 추가** 대화 상자가 열립니다. **매핑 형식** 필드에서 **Id**를 선택하고 **확인**을 클릭합니다.

c. JPA 세부 정보 보기에서 ID 필드를 매핑할 데이터베이스 테이블 열을 선택합니다.

9. 특성 매핑

이 단계에서는 특성을 데이터베이스 열에 매핑합니다.

a. JPA 구조 보기의 **엔터티 매핑** 분기 아래에서 해당 클래스를 확장하고 관련 특성(예: **host_hostname**)을 마우스 오른쪽 버튼으로 클릭한 다음 **XML 및 맵에 특성 추가...**를 선택합니다.

b. **영구 특성 추가** 대화 상자가 열립니다. **매핑 형식** 필드에서 **기본**을 선택하고 **확인**을 클릭합니다.

c. JPA 세부 정보 보기에서 특성 필드를 매핑할 데이터베이스 테이블 열을 선택합니다.

10. 유효한 링크 매핑

"ORM 파일 생성 –UCMDB 클래스를 데이터베이스 테이블에 매핑"(138페이지) 단계에 설명된 UCMDB 클래스 매핑 단계를 수행하여 유효한 링크를 표시합니다. 그러한 각 클래스의 이름은 **<end1 엔터티 이름>_<링크 이름>_<end 2 엔터티 이름>** 구조로 되어 있습니다. 예를 들어 호스트와 위치 사이의 **포함** 링크는 **generic_db_adapter.host_contains_location**이라는 Java 클래스로 표시됩니다. 자세한 내용은 "reconciliation_rules.txt 파일(이전 버전과의 호환용)"(158페이지)을 참조하십시오.

a. 링크 클래스의 ID 특성을 "ID 매핑"(139페이지)에 설명된 대로 매핑합니다. 각 ID 특성에 대해, JPA 세부 정보 보기에서 **세부 정보** 확인란 그룹을 확장하고 **삽입 가능** 및 **업데이트 가능** 확인란의 선택을 취소합니다.

b. 링크 클래스의 **end1** 및 **end2** 특성을 다음과 같이 매핑합니다. 링크 클래스의 각 **end1** 및 **end2** 특성의 경우:

- JPA 구조 보기의 **엔터티 매핑** 분기 아래에서 해당 클래스를 확장하고 관련 특성(예: **end1**)을 마우스 오른쪽 버튼으로 클릭한 다음 **XML 및 맵에 특성 추가...**를 선택합니다.

- **영구 특성 추가** 대화 상자의 **매핑 형식** 필드에서 **다대일** 또는 **일대일**을 선택합니다.
- 지정된 **end1** 또는 **end2** CI에 이 유형의 링크를 여러 개 포함할 수 있는 경우 **다대일**을 선택합니다. 그렇지 않은 경우 **일대일**을 선택합니다. 예를 들어 **host_contains_ip** 링크의 경우 호스트 하나는 여러 개의 IP를 포함할 수 있어야 하므로 **host** 끝은 **다대일**로 매핑하고, IP 하나는 호스트를 하나만 포함할 수 있으므로 **ip** 끝은 **일대일**로 매핑해야 합니다.
- JPA 세부 정보 보기에서 **대상 엔터티**(예: **generic_db_adapter.host**)를 선택합니다.
- JPA 세부 정보 보기의 **조인 열** 섹션에서 **기본값 다시 정의**를 선택합니다. **편집**을 클릭합니다. **조인 열 편집** 대화 상자에서 **end1/end2** 대상 엔터티 테이블의 항목을 가리키는 링크 데이터베이스 테이블의 외래 키 열을 선택합니다. **end1/end2** 대상 엔터티 테이블에서 참조되는 열 이름이 해당 ID 특성에 매핑되어 있는 경우 **참조되는 열 이름**을 변경하지 않고 그대로 두십시오. 그렇지 않으면 외래 키 열이 가리킬 열 이름을 선택합니다. **삽입 가능** 및 **업데이트 가능** 확인란의 선택을 취소하고 **확인**을 클릭합니다.
- **end1/end2** 대상 엔터티에 둘 이상의 ID가 있으면 **추가** 버튼을 클릭하여 조인 열을 더 추가하고 이전 단계에 설명된 방법과 동일하게 이러한 열을 매핑합니다.

11. ORM 파일 생성 -보조 테이블 사용

JPA를 사용하면 Java 클래스 하나를 둘 이상의 데이터베이스 테이블에 매핑할 수 있습니다. 예를 들어 **Host**를 **Device** 테이블에 매핑하여 해당 특성 대부분이 지속되도록 설정하고, **NetworkNames** 테이블에 매핑하여 **host_hostName**이 지속되도록 설정할 수 있습니다. 이 경우 **Device**는 기본 테이블이고 **NetworkNames**는 보조 테이블입니다. 보조 테이블은 원하는 수만큼 정의할 수 있습니다. 유일한 조건은 기본 테이블과 보조 테이블의 항목 간에 일대일 관계가 있어야 한다는 점입니다.

12. 보조 테이블 정의

JPA 구조 보기에서 적절한 클래스를 선택합니다. **JPA 세부 정보** 보기에서 **보조 테이블** 섹션에 액세스한 다음 **추가**를 클릭합니다. **보조 테이블 추가** 대화 상자에서 적절한 보조 테이블을 선택합니다. 다른 필드는 변경하지 마십시오.

기본 테이블과 보조 테이블에 동일한 기본 키가 없으면 **JPA 세부 정보** 보기의 **기본 키 조인 열** 섹션에서 조인 열을 구성합니다.

13. 보조 테이블에 특성 매핑

클래스 특성을 다음과 같이 보조 테이블의 필드에 매핑합니다.

a. "특성 매핑"(139페이지)에 설명된 대로 특성을 매핑합니다.

b. JPA 세부 정보 보기의 **열** 섹션에 있는 **테이블** 필드에서 보조 테이블 이름을 선택하여 기본값을 바꿉니다.

14. 기존 ORM 파일을 기본으로 사용

기존 **orm.xml** 파일을 개발 중인 프로젝트의 기본으로 사용하려면 다음 단계를 수행합니다.

a. 기존 **orm.xml** 파일에 매핑된 모든 CIT를 활성 Eclipse 프로젝트로 가져왔는지 확인합니다.

b. 기존 파일에서 엔터티 매핑의 일부 또는 전부를 선택하고 복사합니다.

c. Eclipse JPA 관점에서 **orm.xml** 파일의 **원본** 탭을 선택합니다.

d. 복사한 모든 엔터티 매핑을 편집한 **orm.xml** 파일의 **<entity-mappings>** 태그 아래에 있는 **<schema>** 태그 밑에 붙여 넣습니다. schema 태그를 "ORM 파일 생성 -UCMDB 클래스를 데이터베이스 테이블에 매핑"(138페이지) 단계에 설명된 대로 구성했는지 확인하십시오. 붙여 넣은 모든 엔터티가 이제 JPA 구조 보기에 표시됩니다. 이제부터는 **orm.xml** 파일의 xml 코드를 통해 그래픽 방식으로 그리고 수동으로 모두 매핑을 편집할 수 있습니다.

e. **저장**을 클릭합니다.

15. 어댑터에서 기존 ORM 파일 가져오기

어댑터가 이미 있는 경우 Eclipse 플러그인을 사용하여 어댑터의 ORM 파일을 그래픽 방식으로 편집할 수 있습니다. orm.xml 파일을 Eclipse로 가져와 플러그인을 사용하여 편집한 다음 다시 UCMDB 컴퓨터에 배포합니다. ORM 파일을 가져오려면 Eclipse 도구 모음에서 버튼을 누릅니다. 확인 대화 상자가 표시됩니다. 확인을 클릭합니다. ORM 파일이 UCMDB 컴퓨터에서 활성 Eclipse 프로젝트로 복사되고 UCMDB 클래스 모델에서 모든 관련 클래스를 가져옵니다.

JPA 구조 보기에 관련 클래스가 표시되지 않으면 프로젝트 탐색기 보기에서 활성 프로젝트를 마우스 오른쪽 버튼으로 클릭하고 닫기를 선택한 다음 열기를 선택합니다.

지금부터 Eclipse를 사용하여 그래픽 방식으로 ORM 파일을 편집한 다음 "CMDB에 ORM 파일 배포"(141페이지)에 설명된 대로 다시 UCMDB 컴퓨터에 배포할 수 있습니다.

16. orm.xml 파일의 정확성 확인 -내장된 정확성 확인 기능

Eclipse JPA 플러그인은 orm.xml 파일에 오류가 있는지 확인하고 오류를 표시합니다. 구문 오류(예: 잘못된 태그 이름, 닫히지 않은 태그, 누락된 ID)와 매핑 오류(예: 잘못된 특성 이름 또는 데이터베이스 테이블 필드 이름)를 모두 확인합니다. 오류가 있는 경우 문제 보기에 오류에 대한 설명이 표시됩니다.

17. 새 통합 포인트 만들기

이 어댑터의 CMDB에 통합 포인트가 없는 경우 통합 스튜디오에서 만들 수 있습니다. 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 통합 스튜디오를 참조하십시오.

열리는 대화 상자에서 통합 포인트 이름을 입력합니다. orm.xml 파일이 어댑터 폴더로 복사됩니다. 가져온 모든 CI 유형을 지원되는 클래스로 사용하여 통합 포인트가 만들어집니다. 단, reconciliation_rules.txt 파일에서 multinode CIT를 구성한 경우 이 CIT는 제외됩니다. 자세한 내용은 "reconciliation_rules.txt 파일(이전 버전과의 호환용)"(158페이지)을 참조하십시오.

18. CMDB에 ORM 파일 배포

orm.xml 파일을 저장하고 UCMDB > ORM 배포를 클릭하여 이 파일을 UCMDB 서버에 배포합니다. orm.xml 파일이 어댑터 폴더로 복사되고 어댑터가 다시 로드됩니다. 작업 결과가 작업 결과 대화 상자에 표시됩니다. 다시 로드 프로세스 중에 오류가 발생하면 Java 예외 스택 추적이 대화 상자에 표시됩니다. 아직 어댑터를 사용하여 통합 포인트를 정의하지 않은 경우 배포할 때 매핑 오류가 감지되지 않습니다.

19. 샘플 TQL 쿼리 실행

a. **모델링 스튜디오에서 쿼리(보기가 아님)를 정의합니다. 자세한 내용은 *HP Universal CMDB 모델***

*링 안내서*에서 모델링 스튜디오를 참조하십시오.

b. "새 통합 포인트 만들기"(141페이지) 단계에서 만든 어댑터를 사용하여 통합 포인트를 만듭니다. 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 새 통합 포인트/통합 포인트 편집 대화 상자를 참조하십시오.

c. 어댑터를 만드는 동안 쿼리에 사용해야 하는 CI 유형이 이 통합 포인트에서 지원되는지 확인합니다.

d. CMDB 플러그인을 구성할 때 설정 대화 상자에서 이 샘플 쿼리 이름을 사용하십시오. 자세한 내용은 위의 "CMDB 플러그인 구성"(138페이지) 단계를 참조하십시오.

e. **TWL 실행** 버튼을 클릭하여 샘플 TQL을 실행하고 새로 만든 **orm.xml** 파일을 사용하여 필요한 결과를 반환하는지 확인합니다.

어댑터 구성 파일

이 섹션에서 논의할 파일은 **C:\hp\UCMDB\UCMDBServer\content\adapters** 폴더의 **db-adapter.zip** 패키지에 있습니다.

이 섹션에서는 다음 구성 파일에 대해 설명합니다.

- "adapter.conf 파일"(143페이지)
- "simplifiedConfiguration.xml 파일"(145페이지)
- "orm.xml 파일"(146페이지)
- "reconciliation_types.txt 파일"(158페이지)
- "reconciliation_rules.txt 파일(이전 버전과의 호환용)"(158페이지)
- "transformations.txt 파일"(160페이지)
- "discriminator.properties 파일"(161페이지)
- "replication_config.txt 파일"(162페이지)
- "fixed_values.txt 파일"(162페이지)
- "Persistence.xml 파일"(162페이지)

일반 구성

- **adapter.conf.** 어댑터 구성 파일입니다. 자세한 내용은 "adapter.conf 파일"(143페이지)을 참조하십시오.

간단한 구성

- **simplifiedConfiguration.xml**. **orm.xml**, **transformations.txt**, **reconciliation_rules.txt**를 기능이 더 적은 파일로 바꾸는 구성 파일입니다. 자세한 내용은 "simplifiedConfiguration.xml 파일"(145페이지)을 참조하십시오.

고급 구성

- **orm.xml**. CMDB CIT와 데이터베이스 테이블 간을 매핑한 개체-관계 매핑 파일입니다. 자세한 내용은 "orm.xml 파일"(146페이지)을 참조하십시오.
- **reconciliation_rules.txt**. 조정 규칙이 포함되어 있습니다. 자세한 내용은 "reconciliation_rules.txt 파일(이전 버전과의 호환용)"(158페이지)을 참조하십시오.
- **transformations.txt.** CMDB 값에서 데이터베이스 값으로 변환하거나 그 반대로 변환하는 데 적용할 변환기를 지정하는 변환 파일입니다. 자세한 내용은 "transformations.txt 파일"(160페이지)을 참조하십시오.
- **Discriminator.properties.** 이 파일은 지원되는 각 CI 유형을 가능한 해당 값 목록(쉼표로 구분된 목록)에 매핑합니다. 자세한 내용은 "discriminator.properties 파일"(161페이지)을 참조하십시오.
- **Replication_config.txt.** 이 파일은 쉼표로 구분된 CI 목록과, 해당 속성 조건이 복제 플러그인을 통해 지원되는 관계 유형을 포함합니다. 자세한 내용은 "replication_config.txt 파일"(162페이지)을 참조하십시오.
- **Fixed_values.txt.**이 파일을 사용하면 특정 CIT의 특정 특성에 대해 고정값을 구성할 수 있습니다. 자세한 내용은 "fixed_values.txt 파일"(162페이지)을 참조하십시오.

Hibernate 구성

- **persistence.xml.** 기본 Hibernate 구성을 다시 정의하는 데 사용됩니다. 자세한 내용은 "Persistence.xml 파일"(162페이지)을 참조하십시오.

어댑터에 대한 임시 테이블 지원 사용

임시 테이블을 사용하도록 설정하면 어댑터가 원격 데이터베이스와 더 효율적으로 작동하여 데이터베이스와 네트워크에 대한 로드를 줄이고 성능을 향상할 수 있습니다.

일반 데이터베이스 어댑터에서 임시 테이블 지원을 사용하도록 설정하려면 다음 조건을 충족해야 합니다.

- **데이터베이스에 연결하기 위해 제공된 자격 증명은 임시 테이블을 만들고, 수정하고, 삭제할 수 있는 권한을 포함해야 합니다.**
- adapter.conf 구성 파일에서 다음 설정을 구성하십시오.

  temp.tables.enabled=true

  performance.enable.single.sql=true

참고: **임시 테이블은 Microsoft SQL 및 Oracle용으로만 지원됩니다.**

adapter.conf 파일

이 파일에는 다음 설정이 포함됩니다.

- **use.simplified.xml.config=false.true**: simplifiedConfiguration.xml을 사용합니다.

  참고: 이 파일은 orm.xml, transformations.txt, reconciliation_rules.txt를 기능이 더 적은 파일로 바꾸는 데 사용합니다.

- **dal.ids.chunk.size=300.** 이 값을 변경하지 마십시오.
- **dal.use.persistence.xml=false**. **true**: 어댑터가 persistence.xml에서 Hibernate 구성을 읽습니다.

  참고: Hibernate 구성을 다시 정의하지 않는 것이 좋습니다.

- **performance.memory.id.filtering=true**. GDBA가 TQLS를 실행할 때, 많은 수의 ID가 검색되어 SQL을 사용하여 데이터베이스로 다시 전송해야 하는 경우가 있습니다. 이러한 과도한 작업을 방지하고 성능을 향상시키기 위해 GDBA는 전체 보기/테이블을 읽고 메모리 내의 결과를 필터링합니다.
- **id.reconciliation.cmdb.id.type=string/bytes**. ID 조정을 사용하여 일반 DB 어댑터를 매핑할 때 **META-INF/ adapter.conf** 속성을 변경하여 **cmdb_id**를 **string** 또는 **bytes/raw** 열 유형에 매핑할 수 있습니다.
- **performance.enable.single.sql=true**. 이 매개 변수는 선택 사항입니다. 이 매개 변수가 파일에 나타나지 않으면 기본값이 **true**입니다. **true**로 설정된 경우, 일반 데이터베이스 어댑터는 실행 중인 각 쿼리(채우기 또는 연합 쿼리 중 하나)에 대해 단일 SQL 문을 생성합니다. 단일 SQL 문을 사용하면 일반 데이터베이스 어댑터의 성능과 메모리 사용률이 향상됩니다. **false**로 설정된 경우, 일반 데이터베이스 어댑터는 다중 SQL 문을 생성하며 단일 SQL 문을 생성할 때보다 소요되는 시간과 메모리가 많을 수 있습니다. 다음과 같은 시나리오에서는 이 특성이 **true**로 설정되어도 어댑터가 단일 SQL 문을 생성하지 않습니다.
  - 어댑터가 연결되어 있는 데이터베이스가 Oracle 또는 SQL Server가 아닐 경우
  - 실행되는 TQL에 0..* 및 1..* 이외의 카디널리티 조건이 포함된 경우(예: 2..* 또는 0..2 카디널리티 조건이 있는 경우)
- **in.expression.size.limit=950**(기본값). 이 매개 변수는 인수 목록의 크기 제한에 도달하면 실행된 SQL의 'IN' 식을 분할합니다.
- **stringlist.delimiter.of.<CIT 이름>.<특성 이름>=<구분 기호>**. 문자열 목록 특성을 일반 데이터베이스 어댑터의 데이터베이스 열에 매핑하려면 연결된 값 목록이 포함된 문자열 열에 특성을 매핑해야 합니다. 예를 들어 CI 유형이 **policy**인 **policy_category** 특성을 매핑하려는 경우 문자열 열에 value1, value2, value3 등 세 가지 값의 목록을 정의하는 value1##value2##value3 값 목록이 포함되어 있으면 **stringlist.delimiter.of.policy.policy_category=##** 설정을 사용합니다.
- **temp.tables.enabled=true**. 성능 향상을 위해 임시 테이블을 사용하도록 설정합니다. **performance.enable.single.sql**이 사용되도록 설정된 경우에만 사용할 수 있습니다(Microsoft SQL 및 Oracle에서만 지원됨). 데이터베이스 서버에서 일부 권한이 필요할 수 있습니다.
- **temp.tables.min.value=50**. 임시 테이블을 사용하는 데 필요한 조건 값(또는 ID)의 수를 정의합니다.

simplifiedConfiguration.xml 파일

이 파일은 UCMDB 클래스를 데이터베이스 테이블에 간단히 매핑하는 데 사용됩니다. 파일 편집에 필요한 템플릿에 액세스하려면 **어댑터 관리 > db-adapter > 구성 파일**로 이동합니다.

이 섹션에는 다음 항목이 포함됩니다.

- "simplifiedConfiguration.xml 파일 템플릿"(145페이지)
- "제한"(146페이지)

simplifiedConfiguration.xml 파일 템플릿

- **CMDB-class-name** 속성은 다음과 같은 multinode 유형(TQL에서 연합 CIT를 연결할 노드)입니다.

```
<?xml version="1.0" encoding="UTF-8"?>
<generic-DB-adapter-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../META-CONF/simplifiedConfiguration.xsd">
  <CMDB-class CMDB-class-name="node" default-table-name="[table_name]">
    <primary-key column-name="[column_name]"/>
```

- **reconciliation-by-two-nodes**. 하나 또는 두 개의 노드를 사용하여 조정을 수행할 수 있습니다. 이 예제에서는 두 개의 노드를 사용하여 조정합니다.
- **connected-node-CMDB-class-name.** 조정 TQL에 필요한 두 번째 클래스 유형입니다.
- **CMDB-link-type.** 조정 TQL에 필요한 관계 유형입니다.
- **link-direction.** 조정 TQL의 관계 방향입니다(node에서 ip_address로의 방향 또는 ip_address에서 node로의 방향).

  ```
  <reconciliation-by-two-nodes connected-node-CMDB-class-name="ip_address" CMDB-link-type="containment" link-direction="main-to-connected">
  ```

  조정 식은 OR 식의 형태이고 각 OR 식에는 AND 식이 포함됩니다.
- **is-ordered.** 조정을 순서 형태로 수행할지 아니면 일반 OR 비교를 통해 수행할지 결정합니다.

  ```
  <or is-ordered="true">
  ```

  조정 속성이 기본 클래스(multinode)에서 검색되는 경우 **attribute** 태그를 사용하고, 그렇지 않으면 **connected-node-attribute** 태그를 사용합니다.
- **ignore-case.true**: UCMDB 클래스 모델의 데이터를 RDBMS의 데이터와 비교할 때 대/소문자는 구분하지 않습니다.

  ```
  <attribute CMDB-attribute-name="name" column-name="[column_name]" ignore-case="true"/>
  ```

  열 이름은 외래 키 열(multinode 기본 키 열을 가리키는 값이 있는 열)의 이름입니다.

  multinode 기본 키 열이 여러 개의 열로 구성된 경우 외래 키 열도 각 기본 키 열에 하나씩 여러 개가 필요합니다.

  ```
  <foreign-primary-key column-name="[column_name]" CMDB-class-primary-key-column="[column_name]"/>
  ```

기본 키 열이 거의 없으면 이 열을 복제합니다.

<primary-key column-name="[column_name]"/>

- **from-CMDB-converter** 및 **to-CMDB-converter** 속성은 다음 인터페이스를 구현하는 Java 클래스입니다.
  - com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.FcmdbDalTransformerFromExternalDB
  - com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.FcmdbDalTransformerToExternalDB

  CMDB 및 데이터베이스의 값이 서로 다른 경우 이러한 변환기를 사용합니다.

  이 예제에서는 괄호 안에 기록된 XML 파일(**generic-enum-transformer-example.xml**)에 따라 열거자를 변환하는 데 GenericEnumTransformer를 사용합니다.

  <attribute CMDB-attribute-name="[CMDB_attribute_name]" column-name="[column_name]" from-CMDB-converter="com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl. GenericEnumTransformer (generic-enum-transformer-example.xml)" to-CMDB-onverter="com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl. GenericEnumTransformer(generic-enum-transformer-example.xml)" />

  <attribute CMDB-attribute-name="[CMDB_attribute_name]" column-name="[column_name]" />

  <attribute CMDB-attribute-name="[CMDB_attribute_name]" column-name="[column_name]" />

  </class>

  </generic-DB-adapter-config>

제한

- 데이터베이스 원본의 단일 노드를 포함하는 TQL 쿼리만 매핑하는 데 사용할 수 있습니다. 예를 들어 node > ticket 및 ticket TQL 쿼리를 실행할 수 있습니다. 데이터베이스에서 노드의 계층 구조를 가져오려면 고급 **orm.xml** 파일을 사용해야 합니다.
- 일대다 관계만 지원됩니다. 예를 들어 각 노드에서 하나 이상의 티켓을 가져올 수 있습니다. 둘 이상의 노드에 속한 티켓은 가져올 수 없습니다.
- 동일한 클래스를 서로 다른 유형의 CMDB CIT에 연결할 수 없습니다. 예를 들어 ticket을 node에 연결하도록 정의하면 application에도 연결할 수 없습니다.

orm.xml 파일

이 파일은 CMDB CIT를 데이터베이스 테이블에 매핑하는 데 사용됩니다.

새 파일을 만드는 데 사용할 템플릿은 **C:\hp\UCMDB\UCMDBServer\runtime\fcmdb\CodeBase\GenericDBAdapter\META-INF** 디렉터리에 있습니다.

배포된 어댑터의 XML 파일을 편집하려면 **어댑터 관리 > db-adapter > 구성 파일**로 이동합니다.

이 섹션에는 다음 항목이 포함됩니다.

orm.xml 파일 템플릿

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0" xsi:schemaLocation=
"http://java.sun.com/xml/ns/persistence/orm http://java.sun.com/xml/ns/persistence/orm_1_
0.xsd">
  <description>Generic DB adapter orm</description>
```

패키지 이름은 변경하지 마십시오.

```
  <package>generic_db_adapter</package>
```

entity. CMDB CIT의 이름이며, multinode 엔터티입니다.

class에는 **generic_db_adapter** 접두사를 포함해야 합니다.

```
  <entity class="generic_db_adapter.node">
    <table name="[table_name]" />
```

엔터티가 두 개 이상의 테이블에 매핑된 경우 보조 테이블을 사용합니다.

```
    <secondary-table name="" />
    <attributes>
```

판별자가 있는 단일 테이블 상속의 경우 다음 코드를 사용합니다.

```
    <inheritance strategy="SINGLE_TABLE" />
    <discriminator-value>node</discriminator-value>
    <discriminator-column name="[column_name]" />
```

id 태그가 있는 특성은 기본 키 열입니다. 이러한 기본 키 열의 이름 지정 규칙이 **idX**(id1, id2 등)인지 확인합니다. 여기서 **X**는 기본 키의 열 인덱스입니다.

```
    <id name="id1">
```

기본 키의 열 이름만 변경합니다.

```
      <column updatable="false" insertable="false" name="[column_name]" />
      <generated-value strategy="TABLE"/>
    </id>
```

basic. CMDB 특성을 선언하는 데 사용됩니다. **name** 및 **column_name** 속성만 편집해야 합니다.

```
    <basic name="name">
      <column updatable="false" insertable="false" name="[column_name]" />
    </basic>
```

판별자가 있는 단일 테이블 상속의 경우 다음과 같이 확장 클래스를 매핑합니다.

```
<entity name="[cmdb_class_name]" class="generic_db_adapter.nt" name="nt">

  <discriminator-value>nt      </discriminator-value>
  <attributes>
</entity>
<entity class="generic_db_adapter.unix" name="unix">
  <discriminator-value>unix</discriminator-value>
  <attributes>
</entity>
<entity name="[CMDB_class_name]" class="generic_db_adapter.[CMDB[cmdb_class_name]">
  <table name="[default_table_name]" />
  <secondary-table name="" />
  <attributes>
    <id name="id1">
      <column updatable="false" insertable="false" name="[column_name]" />
      <generated-value strategy="TABLE"/>
    </id>
    <id name="id2">
      <column updatable="false" insertable="false" name="[column_name]" />
      <generated-value strategy="TABLE"/>
    </id>
    <id name="id3">
      <column updatable="false" insertable="false" name="[column_name]" />
      <generated-value strategy="TABLE"/>
    </id>
```

다음 예는 접두사 없이 CMDB 특성 이름을 보여 줍니다.

```
    <basic name="[CMDB_attribute_name]">
```

```
        <column updatable="false" insertable="false" name="[column_name]" />
      </basic>
      <basic name="[CMDB_attribute_name]">
        <column updatable="false" insertable="false" name="[column_name]" />
      </basic>
      <basic name="[CMDB_attribute_name]">
        <column updatable="false" insertable="false" name="[column_name]" />
      </basic>
    </attributes>
  </entity>
```

관계 엔터티이며, 이름 지정 규칙은 **end1Type_linkType_end2Type**입니다. 이 예에서 **end1Type**은 **node**이고 **linkType**은 **composition**입니다.

```
  <entity name="node_composition_[CMDB_class_name]" class="generic_db_adapter.node_
composition_[CMDB_class_name]">
    <table name="[default_table_name]" />
    <attributes>
      <id name="id1">
        <column updatable="false" insertable="false" name="[column_name]" />
        <generated-value strategy="TABLE"/>
      </id>
```

대상 엔터티는 이 속성이 가리키는 엔터티입니다. 이 예에서 **end1**은 **node** 엔터티에 매핑됩니다.

many-to-one. 여러 관계를 하나의 노드에 연결할 수 있습니다.

join-column. end1 ID(대상 엔터티 ID)가 포함된 열입니다.

referenced-column-name. 조인 열에 사용되는 ID가 포함된 대상 엔터티의 열 이름(**node**)입니다.

```
      <many-to-one target-entity="node" name="end1">
        <join-column updatable="false" insertable="false" referenced-column-name="[column_
name]" name="[column_name]" />
      </many-to-one>
```

one-to-one. 하나의 관계를 하나의 **[CMDB_class_name]**에 연결할 수 있습니다.

```
      <one-to-one target-entity="[CMDB_class_name]" name="end2">
        <join-column updatable="false" insertable="false" referenced-column-name="" name="
[column_name]" />
      </one-to-one>
    </attributes>
  </entity>
</entity-mappings>
```

node attribute. 노드 특성 추가 방법에 대한 예입니다.

```
<entity class="generic_db_adapter.host_node">
 <discriminator-value>host_node</discriminator-value>
  <attributes/>
</entity>
<entity class="generic_db_adapter.nt">
 <discriminator-value>nt</discriminator-value>
  <attributes>
   <basic name="nt_servicepack">
    <column updatable="false" insertable="false" name="specific_type_value"/>
   </basic>
  </attributes>
</entity>
```

다중 ORM 파일

다중 매핑 파일이 지원됩니다. 각 매핑 파일 이름은 orm.xml로 끝나야 합니다. 모든 매핑 파일은 어댑터의 META-INF 폴더 아래에 배치되어 있습니다.

이름 지정 규칙

- **각 엔터티에서 클래스 속성은** generic_db_adapter**의 접두사가 포함된 이름 속성과 일치해야 합니다.**
- **기본 키 열은 idX 형식의 이름을 사용해야 합니다. 여기서 X는 테이블의 기본 키 번호에 따라 1, 2, ...입니다.**
- **특성 이름은 클래스 특성 이름과 일치해야 하며 대/소문자도 같아야 합니다.**
- **관계 이름은** end1Type_linkType_end2Type **형식입니다.**
- **Java에서 예약어이기도 한 CMDB CIT는 gdba_라는 접두사를 포함해야 합니다. 예를 들어 CMDB CIT goto의 경우 ORM 엔터티의 이름은 gdba_goto여야 합니다.**

테이블 이름 대신 인라인 SQL 문 사용

엔터티를 데이터베이스 테이블 대신 인라인 select **절에 매핑할 수 있습니다. 이 매핑은 데이터베이스에서 보기를 정의하고 엔터티를 이 보기에 매핑하는 것과 같습니다. 예:**

```
  <entity class="generic_db_adapter.node">
    <table name="(select d.id as id1, d.name as name , d.os as host_os from
Device d)" />
```

이 예에서 노드 특성은 id, name **및** os**가 아니라** columns id1, name **및** host_os**에 매핑되어야 합니다.**

다음 제한이 적용됩니다.

- 인라인 SQL 문은 Hibernate를 JPA 공급자로 사용하는 경우에만 사용할 수 있습니다.
- 인라인 SQL select 절은 반드시 둥근 괄호 안에 포함해야 합니다.
- **<schema>** 요소는 **orm.xml** 파일에 표시되지 않아야 합니다. Microsoft SQL Server 2005의 경우 이는 <schema>dbo</schema>를 사용하여 전체적으로 테이블 이름을 정의하지 않고 모든 테이블 이름에 dbo.라는 접두사를 추가해야 함을 의미합니다.

orm.xml 스키마

다음 표에는 **orm.xml** 파일의 공통 요소가 설명되어 있습니다. 전체 스키마는 http://java.sun.com/xml/ns/persistence/orm_1_0.xsd에 있습니다. 이 목록은 완벽하지 않지만 일반 데이터베이스 어댑터용 표준 Java Persistence API의 특정 동작이 대부분 설명되어 있습니다.

요소 이름 및 경로	설명	특성
entity-mappings	엔터티 매핑 문서의 루트 요소입니다. 이 요소는 GDBA 샘플 파일에 주어진 요소와 같아야 합니다.	
description (entity-mappings)	엔터티 매핑 문서에 대한 자유로운 텍스트 설명입니다 (선택 사항).	
package (entity-mappings)	매핑 클래스를 포함할 Java 패키지의 이름입니다. 항상 `generic_db_adapter` 텍스트를 포함해야 합니다.	1. **이름:** name **설명:** 이 엔터티를 매핑할 UCMDB CI 유형의 이름입니다. CMDB의 링크에 매핑된 엔터티인 경우 이 엔터티의 이름은 `<end_1>_<link_name>_<end_2>` 형식이어야 합니다. 예를 들어 `node_composition_cpu`는 노드와 CPU 간의 Composition 링크에 매핑할 엔터티를 정의합니다. CI 유형의 이름이 패키지 접두사가 없는 Java 클래스의 이름과 같으면 이 필드를 생략할 수 있습니다. **필수 여부:** 선택적 **유형:** 문자열 2. **이름:** class **설명:** 이 DB 엔터티에 대해 만들 Java 클래스의 정규화된 이름입니다. Java 클래스의 패키지 이름은 `package` 요소에 주어진 이름과 같아야 합니다. 클래스 이름으로

요소 이름 및 경로	설명	특성
		Java 예약어(예: 인터페이스 또는 스위치)는 사용할 수 없습니다. 대신 `gdba_`라는 접두사를 이름에 추가합니다(따라서 인터페이스는 `generic_db_adapter.gdba_interface`가 됨). **필수 여부:** 필수 **유형:** 문자열
table (entity-mappings > entity)	이 요소는 DB 엔터티의 기본 테이블을 정의합니다. 한 번만 표시될 수 있습니다. 필수 항목입니다.	**이름:** name **설명:** 기본 테이블의 이름입니다. 테이블 이름에 테이블이 속하는 스키마가 포함되어 있지 않으면 통합 포인트를 만드는 데 사용된 사용자의 스키마에서만 테이블을 검색합니다. 이는 유효한 SELECT 문일 수도 있습니다. SELECT 문인 경우에는 괄호로 캡슐화해야 합니다. **필수 여부:** 필수 **유형:** 문자열
secondary-table (entity-mappings > entity)	이 요소는 DB 엔터티의 보조 테이블을 정의하는 데 사용할 수 있습니다. 이 테이블은 기본 테이블에 일대일 관계로 연결해야 합니다. 보조 테이블을 둘 이상 정의할 수 있으며, 선택 사항입니다.	**이름:** name **설명:** 보조 테이블의 이름입니다. 테이블 이름에 테이블이 속하는 스키마가 포함되어 있지 않으면 통합 포인트를 만드는 데 사용된 사용자의 스키마에서만 테이블을 검색합니다. 이는 유효한 SELECT 문일 수도 있습니다. SELECT 문인 경우에는 괄호로 캡슐화해야 합니다. **필수 여부:** 필수 **유형:** 문자열
primary-key-join-column (entity-mappings > entity > secondary-table)	이름이 같은 필드를 사용하여 보조 테이블과 기본 테이블을 연결하지 않으면 이 요소는 기본 테이블의 기본 키 필드에 연결해야 하는 보조 테이블의 기본 키 필드 이름을 정의합니다.	**이름:** name **설명:** 보조 테이블의 기본 키 필드 이름입니다. 이 요소가 없으면 기본 키 필드 이름이 기본 테이블의 기본 키 필드 이름과 같다고 간주합니다. **필수 여부:** 선택적 **유형:** 문자열
inheritance (entity-mappings > entity)	현재 엔터티가 DB 엔터티 패밀리의 상위 엔터티인 경우 이 요소를 사용하여 해당 상위 엔터티로 표시	**이름:** strategy **설명:** DB에서 상속을 구현하는 방법을 정의합니다.

요소 이름 및 경로	설명	특성
	하며, 선택 사항입니다.	**필수 여부:** 필수 **유형:** 다음 값 중 하나입니다. • SINGLE_TABLE: 이 엔터티와 모든 하위 엔터티가 같은 테이블에 있습니다. • JOINED: 하위 엔터티가 조인된 테이블에 있습니다. • TABLE_PER_CLASS: 각 엔터티가 완전히 별도의 테이블에 정의되어 있습니다.
discriminator-column (entity-mappings > entity)	상속이 SINGLE_TABLE 유형인 경우 이 요소를 사용하여 각 행의 엔터티 유형을 확인하는 데 사용되는 필드의 이름을 정의합니다.	**이름:** name **설명:** 판별자 열의 이름입니다. **필수 여부:** 필수 **유형:** 문자열
discriminator-value (entity-mappings > entity)	이 요소는 상속 트리의 특정 엔터티 유형을 정의합니다. 이 이름은 이 특정 엔터티 유형의 값 그룹에 대해 **discriminator.properties** 파일에 정의된 이름과 같아야 합니다.	
attributes (entity-mappings > entity)	엔터티에 대한 모든 특성 매핑의 루트 요소입니다.	
id (entity-mappings > entity attributes)	이 요소는 엔터티의 키 필드를 정의합니다. 하나 이상의 id 필드를 정의해야 합니다. id 요소가 둘 이상 있으면 해당 필드가 엔터티에 대한 복합 키를 만듭니다. 링크가 아닌 CI 엔터티에 대한 복합 키는 피해야 합니다.	**이름:** name **설명:** idX 유형의 문자열입니다. 여기서 X는 1부터 9까지의 숫자입니다. 첫 번째 id는 id1, 두 번째는 id2 등으로 표시합니다. UCMDB의 키 특성 이름은 아닙니다. **필수 여부:** 필수 **유형:** 문자열
basic (entity-mappings > entity attributes)	이 요소는 테이블의 기본 키에는 포함되지 않는 테이블의 필드와 UCMDB 특성 간 매핑을 정의합니다.	**이름:** name **설명:** 이 필드를 매핑할 UCMDB 특성의 이름입니다. 이 특성은 현재 엔터티를 매핑할 UCMDB CI 유형에 있어야 합니다. **필수 여부:** 필수

요소 이름 및 경로	설명	특성
		유형: 문자열
column (entity-mappings > entity > attributes >id -또는- (entity-mappings > entity > attributes > basic)	기본 매핑 또는 id 필드에 대한 테이블의 열 이름을 정의합니다.	1. **이름:** name **설명:** 필드의 이름입니다. **필수 여부:** 필수 **유형:** 문자열 2. **이름:** table **설명:** 필드가 속하는 테이블의 이름입니다. 이 테이블 이름은 엔터티에 대해 정의된 기본 테이블이거나 보조 테이블 중 하나여야 합니다. 이 특성이 생략되면 해당 필드가 기본 테이블에 속하는 것으로 간주됩니다. **필수 여부:** 선택적 **유형:** 문자열
one-to-one (entity-mappings > entity > attributes)	값이 다른 테이블에 있는 열을 정의하고, 두 테이블을 일대일 관계를 사용하여 연결합니다. 이 요소는 링크 엔터티 매핑에 대해서만 지원되고 다른 CI 유형에 대해서는 지원되지 않습니다. 이 요소는 테이블과 UCMDB 링크 간의 매핑을 정의하는 유일한 방법입니다.	1. **이름:** name **설명:** 이 필드가 나타내는 양끝 중 하나입니다. **필수 여부:** 필수 **유형:** `end1` 또는 `end2` 2. **이름:** target-entity **설명:** 끝이 참조할 엔터티의 이름입니다. **필수 여부:** 필수 **유형:** 엔터티 매핑 문서에 정의된 엔터티 이름 중 하나입니다.
join-column (entity-mappings > entity attributes > one-to-one)	상위 일대일 요소에 정의된 대상-엔터티와 현재 엔터티를 조인하는 방법을 정의합니다.	1. **이름:** name **설명:** 현재 테이블에서 일대일 조인을 수행하는 데 사용할 필드의 이름입니다. **필수 여부:** 필수 **유형:** 문자열 2. **이름:** name **설명:** 조인을 수행할 기준이 되는 조인트 엔터티의 필드 이름입니다. 이 특성을 생략하면 조인트 테이블에 이름 특성에서 정의된 필드와 이름이 같은 열이 있는 것으로 간

요소 이름 및 경로	설명	특성
		주합니다. **필수 여부:** 선택적 **유형:** 문자열

orm.xml 파일 만들기의 예

여기에 제시된 예에서는 orm.xml 파일을 만드는 방법을 보여줍니다. 이 예제에서는 원격 데이터베이스의 SQL 테이블이 UCMDB의 CI 유형에 매핑됩니다.

원격 데이터베이스에서 다음과 같은 형식의 테이블에서 노드로 Hosts 테이블을 채우고, IP 주소로 IP_Addresses 테이블을 채운 후, 다음과 같이 노드와 IP 주소 간에 링크를 만듭니다.

Hosts 테이블

host_name	host_id
Test1	1
Test2	2
Test3	3

IP_Addresses 테이블

ip_address	ip_id
10.1.1.1	1
10.2.2.2	2
10.3.3.2	3
10.4.4.4	4

Host_IP_Link 테이블(노드와 IP 주소 사이의 링크)

host_id	ip_id
1	1
2	2
2	3
3	4

Hosts 테이블의 기본 키는 host_id 필드이며 IP_Addresses Table 테이블의 기본 키는 ip_id 필드입니다. Host_IP_Link 테이블에서 host_id와 ip_id는 Hosts Table 및 IP_Addresses Table의 외래 키입니다.

위의 테이블에 따라 다음 단계를 수행하여 **orm.xml** 파일을 만듭니다. 이 예에서 사용되는 엔터티는 **node**, **ip_address** 및 **node_containment_ip_address**입니다.

1. 다음과 같이 **Hosts** 테이블에서 **host_id**를 매핑하여 **node** 엔터티를 만듭니다.

```
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0"
      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
     /orm_1_0.xsd">
       <description>test_integration</description>
       <package>generic_db_adapter</package>
       <entity class="generic_db_adapter.node">
            <table name="Hosts"/>
            <attributes>
      <id name="id1">
           <column updatable="false" insertable="false" name=
                "host_id"/>
           <generated-value strategy="TABLE"/>
      </id>
      <basic name="name">
           <column updatable="false" insertable="false" name=
                "host_name"/>
      </basic>
            </attributes>
       </entity>
```

엔터티 **class**는 UCMDB에 이미 있는 CI 유형이어야 합니다. 테이블 **name**은 ID와 호스트 정보를 모두 포함하는 데이터베이스 내의 테이블입니다. ID 특성은 특정 호스트를 식별하는 데 필요하며 나중에 매핑에 사용됩니다. 이 예에서 이 엔터티의 **name** 특성에는 Hosts 테이블의 **host_name** 열 내용을 채웁니다.

2. 다음 엔터티에서는 인터페이스 테이블에서 IP 주소를 매핑합니다.

```
<entity name="ip_address" class="generic_db_adapter.ip_address">
       <table name="IP_Addresses"/>
       <attributes>
            <id name="id1">
```

```
    <column insertable="false" updatable="false" name="ip_id"/>
    <generated-value strategy="TABLE"/>
        </id>
        <basic name="name">
    <column updatable="false" insertable="false" name="ip_address"/>
        </basic>
     </attributes>
</entity>
```

3. 다음으로는 매핑 테이블을 사용하고 **ip_id** 필드를 참조하여 노드와 IP 주소 사이의 링크를 만듭니다 (원하는 경우 **host_id** 및 **ip_id** 필드 모두를 참조 가능).

```
<entity name="node_containment_ip_address"
    class="generic_db_adapter.node_containment_ip_address">
     <table name="Host_IP_Link"/>
     <attributes>
        <id name="id1">
    <column updatable="false" insertable="false" name="ip_id"/>
    <generated-value strategy="TABLE"/>
        </id>
        <many-to-one target-entity="node" name="end1">
    <join-column name="host_id"/>
        </many-to-one>
        <one-to-one target-entity="ip_address" name="end2">
    <join-column name="ip_id"/>
        </one-to-one>
     </attributes>
</entity>
```

컨테이너의 엔터티 이름 형식은 [end1 CIT]_[link CIT]_[end2 CIT]와 같습니다. 따라서 이 예제에서는 링크 CI 유형이 **containment**이므로 컨테이너의 엔터티 이름은 **node_containment_ip_address**이고 엔터티 클래스는 **generic_db_adapter.node_containment_ip_address**입니다. 해당 ID는 이 코드 블록에 필요하며 이 예제에서는 인터페이스의 단일 ID를 사용하지만 양쪽 열에서는 id1 및 id2 참조를 사용할 수 있습니다. 해당 코드는 다음과 같습니다.

```
<id name="id1">
        <column updatable="false" insertable="false" name="ip_id"/>
        <generated-value strategy="TABLE"/>
</id>
<id name="id2">
        <column updatable="false" insertable="false" name="host_id"/>
        <generated-value strategy="TABLE"/>
</id>
```

이 링크의 양쪽 끝은 '다대일' 및 '일대일'이며, 이는 각 IP 주소가 1 노드에 링크되지만 한 노드가 여러 IP 주소에 링크될 수 있음을 의미합니다. 포함되는 열은 Links 테이블에서 가져오며 Hosts 테이블과 Interfaces 테이블을 참조합니다.

각 원격 제품 버전에 대해 특정 orm.xml 구성

어댑터가 지정된 원격 제품 버전에 대해 특정 **orm.xml**을 사용하도록 특정 **orm.xml** 파일을 구성할 수 있습니다. 예를 들어 원격 데이터 저장소에 두 가지 제품 버전 x 및 y가 있는 경우 각 버전에 대해 다른 엔터티 매핑이 있을 수 있습니다.

원격 제품 버전별로 특정 orm.xml 파일을 구성하려면 다음을 수행합니다.

1. **adapter.xml** 파일에 **version** 매개 변수를 추가하고 가능한 버전 값을 **valid-values**로 지정합니다.
2. 어댑터 패키지의 META-INF 폴더 아래에 **VersionOrm** 폴더를 만듭니다.
3. **VersionOrm** 폴더에서 각 특정 버전에 대한 **orm.xml** 파일을 만듭니다. 파일 이름에는 버전 접두사가 포함되어야 합니다. 예를 들어 버전이 **x**인 경우 파일 이름은 **x_orm.xml**이어야 합니다.

참고: 원격 제품 버전에 대해 특정 **orm.xml** 파일을 만드는지 여부에 관계없이 모든 원격 제품 버전에 대해 META-INF 폴더의 **orm.xml** 파일이 로드됩니다. 이 파일에는 모든 버전에 대해 동일한 방식으로 매핑되는 엔터티가 포함될 수 있습니다.

reconciliation_types.txt 파일

UCMDB 10.00 버전부터 **reconciliation_types.txt** 파일은 더 이상 관련이 없습니다. 임의의 CIT를 재조정에 사용할 수 있습니다. 연합 엔진은 자동으로 매핑을 실행합니다.

reconciliation_rules.txt 파일(이전 버전과의 호환용)

어댑터에서 DBMappingEngine을 구성한 경우 조정을 수행하려면 이 파일을 사용하여 조정 규칙을 구성합니다. DBMappingEngine을 사용하지 않는 경우 일반 UCMDB 조정 메커니즘이 사용되므로 이 파일을 구성할 필요가 없습니다.

파일의 각 행은 규칙을 나타냅니다. 예:

multinode[node] expression[^node.name OR ip_address.name] end1_type[node] end2_type[ip_address] link_type[containment]

다중 노드에는 다중 노드 이름(TQL 쿼리에서 연합 데이터베이스 CIT에 연결된 CMDB CIT)이 입력됩니다.

이 식에는 두 개의 다중 노드(CMDB의 다중 노드와 데이터베이스 원본의 다중 노드)가 동일한지 여부를 결정하는 논리가 포함됩니다.

식은 OR 또는 AND로 구성됩니다.

식 부분에서 특성 이름과 관련된 규칙은 [className].[attributeName]입니다. 예를 들어 ip_address 클래스의 attributeName은 작성된 ip_address.name입니다.

순서 일치의 경우(첫 번째 OR 하위 식은 다중 노드가 서로 다른 답을 반환하고, 두 번째 OR 하위 식은 비교하지 않는 경우) 식 대신 순서 식을 사용합니다.

비교할 때 대/소문자를 무시하려면 제어 기호(^)를 사용하십시오.

end1_type, end2_type 및 link_type 매개 변수는 조정 TQL 쿼리에 두 개의 노드가 포함된 경우에만 사용되고 다중 노드 하나만 포함된 경우에는 사용되지 않습니다. 이 예에서 조정 TQL 쿼리는 end1_type > (link_type) > end2_type입니다.

관련 레이아웃은 식에서 가져오므로 추가할 필요가 없습니다.

조정 규칙 유형

조정 규칙은 OR 및 AND 조건 형태를 사용합니다. 서로 다른 여러 노드에 대해 이러한 규칙을 정의할 수 있습니다(예: node의 이름AND/ORip_address의 이름으로 노드 식별).

다음 옵션은 일치하는 항목을 찾습니다.

- **순서 일치.** 조정 식은 왼쪽에서 오른쪽으로 읽습니다. 두 개의 OR 하위 식에 값이 있고 해당 값이 서로 같은 경우 두 하위 식이 같은 것으로 간주됩니다. 두 개의 OR 하위 식에 값이 있고 해당 값이 서로 다른 경우 두 하위 식이 다른 것으로 간주됩니다. 결정된 바가 없는 경우에는 다음 OR 하위 식이 같은지 테스트합니다.

  node의 이름 OR ip_address의 이름. CMDB과 데이터 원본에 모두 이름이 포함되어 있고 이 이름이 서로 같으면 노드가 같은 것으로 간주됩니다. 둘 모두에 이름이 있지만 같지 않은 경우에는 ip_address의 이름을 테스트하지 않고 노드가 다른 것으로 간주됩니다. CMDB 또는 데이터 원본에 node의 이름이 없는 경우 ip_address의 이름을 확인합니다.
- **정규식 일치.** OR 하위 식 중 하나가 같은 경우 CMDB과 데이터 원본이 같은 것으로 간주됩니다.

  node의 이름 OR ip_address의 이름. node의 이름이 일치하지 않으면 ip_address의 이름이 같은지 확인합니다.

클래스 모델에서 조정 엔터티가 관계를 포함한 여러 CIT(예: node)로 모델링되는 복합 조정의 경우 모델링된 모든 CIT의 모든 관련 특성이 상위 집합 노드의 매핑에 포함됩니다.

참고: 따라서 데이터 원본의 모든 조정 특성이 같은 기본 키를 공유하는 테이블에 있어야 한다는 제한이 있습니다.

다른 제한은 조정 TQL 쿼리에 둘 이하의 노드만 포함해야 한다는 것입니다. 예를 들어 node > ticket TQL 쿼리에는 CMDB의 노드 하나와 데이터 원본의 티켓 하나가 있습니다.

결과를 조정하려면 node 및/또는 ip_address에서 이름이 검색되어야 합니다.

CMDB의 이름이 *.m.com 형식이면 CMDB에서 연합 데이터베이스로 또는 그 반대로 변환기를 사용하여 이러한 값을 변환할 수 있습니다.

데이터베이스 티켓 테이블의 node_id 열을 사용하여 엔터티 간에 연결합니다(노드 테이블에서 정의된 연관을 만들 수도 있음).

DB 노드	
PK	node_id
	name

DB IP_Address	
PK	ip_id
	name

DB Ticket	
PK	ticket_id
	node_id

참고: 위의 세 테이블은 CMDB 데이터베이스가 아니라 연합 RDBMS 원본의 테이블이어야 합니다.

transformations.txt 파일

이 파일에는 모든 변환기 정의가 포함되어 있습니다.

각 줄에 새 정의가 포함된 형식입니다.

transformations.txt 파일 템플릿

entity[[CMDB_class_name]] attribute[[CMDB_attribute_name]] to_DB_class
[com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.
transform.impl.GenericEnumTransformer(generic-enum-transformer-example.xml)]
from_DB_class[com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.
GenericEnumTransformer(generic-enum-transformer-example.xml)]

entity. orm.xml 파일에 표시되는 엔터티 이름입니다.

attribute. orm.xml 파일에 표시되는 특성 이름입니다.

to_DB_class.
com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.FcmdbDalTransformerTo
ExternalDB 인터페이스를 구현하는 클래스의 정규화된 이름입니다. 괄호 안의 요소가 이 클래스 생성자

에 지정됩니다. 예를 들어 각 노드 이름에 **.com**이라는 접미사를 추가하려는 경우 이 변환기를 사용하여 CMDB 값을 데이터베이스 값으로 변환합니다.

from_DB_class. **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.FcmdbDalTransformerFromExternalDB** 인터페이스를 구현하는 클래스의 정규화된 이름입니다. 괄호 안의 요소가 이 클래스 생성자에 지정됩니다. 예를 들어 각 노드 이름에 **.com**이라는 접미사를 추가하려는 경우 이 변환기를 사용하여 데이터베이스 값을 CMDB 값으로 변환합니다.

자세한 내용은 "기본 변환기"(165페이지)를 참조하십시오.

discriminator.properties 파일

이 파일은 지원되는 각 CI 유형(orm.xml에서 판별자 값으로도 사용됨)을 판별자 열의 가능한 해당 값 목록(쉼표로 구분된 목록) 또는 판별자 열의 가능한 값과 일치시키는 조건에 매핑합니다.

조건을 사용할 경우 like(condition) 구문을 사용합니다. 여기서 condition은 다음과 같은 와일드카드를 포함할 수 있는 문자열입니다.

- **%**(백분율 기호) - 모든 길이(길이가 0인 문자열 포함)의 문자열을 일치시킬 수 있습니다.
- **_**(밑줄) - 단일 문자를 일치시킬 수 있습니다.

예를 들어, like(%unix%)는 unix, linux, unix-aix 등과 일치합니다. Like 조건은 단일 열에만 적용됩니다.

'all-other'를 표시하여 다른 판별자에 속하지 않는 값에 단일 판별자 값을 매핑할 수도 있습니다.

만들려는 어댑터에서 판별자 기능을 사용할 경우 **discriminator.properties** 파일에서 모든 판별자 값을 정의해야 합니다.

판별자 매핑의 예:

예를 들어, 어댑터가 CI 유형 node, nt 및 unix를 지원하고 데이터베이스에 **type**이라는 열이 있는 단일 테이블 t_nodes가 포함되어 있습니다. 유형이 10001인 경우 행은 노드를 나타내고, 유형이 10004인 경우 unix 시스템을 나타내는 식입니다. **discriminator.properties** 파일의 형태는 다음과 같습니다.

```
node=10001, 10005
nt=10002,10003
unix=2%
mainframe=all-other
```

orm.xml 파일에는 다음 코드가 포함되어 있습니다.

```
<entity class="generic_db_adapter.node" >
   <table name="t_nodes" />
   ...
   <inheritance strategy="SINGLE_TABLE" />
   <discriminator-value>node</discriminator-value>
   <discriminator-column name="type" />
   ...
</entity>
<entity class="generic_db_adapter.nt" name="nt">
```

```
    <discriminator-value>nt      </discriminator-value>
    <attributes>
  </entity>
  <entity class="generic_db_adapter.unix" name="unix">
    <discriminator-value>unix</discriminator-value>
    <attributes>
  </entity>
```

discriminator_column 특성은 다음과 같이 계산됩니다.

- 특정 항목에 대해 **type**에 10002 또는 10003이 포함되어 있으면 이 항목이 **nt** CIT에 매핑됩니다.
- 특정 항목에 대해 **type**에 10001 또는 10005가 포함되어 있으면 이 항목이 **node** CIT에 매핑됩니다.
- 특정 항목에 대해 **type**이 2로 시작되는 경우 이 항목이 **unix** CIT에 매핑됩니다.
- **type** 열의 기타 모든 값은 **mainframe** CIT에 매핑됩니다.

참고: node CIT도 **nt** 및 **unix**의 상위 항목입니다.

replication_config.txt 파일

이 파일은 쉼표로 구분된 CI 목록과, 해당 속성 조건이 복제 플러그인을 통해 지원되는 관계 유형을 포함합니다. 자세한 내용은 "플러그인"(170페이지)을 참조하십시오.

fixed_values.txt 파일

이 파일을 사용하면 특정 CIT의 특정 특성에 대해 고정값을 구성할 수 있습니다. 이 방식으로 데이터베이스에 저장되지 않는 고정값을 이러한 각각의 특성에 할당할 수 있습니다.

이 파일은 다음 형식의 항목을 0개 이상 포함해야 합니다.

entity[<entityName>] attribute[<attributeName>] value[<value>]

예:

entity[ip_address] attribute[ip_domain] value[DefaultDomain]

이 파일은 상수 목록도 지원합니다. 상수 목록을 정의하려면 다음 구문을 사용합니다.

entity[<entityName>] attribute[<attributeName>] value[{<Val1>, <Val2>, <Val3>, ... }]

Persistence.xml 파일

이 파일은 기본 Hibernate 설정을 다시 정의하고 기본이 아닌 데이터베이스 유형(기본 데이터베이스 유형은 Oracle Server, Microsoft SQL Server, MySQL)에 대한 지원을 추가하는 데 사용됩니다.

새 데이터베이스 유형을 지원해야 할 경우 연결 풀 공급자(기본값: c3p0) 및 사용할 데이터베이스용 JDBC 드라이버를 제공(*.jar 파일을 어댑터 폴더에 배치)해야 합니다.

변경할 수 있는 모든 Hibernate 값을 보려면 **org.hibernate.cfg.Environment** 클래스를 확인하십시오(자세한 내용은 http://www.hibernate.org 참조).

persistence.xml 파일의 예:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
   <!-- Don't change this value -->
   <persistence-unit name="GenericDBAdapter">
      <properties>
         <!--  Don't change this value  -->
         <property name="hibernate.archive.autodetection" value="class,
           hbm" />
         <!--The driver class name"/-->
         <property name="hibernate.connection.driver_class" value="com.
          mercury.jdbc.MercOracleDriver" />
         <!--The connection url"/-->
         <property name="hibernate.connection.url" value="jdbc:mercury:
          oracle://artist:1521;sid=cmdb2" />
         <!--DB login credentials"/-->
         <property name="hibernate.connection.username" value="CMDB" />
         <property name="hibernate.connection.password" value="CMDB" />
         <!--connection pool properties"/-->
         <property name="hibernate.c3p0.min_size" value="5" />
         <property name="hibernate.c3p0.max_size" value="20" />
         <property name="hibernate.c3p0.timeout" value="300" />
         <property name="hibernate.c3p0.max_statements" value="50" />
         <property name="hibernate.c3p0.idle_test_period" value="3000" />
         <!--The dialect to use-->
         <property name="hibernate.dialect" value="org.hibernate.dialect.
           OracleDialect" />
      </properties>
   </persistence-unit>
</persistence>
```

NT 인증을 사용하여 데이터베이스에 연결

NT 인증이 필요한 MS SQL Server에 연결할 수 있습니다. 이렇게 하려면 도메인의 구문을 분석할 수 있는 드라이버(즉, jTDS JDBC 드라이버)가 필요합니다.

인증은 현재 실행 중인 프로세스 NT 자격 증명을 사용하지 않고 제공된 매개 변수(도메인, 사용자 이름, 비밀번호)에 따라 수행됩니다.

1. **persistence.xml**에서 다음과 같이 속성을 편집합니다.

```
<!--The driver class name"/-->
```

```
<property name="hibernate.connection.driver_class"
value="net.sourceforge.jtds.jdbc.Driver"/>
<property name="hibernate.connection.url" value="jdbc:jtds:sqlserver://[host name]:
[port];DatabaseName=[database name];domain=[the domain]"/>
<!--DB login credentials"/-->
<property name="hibernate.connection.username" value="[username]"/>
<property name="hibernate.connection.password" value="[password]"/>
```

2. JDBC 드라이버 파일을 **<프로브 설치 폴더>\lib** 아래에 배치합니다.
3. 프로브를 다시 시작합니다.

SCCM 통합에 NTLM 인증을 사용하도록 Persistence.xml 파일을 구성

참고: 이 섹션은 SCCM 통합에만 적용됩니다.

SCCM 통합에 NTLM 인증을 사용하려면 **persistence.xml**을 다음과 같이 구성합니다.

1. JDBC 드라이버 파일을 **<probe installation folder>\lib** 아래에 배치합니다.

   예를 들면, http://sourceforge.net/projects/jtds/files/에 있는 **jtds-1.3.1.jar** 파일을 **DataFlowProbe\lib** 폴더에 넣습니다.
2. 서버 및 프로브를 시작합니다.
3. UCMDB에서 **데이터 흐름 관리 > 어댑터 관리 > SCCMAdapter >** 로 이동합니다.
4. 리소스 창에서 **패키지 > SCCMAdapter > 구성 파일** 폴더에 있는 SCCM 어댑터 구성 파일을 선택합니다.
5. **adapter.conf** 파일에서 **dal.use.persistence.xml=true**를 설정합니다.
6. **persistence.xml** 파일에서 다음을 추가합니다.

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://java.sun.com/xml/ns/persistence
 http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
  <persistence-unit name="GenericDBAdapter">
    <properties>
      <!-- added to fix: org.hibernate.HibernateException: 'hibernate.dialect' must be set when
no Connection avalable -->
      <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>

      <!--The driver class name"/-->
      <property name="hibernate.connection.driver_class"
value="net.sourceforge.jtds.jdbc.Driver"/>
      <property name="hibernate.connection.url" value="jdbc:jtds:sqlserver://<DB_
```

```
host>:<port>;DatabaseName=<DB_name>;domain=<domain_name> "/>
    </properties>
  </persistence-unit>
</persistence>
```

참고: 강조 표시된 부분을 해당 연결 URL로 바꿉니다.

7. **persistence.xml** 파일에 사용자 또는 비밀번호가 필요하지 않습니다.
8. **데이터 흐름 관리 > 통합 스튜디오**로 이동한 후 **새 통합 포인트**  버튼을 클릭합니다.
9. 필수 필드에 값을 입력합니다.

   자격 증명 ID를 입력해야 하는 경우 다음을 수행합니다.

   a. 자격 증명 선택 대화 상자에서 왼쪽 프로토콜 창의 **Generid DB Protocol(SQL)**을 선택합니다.

   b. 오른쪽 자격 증명 창에서 **선택한 프로토콜 유형에 대한 새 연결 세부 정보 만들기**  버튼을 클릭합니다.

   c. 새로 만들기 대화 상자에서 데이터베이스 유형으로 **MicrosoftSQLServerNTLM**을 선택합니다.

   d. 포트 번호를 입력합니다.

   e. 사용자 이름을 **도메인\사용자 이름** 형식으로 입력합니다.

   f. 비밀번호를 입력합니다.

기본 변환기

다음 변환기를 사용하여 연합 쿼리 및 복제 작업을 데이터베이스 데이터로 변환하거나 그 반대로 변환할 수 있습니다.

이 섹션에는 다음 항목이 포함됩니다.

enum-transformer 변환기

이 변환기는 입력 매개 변수로 주어진 XML 파일을 사용합니다.

XML 파일은 하드 코드된 CMDB 값과 데이터베이스 값(열거형)을 매핑합니다. 해당 값 중 하나가 없으면 같은 값 또는 null을 반환하거나 예외를 발생시키도록 선택할 수 있습니다.

변환기는 대/소문자를 구분하거나 대/소문자를 구분하지 않는 메서드를 사용하여 두 문자열을 비교합니다. 기본 동작은 대/소문자를 구분합니다. 대/소문자를 구분하지 않도록 정의하려면 enum-transformer 요소에 case-sensitive="false"를 사용합니다.

각 엔터티 특성에 하나의 XML 매핑 파일을 사용합니다.

참고: 이 변환기는 **transformations.txt** 파일의 to_DB_class 필드와 from_DB_class 필드에 모두 사용할 수 있습니다.

입력 파일 XSD:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
 <xs:element name="enum-transformer">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="value" minOccurs="0" maxOccurs="unbounded"/>
   </xs:sequence>
   <xs:attribute name="db-type" use="required">
    <xs:simpleType>
     <xs:restriction base="xs:string">
      <xs:enumeration value="integer"/>
      <xs:enumeration value="long"/>
      <xs:enumeration value="float"/>
      <xs:enumeration value="double"/>
      <xs:enumeration value="boolean"/>
      <xs:enumeration value="string"/>
      <xs:enumeration value="date"/>
      <xs:enumeration value="xml"/>
      <xs:enumeration value="bytes"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:attribute>
   <xs:attribute name="cmdb-type" use="required">
    <xs:simpleType>
     <xs:restriction base="xs:string">
```

```
            <xs:enumeration value="integer"/>
            <xs:enumeration value="long"/>
            <xs:enumeration value="float"/>
            <xs:enumeration value="double"/>
            <xs:enumeration value="boolean"/>
            <xs:enumeration value="string"/>
            <xs:enumeration value="date"/>
            <xs:enumeration value="xml"/>
            <xs:enumeration value="bytes"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="non-existing-value-action" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="return-null"/>
            <xs:enumeration value="return-original"/>
            <xs:enumeration value="throw-exception"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="case-sensitive" use="optional">
        <xs:simpleType>
          <xs:restriction base="xs:boolean">
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:element name="value">
    <xs:complexType>
      <xs:attribute name="cmdb-value" type="xs:string" use="required"/>
```

```
    <xs:attribute name="external-db-value" type="xs:string" use="required"/>
    <xs:attribute name="is-cmdb-value-null" type="xs:boolean" use="optional"/>
    <xs:attribute name="is-db-value-null" type="xs:boolean" use="optional"/>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

'sys' 값을 'System' 값으로 변환하는 작업의 예:

이 예에서 CMDB의 sys 값은 연합 데이터베이스의 System 값으로 변환되고 연합 데이터베이스의 System 값은 CMDB의 sys 값으로 변환됩니다.

값이 XML 파일에 없으면(예: demo 문자열) 변환기가 받은 입력 값과 같은 값을 반환합니다.

```
<enum-transformer CMDB-type="string" DB-type="string" non-existing-value-action="return-original" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../META-CONF/generic-enum-transformer.xsd">
  <value CMDB-value="sys" external-DB-value="System" />
</enum-transformer>
```

외부 또는 CMDB 값을 Null 값으로 변환하는 예:

이 예제에서는 원격 데이터베이스의 **NNN** 값을 CMDB 데이터베이스에서 null 값으로 변환합니다.

```
<value cmdb-value="null" is-cmdb-value-null="true" external-db-value="NNN"/>
```

이 예제에서는 CMDB의 **OOO** 값을 원격 데이터베이스의 null 값으로 변환합니다.

```
<value cmdb-value="OOO" external-db-value="null" is-db-value-null="true"/>
```

SuffixTransformer 변환기

이 변환기는 CMDB 또는 연합 데이터베이스 원본 값에서 접미사를 추가하거나 제거하는 데 사용됩니다.

두 가지 구현이 있습니다.

- **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.AdapterToCmdb AddSuffixTransformer**. 연합 데이터베이스 값에서 CMDB 값으로 변환할 때 접미사(입력을 통해 지정)를 추가하고 CMDB 값에서 연합 데이터베이스 값으로 변환할 때 접미사를 제거합니다.
- **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.AdapterToCmdb RemoveSuffixTransformer**. 연합 데이터베이스 값에서 CMDB 값으로 변환할 때 접미사(입력을 통해 지정)를 제거하고 CMDB 값에서 연합 데이터베이스 값으로 변환할 때 접미사를 추가합니다.

PrefixTransformer 변환기

이 변환기는 CMDB 또는 연합 데이터베이스 값에서 접두사를 추가하거나 제거하는 데 사용됩니다.

두 가지 구현이 있습니다.

- **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.AdapterToCmdb AddPrefixTransformer**. 연합 데이터베이스 값에서 CMDB 값으로 변환할 때 접두사(입력을 통해 지정)를 추가하고 CMDB 값에서 연합 데이터베이스 값으로 변환할 때 접두사를 제거합니다.
- **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.AdapterToCmdb RemovePrefixTransformer**. 연합 데이터베이스 값에서 CMDB 값으로 변환할 때 접두사(입력을 통해 지정)를 제거하고 CMDB 값에서 연합 데이터베이스 값으로 변환할 때 접두사를 추가합니다.

BytesToStringTransformer 변환기

이 변환기는 CMDB의 바이트 배열을 연합 데이터베이스 원본의 문자열 표시로 변환하는 데 사용됩니다.

변환기는 **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.CmdbToAdapterBytes ToStringTransformer**입니다.

StringDelimitedListTransformer 변환기

이 변환기는 CMDB에서 단일 문자열 목록을 정수/문자열 목록으로 변환하는 데 사용됩니다.

변환기는 **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl. StringDelimitedListTransformer**입니다.

사용자 지정 변환기

사용자 지정 변환기를 처음부터 직접 작성할 수 있습니다. 이 경우 필요에 맞는 변환기를 원하는 대로 만들 수 있습니다.

사용자 지정 변환기를 작성하는 방법에는 다음 두 가지가 있습니다.

1. 컴파일된 Java 변환기 작성
   a. Java IDE(예: Eclipse, Intellij 또는 Netbeans)에서 Java 프로젝트를 만듭니다.
   b. 클래스 경로에 federation-api.jar 및 db-interfaces.jar을 추가합니다.
   c. **db-interfaces.jar**에서 다음 인터페이스를 구현하는 Java 클래스를 만듭니다.
      - FcmdbDalTransformerFromExternalDB
      - FcmdbDalTransformerValuesToExternalDB
      - FcmdbDalTransformerInit
   d. 프로젝트를 컴파일하고 jar 파일을 만듭니다.
   e. jar 파일을 어댑터의 패키지(adapterCode\<어댑터 ID> 아래)에 배치합니다.
   f. 패키지를 배포합니다.
   g. **transformations.txt** 파일에 새 변환기 클래스 이름을 추가합니다.
2. Groovy (스크립트 기반) 변환기 작성

   원본 GDBA 패키지에서 예제 **GroovyExampleTransformer.groovy**를 찾을 수 있습니다.
   a. 어댑터의 패키지(adapterCode\<어댑터 ID> 아래)에 Groovy 파일을 만듭니다. 어댑터 관리 메뉴를 사용하여 직접 이렇게 할 수 있습니다.
   b. **db-interfaces.jar**에서 다음 인터페이스를 구현하는 Groovy 클래스를 만듭니다.

- FcmdbDalTransformerFromExternalDB
- FcmdbDalTransformerValuesToExternalDB
- FcmdbDalTransformerInit

c. **transformations.txt** 파일에 새 변환기 Groovy 클래스 이름을 추가합니다.

참고: Groovy는 Java를 확장하는 스크립팅 언어입니다. 일반 Java 코드는 유효한 Groovy 코드이기도 합니다.

플러그인

일반 데이터베이스 어댑터는 다음 플러그인을 지원합니다.

- 전체 토폴로지 동기화용 플러그인(선택 사항)
- 토폴로지의 변경 내용 동기화용 플러그인(선택 사항). 변경 내용 동기화용 플러그인을 구현하지 않을 경우 차등 동기화를 수행할 수 있지만 이 동기화는 사실상 전체 동기화가 아닙니다.
- 레이아웃 동기화용 플러그인(선택 사항)
- 동기화용으로 지원되는 쿼리를 검색하는 플러그인(선택 사항). 이 플러그인을 정의하지 않는 경우 모든 TQL 이름이 반환됩니다.
- TQL 정의 및 TQL 결과를 변경하는 내부 플러그인(선택 사항)
- 레이아웃 요청 및 CI 결과를 변경하는 내부 플러그인(선택 사항)
- 레이아웃 요청 및 관계 결과를 변경하는 내부 플러그인(선택 사항)
- ID 다시 밀어넣기의 수행을 변경하는 내부 플러그인(선택 사항)

플러그인 구현 및 배포에 대한 자세한 내용은 "플러그인 구현"(131페이지)을 참조하십시오.

구성의 예

이 섹션에서는 구성의 예를 제공합니다.

이 섹션에는 다음 항목이 포함됩니다.

사용 사례

TQL 쿼리는 다음과 같습니다.

node > (composition) > card

여기서 각 항목은 다음과 같습니다.

- **node**: CMDB 엔터티
- **card**: 연합 데이터베이스 원본 엔터티
- **composition**: 두 엔터티 사이의 관계

이 예는 ED 데이터베이스에 대해 실행됩니다. ED node는 Device 테이블에 저장되고 card는 hwCards 테이블에 저장됩니다. 다음에 이어지는 예에서도 card는 항상 같은 방식으로 매핑됩니다.

단일 노드 조정

이 예에서 조정은 name 속성에 대해 실행됩니다.

간단 정의

조정은 node별로 수행되고 **CMDB-class**라는 특수 태그로 강조 표시됩니다.

```
<?xml version="1.0" encoding="UTF-8"?>
<generic-DB-adapter-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../META-CONF/simplifiedConfiguration.xsd">
  <CMDB-class CMDB-class-name="node" default-table-name="Device">
    <primary-key column-name="Device_ID"/>
    <reconciliation-by-single-node>
      <또는>
        <attribute CMDB-attribute-name="name" column-name="Device_Name" />
      </or>
    </reconciliation-by-single-node>
  </CMDB-class>
  <class CMDB-class-name="card" default-table-name="hwCards" connected-CMDB-class-
name="node" link-class-name="composition">
    <foreign-primary-key column-name="Device_ID" CMDB-class-primary-key-column="Device_ID
    <primary-key column-name="hwCards_Seq" />
    <attribute CMDB-attribute-name="card_class" column-name="hwCardClass" />
    <attribute CMDB-attribute-name="card_vendor" column-name="hwCardVendor" />
    <attribute CMDB-attribute-name="card_name" column-name="hwCardName" />
  </class>
</generic-DB-adapter-config>
```

고급 정의

orm.xml 파일

관계 매핑을 추가하는 작업을 살펴보십시오. 자세한 내용은 "orm.xml 파일"(146페이지)의 정의 섹션을 참조하십시오.

orm.xml 파일의 예:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/
persistence/orm http://java.sun.com/xml/ns/persistence/orm_1_0.xsd" version="1.0">
    <description>Generic DB adapter orm</description>
    <package>generic_db_adapter</package>
    <entity class="generic_db_adapter.node" >
        <table name="Device"/>
        <attributes>
            <id name="id1">
                <column name="Device_ID"
                  insertable="false"
                  updatable="false"/>
                <generated-value strategy="TABLE"/>
            </id>
            <basic name="name">
                <column name="Device_Name"/>
            </basic>
        </attributes>
    </entity>
    <entity class="generic_db_adapter.card" >
        <table name="hwCards"/>
        <attributes>
            <id name="id1">
                <column name="hwCards_Seq" insertable="false"
                   updatable="false"/>
                <generated-value strategy="TABLE"/>
            </id>
            <basic name="card_class">
                <column name="hwCardClass" insertable="false"
                   updatable="false"/>
            </basic>
            <basic name="card_vendor">
                <column name="hwCardVendor" insertable="false"
                  updatable="false"/>
            </basic>
            <basic name="card_name">
                <column name="hwCardName" insertable="false"
                  updatable="false"/>
            </basic>
        </attributes>
    </entity>
    <entity class="generic_db_adapter.node_composition_card" >
        <table name="hwCards"/>
        <attributes>
            <id name="id1">
                <column name="hwCards_Seq" insertable="false"
                  updatable="false"/>
```

```
                <generated-value strategy="TABLE"/>
            </id>
            <many-to-one name="end1" target-entity="node">
                <join-column name="Device_ID" insertable="false"
                  updatable="false"/>
            </many-to-one>
            <one-to-one name="end2" target-entity="card"
>                 <join-column name="hwCards_Seq"
                referenced-column-name="hwCards_Seq" insertable=
                "false" updatable="false"/>
            </one-to-one>
        </attributes>
    </entity>
</entity-mappings>
```

reconciliation_rules.txt 파일

자세한 내용은 "reconciliation_rules.txt 파일(이전 버전과의 호환용)"(158페이지)**을 참조하십시오.**

multinode[node] expression[node.name]

transformation.txt 파일

이 예제에서는 값을 변환할 필요가 없으므로 이 파일은 계속 비어 있습니다.

두 노드 조정

이 예에서 조정은 서로 다른 변형을 가지고 있는 node **및** ip_address**의** name **속성에 따라 계산됩니다.**

조정 TQL 쿼리는 node > (containment) > ip_address입니다.

간단 정의

node **또는** ip_address**의** name**에 따라 조정됩니다.**

```
<?xml version="1.0" encoding="UTF-8"?>
<generic-DB-adapter-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../META-CONF/simplifiedConfiguration.xsd">
  <CMDB-class CMDB-class-name="node" default-table-name="Device">
    <primary-key column-name="Device_ID"/>
    <reconciliation-by-two-nodes connected-node-CMDB-class-name="ip_address" CMDB-link-
type="containment">
      <또는>
        <attribute CMDB-attribute-name="name" column-name="Device_Name" />
        <connected-node-attribute CMDB-attribute-name="name" column-name="Device_
PreferredIPAddress" />
      </or>
    </reconciliation-by-two-nodes>
  </CMDB-class>
```

```
  <class CMDB-class-name="card" default-table-name="hwCards" connected-CMDB-class-
name="node" link-class-name="containment">
    <foreign-primary-key column-name="Device_ID" CMDB-class-primary-key-column="Device_ID"
/>
    <primary-key column-name="hwCards_Seq" />
    <attribute CMDB-attribute-name="card_class" column-name="hwCardClass" />
    <attribute CMDB-attribute-name="card_vendor" column-name="hwCardVendor" />
    <attribute CMDB-attribute-name="card_name" column-name="hwCardName" />
  </class>
</generic-DB-adapter-config>
```

node 및 ip_address의 name에 따라 조정됩니다.

```
<?xml version="1.0" encoding="UTF-8"?>
<generic-DB-adapter-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../META-CONF/simplifiedConfiguration.xsd">
  <CMDB-class CMDB-class-name="node" default-table-name="Device">
    <primary-key column-name="Device_ID"/>
    <reconciliation-by-two-nodes connected-node-CMDB-class-name="ip_address" CMDB-link-
type="containment">
      <and>
        <attribute CMDB-attribute-name="name" column-name="Device_Name" />
        <connected-node-attribute CMDB-attribute-name="name" column-name="Device_
PreferredIPAddress" />
      </and>
    </reconciliation-by-two-nodes>
  </CMDB-class>
  <class CMDB-class-name="card" default-table-name="hwCards" connected-CMDB-class-
name="node" link-class-name="containment">
    <foreign-primary-key column-name="Device_ID" CMDB-class-primary-key-column="Device_ID"
/>
    <primary-key column-name="hwCards_Seq" />
    <attribute CMDB-attribute-name="card_class" column-name="hwCardClass" />
    <attribute CMDB-attribute-name="card_vendor" column-name="hwCardVendor" />
    <attribute CMDB-attribute-name="card_name" column-name="hwCardName" />
  </class>
</generic-DB-adapter-config>
```

ip_address의 name에 따라 조정됩니다.

```
<?xml version="1.0" encoding="UTF-8"?>
<generic-DB-adapter-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../META-CONF/simplifiedConfiguration.xsd">
  <CMDB-class CMDB-class-name="node" default-table-name="Device">
    <primary-key column-name="Device_ID"/>
    <reconciliation-by-two-nodes connected-node-CMDB-class-name="ip_address" CMDB-link-
```

```
type="containment">
       <또는>
         <connected-node-attribute CMDB-attribute-name="name" column-name="Device_
PreferredIPAddress" />
       </or>
     </reconciliation-by-two-nodes>
   </CMDB-class>
   <class CMDB-class-name="card" default-table-name="hwCards" connected-CMDB-class-
name="node" link-class-name="containment">
     <foreign-primary-key column-name="Device_ID" CMDB-class-primary-key-column="Device_ID"
/>
     <primary-key column-name="hwCards_Seq" />
     <attribute CMDB-attribute-name="card_class" column-name="hwCardClass" />
     <attribute CMDB-attribute-name="card_vendor" column-name="hwCardVendor" />
     <attribute CMDB-attribute-name="card_name" column-name="hwCardName" />
   </class>
</generic-DB-adapter-config>
```

고급 정의

orm.xml 파일

이 파일에 조정 식이 정의되어 있지 않으므로 모든 조정 식에 같은 버전을 사용해야 합니다.

reconciliation_rules.txt 파일

자세한 내용은 "reconciliation_rules.txt 파일(이전 버전과의 호환용)"(158페이지)**을 참조하십시오.**

```
multinode[node] expression[ip_address.name OR node.name] end1_type[node] end2_type[ip_
address] link_type[containment]

multinode[node] expression[ip_address.name AND node.name] end1_type[node] end2_type[ip_
address] link_type[containment]

multinode[node] expression[ip_address.name] end1_type[node] end2_type[ip_address] link_type
[containment]
```

transformation.txt 파일

이 예제에서는 값을 변환할 필요가 없으므로 이 파일은 계속 비어 있습니다.

두 개 이상의 열이 포함된 기본 키 사용

기본 키가 두 개 이상의 열로 구성된 경우 XML 정의에 다음 코드가 추가됩니다.

간단 정의

기본 키 태그가 두 개 이상 있고 각 열에 하나의 태그가 있습니다.

```
   <class CMDB-class-name="card" default-table-name="hwCards" connected-CMDB-class-
```

```
name="node" link-class-name="containment">
     <foreign-primary-key column-name="Device_ID" CMDB-class-primary-key-column="Device_ID"
/>
     <primary-key column-name="Device_ID"/>
     <primary-key column-name="hwBusesSupported_Seq" />
     <primary-key column-name="hwCards_Seq" />
     <attribute CMDB-attribute-name="card_class" column-name="hwCardClass" />
     <attribute CMDB-attribute-name="card_vendor" column-name="hwCardVendor" />
     <attribute CMDB-attribute-name="card_name" column-name="hwCardName" />
   </class>
```

고급 정의

orm.xml 파일

기본 키 열에 매핑되는 새 id 엔터티가 추가됩니다. 이 id 엔터티를 사용하는 엔터티는 특수 태그를 추가해야 합니다.

그러한 기본 키에 외래 키(join-column 태그)를 사용하는 경우 외래 키의 각 열을 기본 키의 열에 매핑해야 합니다.

자세한 내용은 "orm.xml 파일"(146페이지)을 참조하십시오.

orm.xml 파일의 예:

```
<entity class="generic_db_adapter.card" >
     <table name="hwCards"/>
     <attributes>
       <id name="id1">
         <column name="Device_ID" insertable="false" updatable="false"/>
         <generated-value strategy="TABLE"/>
       </id>
       <id name="id2">
         <column name="hwBusesSupported_Seq" insertable="false" updatable="false"/>
         <generated-value strategy="TABLE"/>
       </id>
       <id name="id3">
         <column name="hwCards_Seq" insertable="false" updatable="false"/>
         <generated-value strategy="TABLE"/>
       </id>

<entity class="generic_db_adapter.node_containment_card" >
     <table name="hwCards"/>
     <attributes>
       <id name="id1">
         <column name="Device_ID" insertable="false" updatable="false"/>
         <generated-value strategy="TABLE"/>
       </id>
       <id name="id2">
         <column name="hwBusesSupported_Seq" insertable="false" updatable="false"/>
```

```
          <generated-value strategy="TABLE"/>
        </id>
        <id name="id3">
          <column name="hwCards_Seq" insertable="false" updatable="false"/>
          <generated-value strategy="TABLE"/>
        </id>
        <many-to-one name="end1" target-entity="node">
          <join-column name="Device_ID" insertable="false" updatable="false"/>
        </many-to-one>
        <one-to-one name="end2" target-entity="card">
          <join-column name="Device_ID" referenced-column-name="Device_ID" insertable="false"
updatable="false"/>
          <join-column name="hwBusesSupported_Seq" referenced-column-
name="hwBusesSupported_Seq" insertable="false" updatable="false"/>
          <join-column name="hwCards_Seq" referenced-column-name="hwCards_Seq"
insertable="false" updatable="false"/>
        </one-to-one>
      </attributes>
    </entity>
</entity-mappings>
```

변환 사용

다음 예에서는 일반 enum 변환기가 name **열의** 1, 2, 3 **값에서 각각** a, b, c **값으로 변환됩니다.**

매핑 파일은 generic-enum-transformer-example.xml**입니다.**

```
<enum-transformer CMDB-type="string" DB-type="string" non-existing-value-action="return-original"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../META-
CONF/generic-enum-transformer.xsd">
  <value CMDB-value="1" external-DB-value="a" />
  <value CMDB-value="2" external-DB-value="b" />
  <value CMDB-value="3" external-DB-value="c" />
</enum-transformer>
```

간단 정의

```
  <CMDB-class CMDB-class-name="node" default-table-name="Device">
    <primary-key column-name="Device_ID"/>
    <reconciliation-by-two-nodes connected-node-CMDB-class-name="ip_address"
     CMDB-link-type="containment">
      <또는>
        <attribute CMDB-attribute-name="name" column-name="Device_Name"
         from-CMDB-converter="com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.
         transform.impl.GenericEnumTransformer(generic-enum-transformer-example.
         xml)" to-CMDB-converter="com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.
         transform.impl.GenericEnumTransformer(generic-enum-transformer-example.
         xml)" />
        <connected-node-attribute CMDB-attribute-name="name"
```

```
        column-name="Device_PreferredIPAddress" />
      </or>
    </reconciliation-by-two-nodes>
  </CMDB-class>
```

고급 정의

transformation.txt 파일에만 변경 내용이 있습니다.

transformation.txt 파일

특성 이름 및 엔터티 이름이 orm.xml 파일의 해당 이름과 같아야 합니다.

```
entity[node] attribute[name]
to_DB_class[com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.
GenericEnumTransformer(generic-enum-transformer-example.xml)] from_DB_class
[com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.
GenericEnumTransformer(generic-enum-transformer-example.xml)]
```

어댑터 로그 파일

계산 흐름 및 어댑터 수명 주기를 이해하고 디버그 정보를 보려면 다음 로그 파일을 참조하면 됩니다.

이 섹션에는 다음 항목이 포함됩니다.

로그 수준

각 로그에 대해 로그 수준을 구성할 수 있습니다.

텍스트 편집기에서 **C:\hp\UCMDB\UCMDBServer\conf\log\fcmdb.gdba.properties** 파일을 엽니다.

기본 로그 수준은 **ERROR**입니다.

```
#loglevel can be any of DEBUG INFO WARN ERROR FATAL
loglevel=ERROR
```

- 모든 로그 파일의 로그 수준을 올리려면 **loglevel=ERROR**를 **loglevel=DEBUG** 또는 **loglevel=INFO**로 변경하십시오.
- 특정 파일의 로그 수준을 변경하려면 특정 **log4j** 범주 줄도 적절히 변경하십시오. 예를 들어 fcmdb.gdba.dal.sql.log의 로그 수준을 **INFO**로 변경하려면 다음 항목을

  ```
  log4j.category.fcmdb.gdba.dal.SQL=${loglevel},fcmdb.gdba.dal.SQL.appender
  ```

  다음과 같이 변경합니다.

  ```
  log4j.category.fcmdb.gdba.dal.SQL=INFO,fcmdb.gdba.dal.SQL.appender
  ```

로그 위치

로그 파일은 **C:\hp\UCMDB\UCMDBServer\runtime\log** 디렉터리에 있습니다.

- **Fcmdb.gdba.log**

  어댑터 수명 주기 로그입니다. 어댑터가 시작한 시간 또는 중지한 시간 및 이 어댑터에서 지원하는 CIT에 대한 세부 정보를 제공합니다.

  초기화 오류(어댑터 로드/언로드)를 참조하십시오.

- **fcmdb.log**

  예외를 참조하십시오.

- **cmdb.log**

  예외를 참조하십시오.

- **Fcmdb.gdba.mapping.engine.log**

  매핑 엔진 로그입니다. 매핑 엔진에서 사용하는 조정 TQL 쿼리 및 연결 단계에서 비교되는 조정 토폴로지에 대한 세부 정보를 제공합니다.

  데이터베이스에 관련 CI가 있는데도 TQL 쿼리가 결과를 반환하지 않거나 예기치 못한 결과를 제공하는 경우 이 로그를 참조하십시오(조정 확인).

- **Fcmdb.gdba.TQL.log**

  TQL 로그입니다. TQL 쿼리 및 그 결과에 대한 세부 정보를 제공합니다.

  TQL 쿼리가 결과를 반환하지 않는 경우 그리고 매핑 엔진 로그에 연합 데이터 원본에 결과가 없다고 표시될 경우 이 로그를 참조하십시오.

- **Fcmdb.gdba.dal.log**

  DAL 수명 주기 로그입니다. CIT 생성 및 데이터베이스 연결 정보에 대한 세부 정보를 제공합니다.

  데이터베이스에 연결할 수 없거나, 쿼리에서 지원하지 않는 CIT 또는 특성이 있을 때는 이 로그를 참조하십시오.

- **Fcmdb.gdba.dal.command.log**

  DAL 작업 로그입니다. 호출한 내부 DAL 작업에 대한 세부 정보를 제공합니다. 이 로그는 cmdb.dal.command.log와 유사합니다.

- **Fcmdb.gdba.dal.SQL.log**

  DAL SQL 쿼리 로그입니다. 호출한 JPAQL(개체 지향 SQL 쿼리) 및 그 결과에 대한 세부 정보를 제공합니다.

  데이터베이스에 연결할 수 없거나, 쿼리에서 지원하지 않는 CIT 또는 특성이 있을 때는 이 로그를 참조하십시오.

- **Fcmdb.gdba.hibrnate.log**

  Hibernate 로그입니다. 실행한 SQL 쿼리, 각 JPAQL을 SQL로 구문 분석한 내용, 쿼리의 결과, Hibernate 캐싱과 관련된 데이터 등에 대한 세부 정보를 제공합니다. Hibernate에 대한 자세한 내용은 "Hibernate를 JPA 공급자로 사용"(114페이지)을 참조하십시오.

외부 참조

JavaBeans 3.0 사양에 대한 세부 사항은
http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html을 참조하십시오.

문제 해결 및 제한 사항 - 일반 데이터베이스 어댑터 개발

이 섹션에서는 일반 데이터베이스 어댑터에 대한 문제 해결 및 제한 사항을 설명합니다.

일반 제한

어댑터 패키지를 업데이트할 때 템플릿 파일을 편집하려면 Microsoft Corporation의 메모장(모든 버전)이 아니라 Notepad++, UltraEdit 또는 기타 타사 텍스트 편집기를 사용합니다. 메모장에서는 특수 기호를 사용할 수 없어 준비한 패키지를 배포하지 못하게 됩니다.

JPA 제한

- 모든 테이블에 기본 키 열이 있어야 합니다.
- CMDB 클래스 특성 이름은 JavaBeans 이름 지정 규칙(예: 이름은 소문자로 시작해야 함)을 따라야 합니다.
- 클래스 모델에서 하나의 관계로 연결된 두 CI는 데이터베이스에 직접 연관을 가지고 있어야 합니다. 예를 들어 node가 ticket에 연결된 경우 이 둘을 연결하는 외래 키나 연결 테이블이 있어야 합니다.
- 같은 CIT에 매핑된 여러 테이블은 같은 기본 키 테이블을 공유해야 합니다.

기능 제한

- CMDB와 연합 CIT 사이에 수동 관계를 만들 수 없습니다. 가상 관계를 정의할 수 있으려면 특수 관계 논리를 정의해야 합니다. 이 논리는 연합 클래스의 속성에 기반할 수 있습니다.
- 연합 CIT는 영향 규칙의 트리거 CIT가 될 수 없습니다. 그러나 영향 분석 TQL 쿼리에는 포함될 수 있습니다.
- 연합 CIT를 엔리치먼트 TQL에 포함할 수는 있지만 엔리치먼트가 수행되는 노드로는 사용할 수 없습니다. 연합 CIT는 추가하거나, 업데이트하거나, 삭제할 수 없습니다.
- 조건에서 클래스 한정자를 사용할 수 없습니다.
- 하위 그래프는 지원되지 않습니다.
- 복합 관계는 지원되지 않습니다.
- 외부 CI CMDBid는 기본 키 특성이 아니라 기본 키로 구성됩니다.
- 바이트 유형의 열은 Microsoft SQL Server에서 기본 키로 사용할 수 없습니다.
- 연합 노드에 정의된 특성 조건의 이름이 **orm.xml** 파일에 매핑되어 있지 않으면 TQL 쿼리가 계산되지

않습니다.

6장: Java 어댑터 개발

이 장의 내용:

연합 프레임워크 개요

참고:

- **관계**라는 용어는 **링크**라는 용어와 같습니다.
- **CI**라는 용어는 **개체**라는 용어와 같습니다.
- 그래프는 노드와 링크의 컬렉션입니다.

연합 프레임워크 기능은 API를 사용하여 연합 원본에서 정보를 검색합니다. 연합 프레임워크의 주요 기능은 다음 세 가지입니다.

- 실시간 **연합**. 모든 쿼리가 원래 데이터 저장소 전체에서 실행되고 결과가 CMDB에서 즉석으로 작성됩니다.
- **채우기.** 외부 데이터 원본에서 CMDB로 데이터(토폴로지 데이터 및 CI 속성)를 채웁니다.
- **데이터 밀어넣기.** 로컬 CMDB에서 원격 데이터 원본으로 데이터(토폴로지 데이터 및 CI 속성)를 밀어넣습니다.

모든 수행 유형에는 각 데이터 저장소에 대한 어댑터가 필요합니다. 그래야 데이터 저장소의 특정 기능을 제공하고 필요한 데이터를 검색 및/또는 업데이트할 수 있습니다. 데이터 저장소에 대한 모든 요청은 해당 어댑터를 통해 이루어집니다.

이 섹션에는 다음 항목도 포함됩니다.

실시간 연합

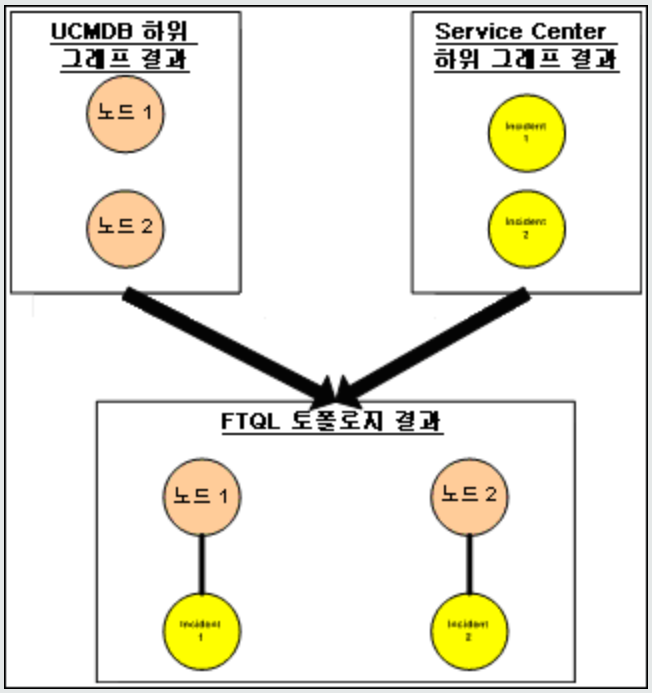
연합 TQL 쿼리를 사용하면 데이터를 복제하지 않고도 외부 데이터 저장소에서 데이터를 검색할 수 있습니다.

연합 TQL 쿼리는 외부 데이터 저장소를 나타내는 어댑터를 사용하여 다른 외부 데이터 저장소의 CI와 UCMDB CI 간에 적절한 외부 관계를 만듭니다.

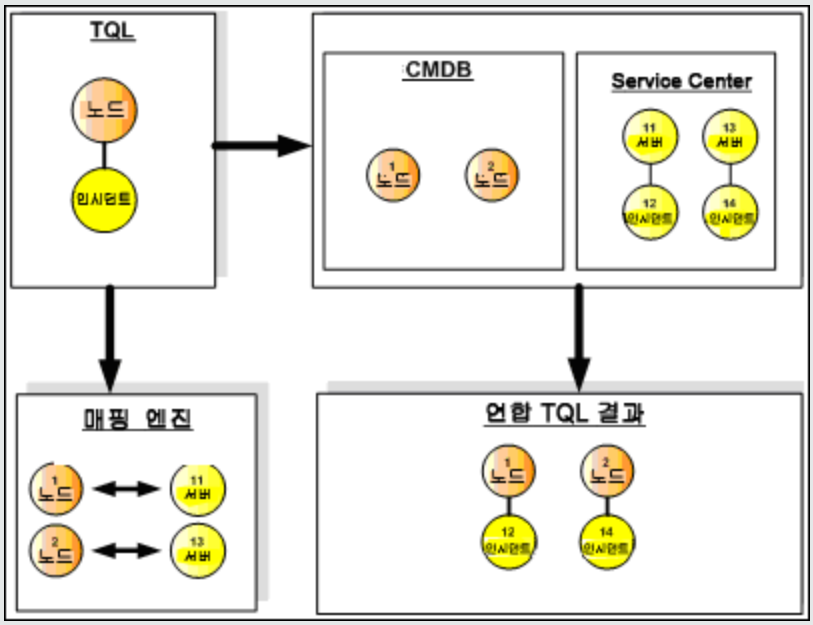
실시간 연합 흐름의 예:

1. **연합 프레임워크는 연합 TQL 쿼리를 몇 개의 하위 그래프로 나눕니다. 여기서 하위 그래프의 모든 노드는 동일한 데이터 저장소를 참조합니다. 각 하위 그래프는 가상 관계에 따라 다른 하위 그래프에 연결됩니다. 그러나 하위 그래프 자체에는 가상 관계가 포함되지 않습니다.**

2. **연합 TQL 쿼리가 하위 그래프로 나눠지면 연합 프레임워크에서 각 하위 그래프의 토폴로지를 계산하고 적합한 노드 간에 가상 관계를 만들어 적합한 하위 그래프 두 개를 연결합니다.**



3. 연합 TQL 토폴로지를 계산한 후 연합 프레임워크에서 토폴로지 결과에 대한 레이아웃을 검색합니다.



데이터 밀어넣기

현재 로컬 CMDB의 데이터를 원격 서비스나 대상 데이터 저장소에 동기화하려면 데이터 밀어넣기 흐름을 사용합니다.

데이터 밀어넣기에서 데이터 저장소는 원본(로컬 CMDB)과 대상이라는 두 개의 범주로 구분됩니다. 데이터는 원본 데이터 저장소에서 검색되고 대상 데이터 저장소로 업데이트됩니다. 데이터 밀어넣기 프로세스는 쿼리 이름을 기준으로 진행됩니다. 즉, 원본(로컬 CMDB) 데이터 저장소와 대상 데이터 저장소 간에 데이터를 동기화한 후 로컬 CMDB에서 TQL 쿼리 이름을 기준으로 데이터를 검색합니다.

데이터 밀어넣기 프로세스 흐름은 다음 단계로 이루어집니다.

1. 원본 데이터 저장소에서 서명이 있는 토폴로지 결과를 검색합니다.
2. 새 결과를 이전 결과와 비교합니다.
3. 변경된 결과에 대해서만 CI 및 관계의 전체 레이아웃(모든 CI 속성)을 검색합니다.
4. 받은 CI 및 관계의 전체 레이아웃을 사용하여 대상 데이터 저장소를 업데이트합니다. 쿼리가 단독 쿼리인 경우 원본 데이터 저장소에서 CI 또는 관계를 삭제하면 복제 프로세스에서 대상 데이터 저장소의 CI 또는 관계도 제거합니다.

CMDB에는 두 개의 숨겨진 데이터 원본(**hiddenRMIDataSource** 및 **hiddenChangesDataSource**)이 있는데, 이러한 원본은 데이터 밀어넣기 흐름에서 항상 `원본' 데이터 원본입니다. 데이터 밀어넣기 흐름에 대해 새 어댑터를 구현하려면 '대상' 어댑터만 구현하면 됩니다.

채우기

CMDB를 외부 원본의 데이터로 채우려면 채우기 흐름을 사용합니다.

이 흐름은 항상 '원본' 데이터 원본 하나를 사용하여 데이터를 검색하고 검색된 데이터를 디스커버리 작업 흐름과 비슷한 프로세스로 프로브에 밀어 넣습니다.

채우기 흐름을 위한 새 어댑터를 구현하려면 원본 어댑터만 구현하면 됩니다. Data Flow Probe가 대상 역할을 하기 때문입니다.

채우기 흐름의 어댑터는 프로브에서 실행됩니다. 디버깅 및 로깅은 CMDB가 아니라 프로브에서 실행해야 합니다.

채우기 흐름은 쿼리 이름을 기준으로 진행됩니다. 즉, 원본 데이터 저장소와 Data Flow Probe 간에 데이터를 동기화한 후 원본 데이터 저장소에서 쿼리 이름을 기준으로 데이터를 검색합니다. 예를 들어 UCMDB에서 쿼리 이름은 TQL 쿼리의 이름입니다. 그러나 다른 데이터 저장소에서는 쿼리 이름이 데이터를 반환하는 코드 이름일 수 있습니다. 어댑터는 쿼리 이름을 올바르게 처리하도록 설계되어 있습니다.

각 작업은 배타적 작업으로 정의할 수 있습니다. 즉, 작업 결과의 CI 및 관계가 로컬 CMDB에서 고유하며 다른 쿼리가 이들을 대상으로 가져올 수 없음을 의미합니다. 원본 데이터 저장소의 어댑터는 특정 쿼리를 지원하며 이 데이터 저장소에서 데이터를 검색할 수 있습니다. 대상 데이터 저장소의 어댑터를 사용하면 이 데이터 저장소에서 검색된 데이터를 업데이트할 수 있습니다.

SourceDataAdapter 흐름

- 원본 데이터 저장소에서 서명이 있는 토폴로지 결과를 검색합니다.
- 새 결과를 이전 결과와 비교합니다.
- 변경된 결과에 대해서만 CI 및 관계의 전체 레이아웃(모든 CI 속성)을 검색합니다.
- 받은 CI 및 관계의 전체 레이아웃을 사용하여 대상 데이터 저장소를 업데이트합니다. 쿼리가 단독 쿼

리인 경우 원본 데이터 저장소에서 CI 또는 관계를 삭제하면 복제 프로세스에서 대상 데이터 저장소의 CI 또는 관계도 제거합니다.

SourceChangesDataAdapter 흐름

- 주어진 마지막 날짜 이후에 발생한 토폴로지 결과를 검색합니다.
- 변경된 결과에 대해서만 CI 및 관계의 전체 레이아웃(모든 CI 속성)을 검색합니다.
- 받은 CI 및 관계의 전체 레이아웃을 사용하여 대상 데이터 저장소를 업데이트합니다. 쿼리가 단독 쿼리인 경우 원본 데이터 저장소에서 CI 또는 관계를 삭제하면 복제 프로세스에서 대상 데이터 저장소의 CI 또는 관계도 제거합니다.

PopulateDataAdapter 흐름

- 요청된 레이아웃 결과가 있는 전체 토폴로지를 검색합니다.
- 토폴로지 청크 메커니즘을 사용하여 청크의 데이터를 검색합니다.
- 이전 실행에서 이미 가져온 모든 데이터를 프로브에서 필터링합니다.
- 받은 CI 및 관계의 레이아웃을 사용하여 대상 데이터 저장소를 업데이트합니다. 쿼리가 단독 쿼리인 경우 원본 데이터 저장소에서 CI 또는 관계를 삭제하면 복제 프로세스에서 대상 데이터 저장소의 CI 또는 관계도 제거합니다.

PopulateChangesDataAdapter 흐름

- 마지막 실행 이후에 변경되었고 요청된 레이아웃 결과가 있는 토폴로지를 검색합니다.
- 토폴로지 청크 메커니즘을 사용하여 청크의 데이터를 검색합니다.
- 이전 실행(이 흐름 포함)에서 이미 가져온 데이터를 프로브에서 필터링합니다.
- 받은 CI 및 관계의 레이아웃을 사용하여 대상 데이터 저장소를 업데이트합니다. 쿼리가 단독 쿼리인 경우 원본 데이터 저장소에서 CI 또는 관계를 삭제하면 복제 프로세스에서 대상 데이터 저장소의 CI 또는 관계도 제거합니다.

인스턴스 기반 채우기 흐름

어댑터가 "XML 구성 태그 및 속성"(207페이지)에 설명된 **<instance-based-data>** 태그를 통해 인스턴스 기반 흐름을 지원하도록 정의되어 있는 경우 채우기 엔진은 자동으로 인스턴스 내에서 제거된 CI를 찾은 다음 UCMDB에서 제거합니다(특정 채우기 작업에 대해 삭제가 허용되는 경우). 각 인스턴스에는 TQL 정의에서 **Root** 이름으로 표시된 루트 CI가 있어야 합니다. 루트 CI가 전달될 때마다 전체 인스턴스(루트 CI에 연결된 모든 CI)를 마지막으로 UCMDB로 보냈을 때와 비교하여 이전에 루트에 연결되어 있었지만 지금은 연결되지 않은 CI를 UCMDB에서 삭제합니다. 어댑터가 인스턴스 기반 흐름을 올바르게 지원하려면 전체 인스턴스의 CI 또는 특성에 변경 내용이 있을 경우 전체 인스턴스를 UCMDB로 다시 보내야 합니다.

연합 프레임워크와의 어댑터 및 매핑 상호 작용

어댑터는 UCMDB의 항목으로, 외부 데이터(UCMDB에 저장되지 않은 데이터)를 나타냅니다. 연합 흐름에서는 외부 데이터 원본과의 모든 상호 작용이 어댑터를 통해 수행됩니다. 연합 프레임워크 상호 작용 흐름 및 어댑터 인터페이스는 복제를 수행할 때와 통합 TQL 쿼리를 수행할 때 서로 다릅니다.

이 섹션에는 다음 항목도 포함됩니다.

어댑터 수명 주기

각 외부 데이터 저장소마다 하나의 어댑터 인스턴스가 만들어집니다. 어댑터의 수명 주기는 이 어댑터에 적용되는 첫 번째 수행(예: TQL 계산 또는 데이터 검색/업데이트)으로 시작됩니다. start 메서드가 호출되면 어댑터가 환경 정보(예: 데이터 저장소 구성, 로거 등)를 받습니다. 구성에서 데이터 저장소가 제거되면 어댑터 수명 주기가 종료되고 shutdown 메서드가 호출됩니다. 이는 어댑터가 상태 기반이며 필요한 경우 외부 데이터 저장소에 대한 연결을 포함할 수 있음을 의미합니다.

어댑터 assist 메서드

어댑터에는 외부 데이터 저장소 구성을 추가할 수 있는 assist 메서드가 몇 가지 있습니다. 이러한 메서드는 어댑터 수명 주기에 포함되지 않으며 호출할 때마다 새 어댑터를 만듭니다.

- **첫 번째 메서드는 주어진 구성의 외부 데이터 저장소에 대한 연결을 테스트합니다. testConnection은 어댑터 유형에 따라 UCMDB 서버 또는 Data Flow Probe에서 실행할 수 있습니다.**
- **두 번째 메서드는 원본 어댑터에만 관련이 있으며 복제를 위해 지원되는 쿼리를 반환합니다. 이 메서드는 프로브에서만 실행됩니다.**
- **세 번째 메서드는 연합 흐름과 채우기 흐름에만 관련이 있으며 외부 데이터 저장소에서 지원되는 외부 클래스를 반환합니다. 이 메서드는 UCMDB 서버에서 실행됩니다.**

이 모든 메서드는 통합 구성을 만들거나 볼 때 사용됩니다.

통합 TQL 쿼리에 대한 연합 프레임워크

이 섹션에는 다음 항목이 포함됩니다.

연합 프레임워크, UCMDB, 어댑터, 매핑 엔진 간의 상호 작용을 보여주는 다이어그램은 "연합 프레임워크, 서버, 어댑터 및 매핑 엔진 간의 상호 작용"(188페이지)을 참조하십시오.

정의 및 용어

조정 데이터. CMDB 및 외부 데이터 저장소에서 받은 지정된 유형의 CI를 일치시키는 규칙입니다. 조정 규칙에는 다음 세 가지 유형이 있습니다.

- **ID 조정.** 이 규칙은 외부 데이터 저장소에 조정 개체의 CMDB ID가 포함된 경우에만 사용할 수 있습니다.
- **속성 조정.** 이 규칙은 조정 CI 유형의 속성으로 일치를 수행할 수 있는 경우에만 사용됩니다.
- **토폴로지 조정.** 이 규칙은 조정 CI에서 일치를 수행하는 데 추가 CIT(조정 CIT 외에도 포함)의 속성이 필

요한 경우에 사용됩니다. 예를 들어 ip_address CIT에 속하는 name 속성으로 노드 유형의 조정을 수행할 수 있습니다.

조정 개체. 이 개체는 받은 조정 데이터에 따라 어댑터에서 만들어집니다. 이 개체는 외부 CI를 참조해야 하며 매핑 엔진에서 외부 CI와 CMDB CI 사이를 연결하는 데 사용됩니다.

조정 CI 유형. 조정 개체를 나타내는 CI의 유형입니다. 이러한 CI는 CMDB와 외부 데이터 저장소 두 곳에 모두 저장해야 합니다.

매핑 엔진. 서로 다른 데이터 저장소에서 받았으나 서로 간에 가상 관계를 갖고 있는 CI 사이의 관계를 식별하는 구성 요소입니다. 식별 작업은 CMDB 조정 개체와 외부 CI 조정 개체를 조정하는 방식으로 수행됩니다.

매핑 엔진

연합 프레임워크는 매핑 엔진을 사용하여 통합 TQL 쿼리를 계산합니다. 매핑 엔진은 서로 다른 데이터 저장소에서 받았으나 가상 관계로 연결되어 있는 CI 사이를 연결합니다. 매핑 엔진은 가상 관계에도 조정 데이터를 제공합니다. 가상 관계의 한끝은 CMDB를 참조해야 합니다. 이 끝이 조정 유형입니다. 두 하위 그래프를 계산하려는 경우 가상 관계는 어느 쪽 끝 노드에서나 시작할 수 있습니다.

연합 어댑터

연합 어댑터는 외부 데이터 저장소에서 외부 CI 데이터와 외부 CI에 속한 조정 개체 등 두 가지 종류의 데이터를 가져옵니다.

- **외부 CI 데이터.** CMDB에 없는 외부 데이터입니다. 이 데이터는 외부 데이터 저장소의 대상 데이터입니다.
- **조정 개체 데이터.** 연합 프레임워크에서 CMDB CI와 외부 데이터를 연결하는 데 사용되는 보조 데이터입니다. 각 조정 개체는 외부 CI를 참조해야 합니다. 조정 개체의 유형은 데이터가 검색되는 가상 관계 한끝의 유형(또는 하위 유형)입니다. 조정 개체는 받은 어댑터를 조정 데이터에 맞춰야 합니다. 조정 개체는 IdReconciliationObject, PropertyReconciliationObject, TopologyReconciliationObject 등 세 가지 유형 중 하나일 수 있습니다.

DataAdapter 기반 인터페이스(DataAdapter, PopulateDataAdapter, PopulateChangesDataAdapter)에서는 쿼리 정의의 일부로 조정을 요청합니다.

연합 프레임워크, 서버, 어댑터 및 매핑 엔진 간의 상호 작용

다음 다이어그램은 연합 프레임워크, UCMDB, 어댑터, 매핑 엔진 간의 상호 작용을 나타냅니다. 예제 다이어그램에서 통합 TQL 쿼리에는 가상 관계가 하나뿐이므로 UCMDB과 외부 데이터 저장소 하나만 이 통합 TQL 쿼리에 관여됩니다.

이 섹션에는 다음 항목이 포함됩니다.

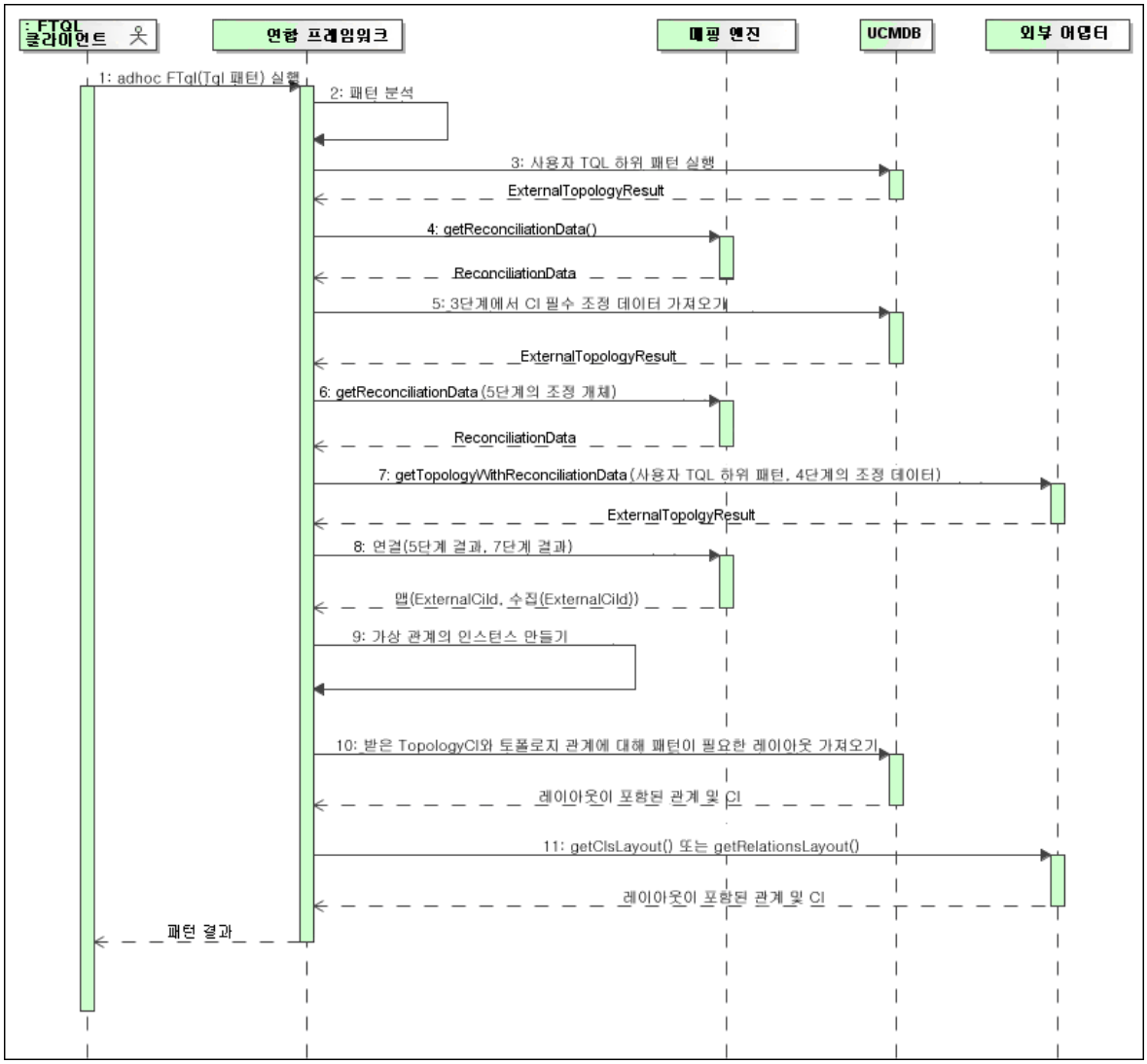
- " 서버 끝에서 계산 시작"(189페이지)
- "외부 어댑터 끝에서 계산 시작"(191페이지)

- "통합 TQL 쿼리에 대한 연합 프레임워크 흐름의 예"(192페이지)

첫 번째 다이어그램에서는 UCMDB에서 계산이 시작되고 두 번째 다이어그램에서는 외부 어댑터에서 계산이 시작됩니다. 다이어그램의 각 단계에는 어댑터나 매핑 엔진 인터페이스의 적절한 메서드 호출에 대한 참조가 포함되어 있습니다.

서버 끝에서 계산 시작

다음 시퀀스 다이어그램은 연합 프레임워크, UCMDB, 어댑터, 매핑 엔진 간의 상호 작용을 나타냅니다. 예제 다이어그램에서 통합 TQL 쿼리에는 가상 관계가 하나뿐이므로 UCMDB과 외부 데이터 저장소 하나만이 통합 TQL 쿼리에 관여됩니다.



다음은 이 그림에 표시된 번호에 대한 설명입니다.

번호	설명
1	연합 프레임워크가 통합 TQL 계산에 대한 호출을 받습니다.
2	연합 프레임워크는 어댑터를 분석하고, 가상 관계를 찾고, 원래 TQL을 두 하위 어댑터로 나눕니다. 그 중 하나는 UCMDB용이고 다른 하나는 외부 데이터 저장소용입니다.
3	연합 프레임워크가 UCMDB에서 하위 TQL의 토폴로지를 요청합니다.
4	연합 프레임워크는 토폴로지 결과를 받은 후에 현재 가상 관계에 적절한 매핑 엔진을 호출하고 조정 데이터를 요청합니다. 이 단계에서는 reconciliationObject 매개 변수가 비어 있습니다. 즉, 이 호출에서는 조정 데이터에 추가되는 조건이 없습니다. 반환되는 조정 데이터는 UCMDB의 조정 CI를 외부 데이터 저장소와 일치시키는 데 필요한 데이터를 정의합니다. 조정 데이터는 다음 유형 중 하나일 수 있습니다. • **IdReconciliationData.** CI는 ID에 따라 조정됩니다. • **PropertyReconciliationData.** CI는 CI 중 하나의 속성에 따라 조정됩니다. • **TopologyReconciliationData.** CI는 토폴로지에 따라 조정됩니다. 예를 들어 노드 CI를 조정하려면 **IP**의 IP 주소도 필요합니다.
5	연합 프레임워크가 "3"(190페이지)단계에 UCMDB에서 받은 가상 관계 끝의 CI에 대한 조정 데이터를 요청합니다.
6	연합 프레임워크가 조정 데이터를 검색하기 위해 매핑 엔진을 호출합니다. 이 상태에서는 "3"(190페이지)단계와는 반대로 매핑 엔진이 "5"(190페이지)단계의 조정 개체를 매개 변수로 받습니다. 매핑 엔진은 받은 조정 개체를 조정 데이터에 대한 조건으로 변환합니다.
7	연합 프레임워크가 외부 데이터 저장소에서 하위 TQL의 토폴로지를 요청합니다. 외부 어댑터는 "6"(190페이지)단계의 조정 데이터를 매개 변수로 받습니다.
8	연합 프레임워크는 받은 결과 간을 연결하기 위해 매핑 엔진을 호출합니다. firstResult 매개 변수는 "5"(190페이지)단계에 UCMDB에서 받은 외부 토폴로지 결과이고 secondResult 매개 변수는 "7"(190페이지)단계에 외부 어댑터에서 받은 외부 토폴로지 결과입니다. 매핑 엔진은 첫 번째 데이터 저장소(이 경우에는 UCMDB)의 외부 CI ID를 두 번째(외부) 데이터 저장소의 외부 CI ID에 매핑한 맵을 반환합니다.
9	각 매핑에 대해 연합 프레임워크는 가상 관계를 만듭니다.
10	토폴로지 단계에서만 통합 TQL 쿼리 결과를 계산한 후에, 연합 프레임워크는 해당하는 데이터 저장소에서 결과 CI 및 관계에 대한 원래 TQL 레이아웃을 검색합니다.

외부 어댑터 끝에서 계산 시작

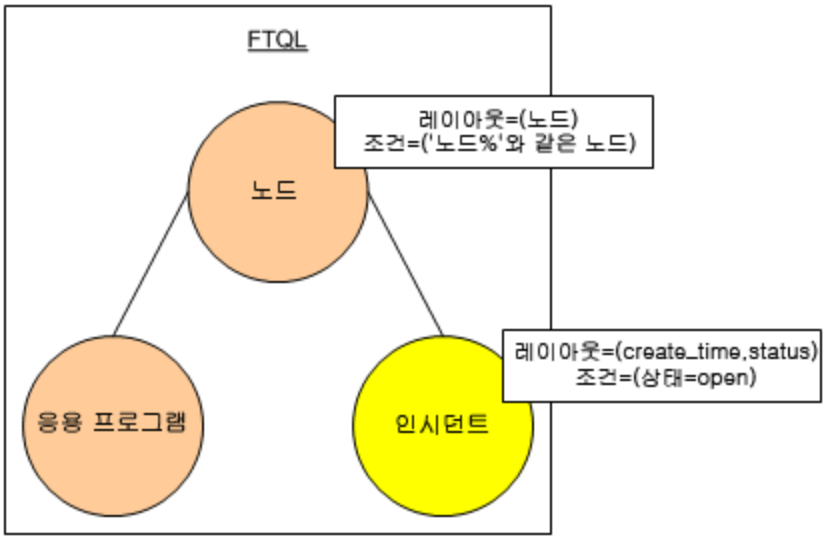
다음은 이 그림에 표시된 번호에 대한 설명입니다.

번호	설명
1	연합 프레임워크가 통합 TQL 계산에 대한 호출을 받습니다.
2	연합 프레임워크는 어댑터를 분석하고, 가상 관계를 찾고, 원래 TQL을 두 하위 어댑터로 나눕니다. 그 중 하나는 UCMDB용이고 다른 하나는 외부 데이터 저장소용입니다.
3	연합 프레임워크가 외부 어댑터에서 하위 TQL의 토폴로지를 요청합니다. 조정 데이터는 요청에 포함되지 않으므로 반환된 ExternalTopologyResult에 조정 개체가 포함되지 않아야 합니다.
4	

번호	설명
	연합 프레임워크는 토폴로지 결과를 받은 후에 현재 가상 관계에 적절한 매핑 엔진을 호출하고 조정 데이터를 요청합니다. 이 상태에서는 reconciliationObjects 매개 변수가 비어 있습니다. 즉, 이 호출에서는 조정 데이터에 추가되는 조건이 없습니다. 반환되는 조정 데이터는 UCMDB의 조정 CI를 외부 데이터 저장소와 일치시키는 데 필요한 데이터를 정의합니다. 조정 데이터는 다음 세 가지 유형 중 하나일 수 있습니다. • **IdReconciliationData.** CI는 ID에 따라 조정됩니다. • **PropertyReconciliationData.** CI는 CI 중 하나의 속성에 따라 조정됩니다. • **TopologyReconciliationData.** CI는 토폴로지에 따라 조정됩니다. 예를 들어 노드 CI를 조정하려면 **IP**의 IP 주소도 필요합니다.
5	연합 프레임워크가 3단계에 외부 데이터 저장소에서 받은 CI에 대한 조정 개체를 요청합니다. 연합 프레임워크가 외부 어댑터에서 **getTopologyWithReconciliationData()** 메서드를 호출합니다. 여기서 요청된 토폴로지는 3단계에 받은 CI가 4단계의 ID 조건 및 조정 데이터로 포함된 단일 노드 토폴로지입니다.
6	연합 프레임워크가 조정 데이터를 검색하기 위해 매핑 엔진을 호출합니다. 이 상태에서는 3단계와는 반대로 매핑 엔진이 5단계의 조정 개체를 매개 변수로 받습니다. 매핑 엔진은 받은 조정 개체를 조정 데이터에 대한 조건으로 변환합니다.
7	연합 프레임워크가UCMDB에서 6단계의 조정 데이터가 포함된 하위 TQL의 토폴로지를 요청합니다.
8	연합 프레임워크는 받은 결과 간을 연결하기 위해 매핑 엔진을 호출합니다. firstResult 매개 변수는 5단계에 외부 어댑터에서 받은 외부 토폴로지 결과이고 secondResult 매개 변수는 7단계에 UCMDB에서 받은 외부 토폴로지 결과입니다. 매핑 엔진은 첫 번째 데이터 저장소(이 경우에서는 외부 데이터 저장소)의 외부 CI ID를 두 번째 데이터 저장소(UCMDB)의 외부 CI ID에 매핑한 맵을 반환합니다.
9	각 매핑에 대해 연합 프레임워크는 가상 관계를 만듭니다.
10	토폴로지 단계에서만 통합 TQL 쿼리 결과를 계산한 후에, 연합 프레임워크는 해당하는 데이터 저장소에서 결과 CI 및 관계에 대한 원래 TQL 레이아웃을 검색합니다.

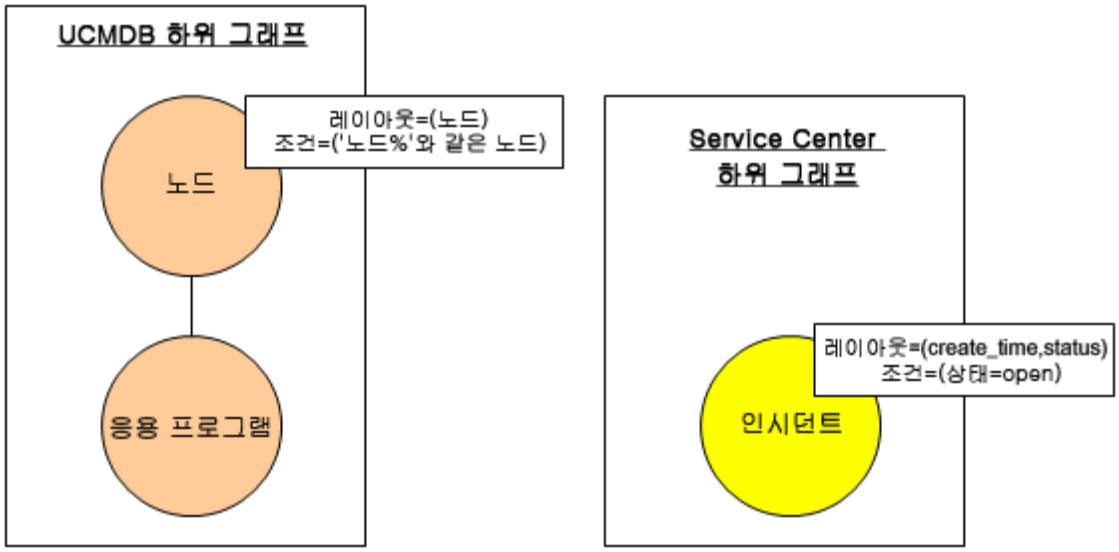
통합 TQL 쿼리에 대한 연합 프레임워크 흐름의 예

이 예에서는 특정 노드에 열려 있는 모든 인시던트를 보는 방법을 설명합니다. ServiceCenter 데이터 저장소는 외부 데이터 저장소입니다. 노드 인스턴스는 UCMDB에 저장되고 인시던트 인스턴스는 ServiceCenter에 저장됩니다. 인시던트 인스턴스를 적절한 노드에 연결하려면 호스트와 IP의 node 및 ip_address 속성이 필요합니다. 이러한 속성은 UCMDB의 ServiceCenter에서 노드를 식별하는 조정 속성입니다.

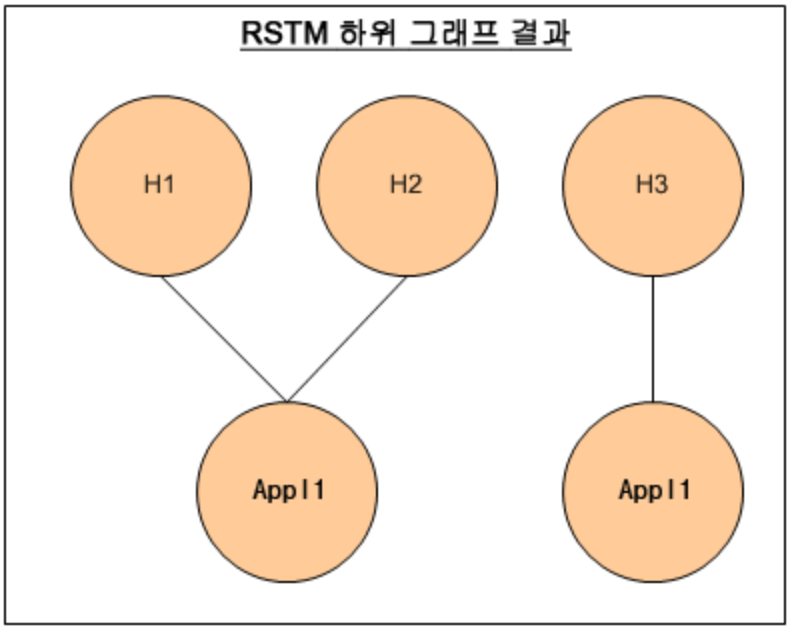


참고: 특성 연합을 위해 어댑터의 **getTopology** 메서드가 호출됩니다. 조정 데이터는 사용자 TQL(이 예에서는 CI 요소)에서 사용됩니다.

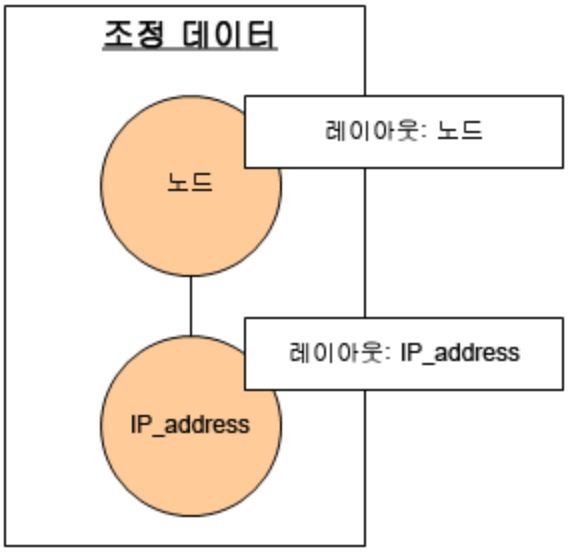
1. 어댑터를 분석한 후 연합 프레임워크는 노드와 인시던트의 가상 관계를 인식하고 통합 TQL 쿼리를 두 개의 하위 그래프로 나눕니다.



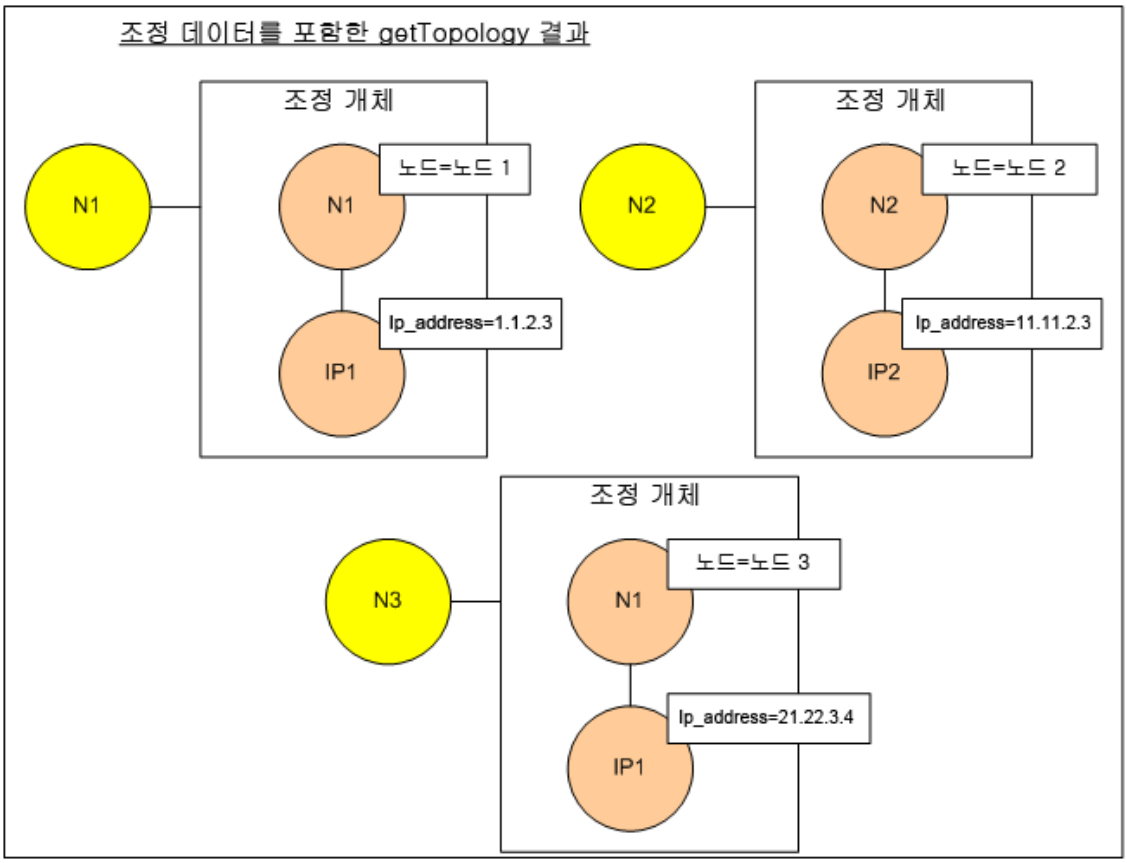
2. 연합 프레임워크는 UCMDB 하위 그래프를 실행하여 토폴로지를 요청하고 다음 결과를 받습니다.



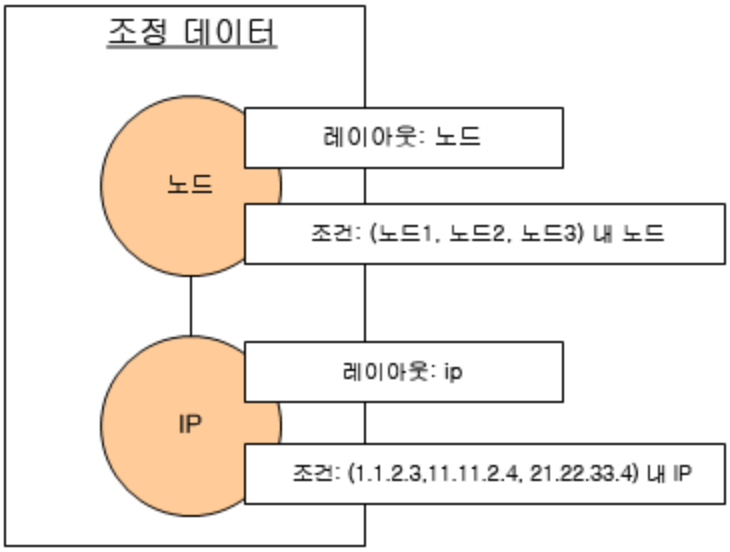
3. 연합 프레임워크는 적절한 매핑 엔진에서 첫 번째 데이터 저장소(UCMDB)에 대한 조정 데이터를 요청합니다. 이 조정 데이터에는 두 데이터 저장소로부터 받은 데이터를 연결하는 데 필요한 정보가 포함되어 있습니다. 이 예에서 조정 데이터는 다음과 같습니다.



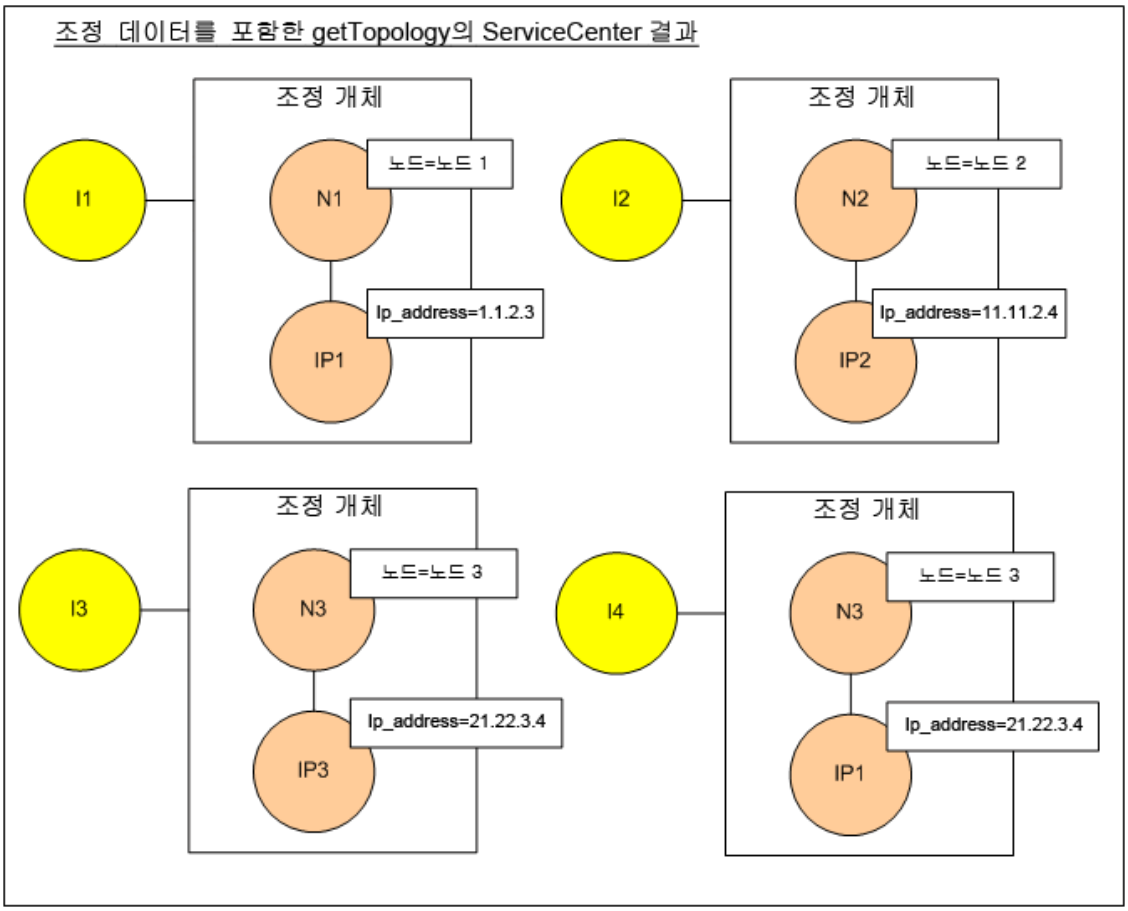
4. 연합 프레임워크는 이전 결과에서 받은 노드 및 ID 조건(H1, H2, H3의 node)이 포함된 단일 노드 토폴로지 쿼리를 만들고, UCMDB에서 필요한 조정 데이터로 이 쿼리를 실행합니다. 결과에는 각 CI의 ID 조건 및 적절한 조정 개체와 관련된 노드 CI가 포함됩니다.



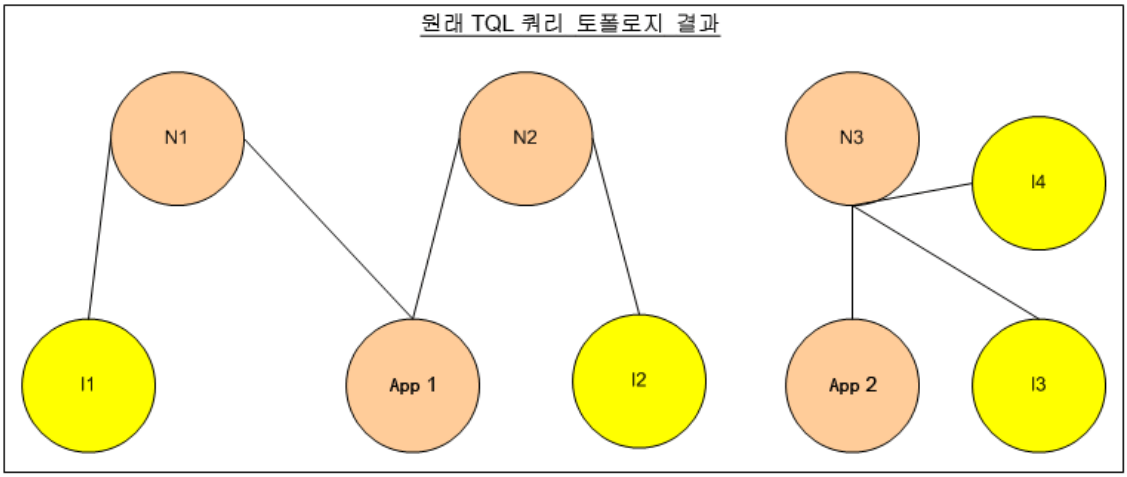
5. ServiceCenter의 조정 데이터는 UCMDB으로부터 받은 조정 개체에서 파생된 node 및 ip에 대한 조건을 포함해야 합니다.



6. 연합 프레임워크는 조정 데이터가 포함된 ServiceCenter 하위 그래프를 실행하여 토폴로지 및 적절한 조정 개체를 요청하고 다음 결과를 받습니다.



7. 매핑 엔진에서 연결을 완료하고 가상 관계를 만든 후의 결과는 다음과 같습니다.



8. 연합 프레임워크가 UCMDB 및 ServiceCenter에서 받은 인스턴스의 원래 TQL 레이아웃을 요청합니다.

채우기에 대한 연합 프레임워크 흐름

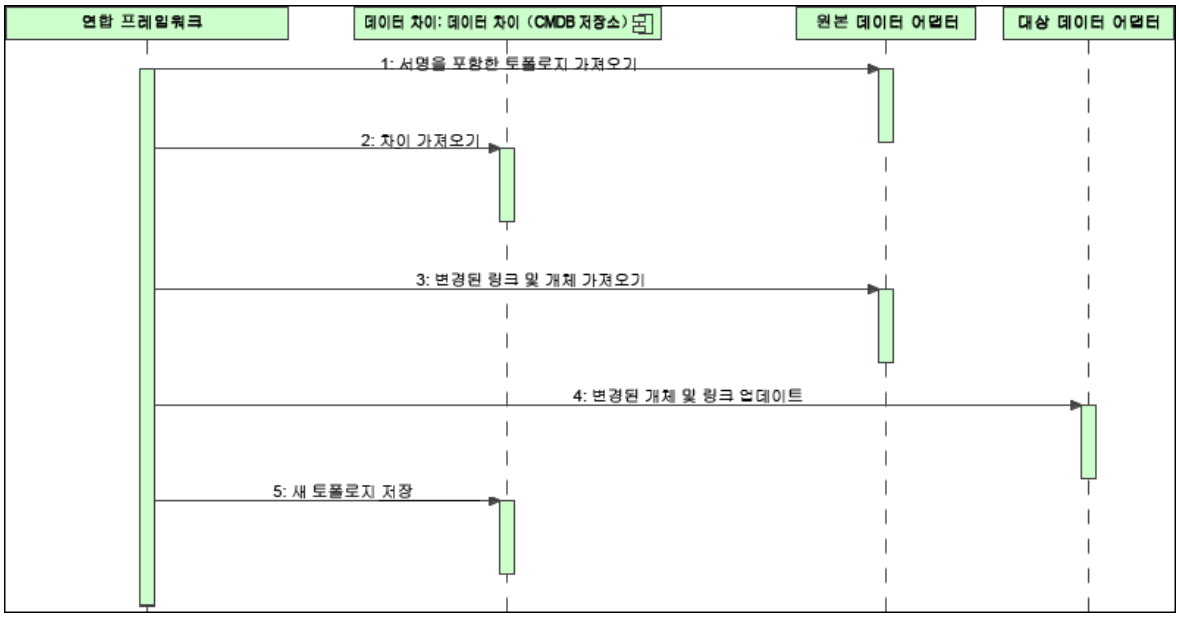
이 섹션에는 다음 항목이 포함됩니다.

정의 및 용어

서명. CI에서 속성의 상태를 표시합니다. CI에서 속성 값을 변경한 경우 CI 서명도 변경해야 합니다. CI 서명은 모든 CI 속성을 검색하여 비교하지 않고도 C 변경 여부를 감지하는 데 도움이 됩니다. CI와 CI 서명은 모두 해당하는 어댑터에서 제공합니다. CI 속성이 변경되면 어댑터에서 CI 서명을 변경해야 합니다.

흐름 다이어그램

다음 시퀀스 다이어그램은 채우기 흐름에서 연합 프레임워크와 원본 및 대상 어댑터 간의 상호 작용을 나타냅니다.



1. 연합 프레임워크가 원본 어댑터에서 쿼리 결과의 토폴로지를 받습니다. 어댑터는 이름을 기준으로 쿼리를 인식하고 외부 데이터 저장소에서 쿼리를 실행합니다. 토폴로지 결과에는 각 CI의 ID 및 서명과 결과의 관계가 포함됩니다. ID는 외부 데이터 저장소에서 CI를 고유하게 정의하는 논리적 ID입니다. CI 또는 관계가 수정되면 서명을 수정해야 합니다.
2. 연합 프레임워크는 서명을 사용하여 새로 받은 토폴로지 쿼리 결과와 저장된 결과를 비교하고 어떤 CI가 변경되었는지 확인합니다.
3. 연합 프레임워크에서 변경된 CI와 관계를 찾은 다음 변경된 CI와 관계의 ID가 있는 원본 어댑터를 매개 변수로 호출하여 전체 레이아웃을 검색합니다.

4. 연합 프레임워크가 업데이트를 대상 어댑터로 보냅니다. 대상 어댑터가 외부 데이터 원본을 받은 데이터로 업데이트합니다.
5. 업데이트 후에 연합 프레임워크에서 마지막 쿼리 결과를 저장합니다.

어댑터 인터페이스

이 섹션에는 다음 항목이 포함됩니다.

정의 및 용어

외부 관계. 동일한 어댑터에서 지원하는 두 외부 CI 유형 간의 관계입니다.

통합 TQL 쿼리 관련 어댑터 인터페이스

다음과 같이 각 어댑터에 해당하는 어댑터 인터페이스를 사용하십시오.

- **단일 노드 토폴로지 인터페이스**는 어댑터가 외부 관계를 지원하지 않는 경우, 즉 어댑터가 둘 이상의 외부 CI가 포함된 요청을 받지 않도록 되어 있는 경우에 사용됩니다. 작업을 완료하는 데 필요한 조정 데이터는 복합 쿼리로 설명할 수 있습니다(아래의 SingleNodeFederationTopologyReconciliationAdapter 참조).

  모든 SingleNode 인터페이스는 워크플로를 단순화하기 위해 만들어졌으므로, 좀 더 광범위한 쿼리를 사용해야 할 경우에는 **FederationTopologyAdapter** 인터페이스를 사용하십시오.
- **FederationTopologyAdapter 인터페이스**는 복잡한 연합 쿼리를 지원하는 어댑터를 정의하는 데 사용됩니다. 이러한 어댑터에서 조정 요청은 **QueryDefinition** 매개 변수의 일부입니다.

  연합 엔진은 연합된 데이터를 적절한 로컬 CI에 연결하기 위해 조정 데이터를 사용합니다. 조정 데이터를 둘 이상의 요청에 가져올 수 있습니다(결과에 따라 재귀적으로 계산됨). 이 경우 어댑터는 조정 데이터만 포함된 요청을 받습니다.

SingleNode 인터페이스

다음 인터페이스는 다른 유형의 조정 데이터를 포함합니다.

- **SingleNodeFederationIdReconciliationAdapter.** 어댑터가 **단일 노드 TQL**을 지원하고 데이터 저장소 간의 조정이 ID를 기준으로 계산되는 경우에 사용합니다.
- **SingleNodeFederationPropertyReconciliationAdapter.** 어댑터가 **단일 노드 TQL**을 지원하고 데이터 저장소 간의 조정이 한 개 CI의 속성을 기준으로 수행되는 경우에 사용합니다.
- **SingleNodeFederationTopologyReconciliationAdapter.** 어댑터가 **단일 노드 TQL**을 지원하고 데이터 저장소 간의 조정이 토폴로지를 기준으로 수행되는 경우에 사용합니다. 어댑터는 쿼리 요소가 비어 있고 조정 토폴로지만 요청되는 경우를 지원해야 합니다.

데이터 어댑터 인터페이스

- **FederationTopologyAdapter.** 복잡한 통합 TQL 쿼리를 지원하려면 이 어댑터를 사용합니다. 대부분의 복잡한 쿼리를 사용할 수 있습니다. 어댑터는 쿼리 정의가 조정 데이터만 설명하는 경우를 지원해야 합니다.
- **PopulateDataAdapter.** 복잡한 통합 TQL 쿼리 및 채우기 흐름을 지원하려면 이 어댑터를 사용합니다. 채우기 흐름에서 이 어댑터는 전체 데이터 집합을 검색하고 프로브를 통해 마지막 작업 실행 이후에 변경된 내용을 필터링할 수 있습니다.
- **PopulateChangesDataAdapter.** 복잡한 통합 TQL 쿼리 및 채우기 흐름을 지원하려면 이 어댑터를 사용합니다. 채우기 흐름에서 이 어댑터는 마지막 작업 실행 이후에 변경된 내용만 검색할 수 있습니다.

참고: 대용량 데이터 집합을 반환하는 어댑터를 개발하는 경우, ChunkGetter 인터페이스를 구현하여 청크를 허용하는 것이 중요합니다. 자세한 내용은 특정 어댑터에 대한 Java 문서를 참조하십시오.

리소스 보고 인터페이스

다음 인터페이스를 통해 어댑터는 어댑터의 동작을 사용자 지정하도록 구성할 수 있는 리소스를 보고할 수 있습니다. 이 경우 통합 스튜디오에서 직접 이러한 리소스를 편집할 수 있습니다. 이러한 인터페이스는 위의 일반 어댑터 인터페이스 외에 추가로 사용해야 합니다.

- **PopulationQueriesResourcesLocator.** 각 특정 채우기 쿼리에 대해 편집할 수 있는 리소스를 정의합니다.
- **PushQueriesResourceLocator.** 각 데이터 밀어넣기 쿼리에 대해 편집할 수 있는 리소스를 정의합니다.
- **GeneralResourcesLocator.** 이 어댑터에서 편집할 수 있는 일반 리소스를 정의합니다.

기타 인터페이스

- **SortResultDataAdapter.** 외부 데이터 저장소에서 결과 CI를 정렬할 수 있는 경우에 사용합니다.
- **FunctionalLayoutDataAdapter.** 외부 데이터 저장소에서 기능 레이아웃을 계산할 수 있는 경우에 사용합니다.

동기화 관련 어댑터 인터페이스

- **SourceDataAdapter.** 채우기 흐름에서 원본 어댑터에 사용합니다.
- **TargetDataAdapter.** 데이터 밀어넣기 흐름에서 대상 어댑터에 사용합니다.

어댑터 리소스 디버그

이 작업에서는 디버깅 및 개발을 목적으로 JMX 콘솔을 사용하여 어댑터 상태 리소스(UCMDB 데이터베이스 또는 프로브 데이터베이스에 저장된 DataAdapterEnvironment 인터페이스의 리소스 조작 메서드를 사용하여 만든 모든 리소스)를 만들고, 보고, 삭제하는 방법에 대해 설명합니다.

1. 웹 브라우저를 시작하고 서버 주소를 다음과 같이 입력합니다.
   - UCMDB 서버: http://localhost:8080/jmx-console
   - 프로브: http://localhost:1977

   사용자 이름과 비밀번호를 입력하여 로그인해야 할 수도 있습니다.
2. JMX MBEAN 보기 페이지를 열려면 다음 중 하나를 클릭합니다.
   - UCMDB 서버:
     UCMDB:service=FCMDB Adapter State Resource Services
   - 프로브: **type=AdapterStateResources**
3. 사용할 작업에 값을 입력하고 **Invoke**를 클릭합니다.

새 외부 데이터 원본에 대해 어댑터 추가

이 작업에서는 새 외부 데이터 원본을 지원할 수 있도록 어댑터를 정의하는 방법을 설명합니다.

이 작업에는 다음 단계가 포함됩니다.

1. 선행 조건

   UCMDB 데이터 모델의 CI 및 관계에 대한 모델 지원 어댑터 클래스가 필요합니다. 어댑터 개발자는 다음 사항을 준비해야 합니다.
   - UCMDB CI 유형의 계층 구조에 대한 지식을 갖추어 외부 CIT와 UCMDB CIT의 관련성 이해
   - UCMDB 클래스 모델에서 외부 CIT 모델링
   - 새로운 CI 유형 및 이들의 관계에 대한 정의 추가
   - 어댑터 내부 클래스 간의 유효한 관계를 위해 UCMDB 클래스 모델에서 유효한 관계 정의 CIT는 UCMDB 클래스 모델 트리의 어떤 수준에나 배치할 수 있습니다.

모델링은 연합 유형(실시간 또는 복제)에 관계없이 동일해야 합니다. UCMDB 클래스 모델에 새 CIT 정의를 추가하는 방법에 대한 자세한 내용은 *HP Universal CMDB 모델링 안내서*에서 CI 선택기 사용을 참조하십시오.

어댑터가 CIT에 대한 연합 특성을 지원할 수 있도록 하려면 이 CIT에 대한 조정 규칙 및 지원되는 특성이 있는 지원되는 클래스에 이 CIT를 추가하십시오.

2. 가상 관계에 유효한 관계 정의

참고: 이 섹션은 연합에만 관련이 있습니다.

로컬 CMDB CIT에 연결된 연합 CIT를 검색하려면 CMDB의 두 CIT 사이에 유효한 링크 정의가 있어야 합니다.

a. 이러한 링크가 포함된 유효한 링크 XML 파일이 아직 없는 경우 새로 만듭니다.

b. **\validlinks** 폴더에서 어댑터 패키지에 링크 XML 파일을 추가합니다. 자세한 내용은 *HP Universal CMDB 관리 안내서*에서 참조하십시오.

유효한 관계 정의의 예:

다음 예에서 node 유형의 인스턴스와 myclass1 유형의 인스턴스 간의 Containment 유형의 관계는 유효한 관계 정의입니다.

```
<Valid-Links>
  <Valid-Link>
    <Class-Ref class-name="containment">
    <End1 class-name="node">
    <End2 class-name="myclass1">
    <Valid-Link-Qualifiers>
  </Valid-Link>
</Valid-Links>
```

3. 어댑터 구성 정의

a. **어댑터 관리**로 이동합니다.

b. **새 리소스 만들기**  버튼을 클릭하고 **새 어댑터**를 선택합니다.

c. 새 어댑터 대화 상자에서 **통합** 및 **Java 어댑터**를 선택합니다.

d. 만든 어댑터를 마우스 오른쪽 버튼으로 클릭하고 바로 가기 메뉴에서 **어댑터 원본 편집**을 선택합니다.

e. 다음 XML 태그를 편집합니다.

```
<pattern xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
id="newAdapterIdName"
xsi:noNamespaceSchemaLocation="../../Patterns.xsd"
description="Adapter Description" schemaVersion="9.0"
displayName="New Adapter Display Name">
```

```
<deletable>true</deletable>
<discoveredClasses>
<discoveredClass>link</discoveredClass>
<discoveredClass>object</discoveredClass>
</discoveredClasses>
<taskInfo
className="com.hp.ucmdb.discovery.probe.services.dynamic.core.
 AdapterService">
<params
className="com.hp.ucmdb.discovery.probe.services.dynamic.core.
 AdapterServiceParams" enableAging="true"
enableDebugging="false" enableRecording=
"false" autoDeleteOnErrors="success" recordResult="false"
maxThreads="1" patternType="java_adapter"
maxThreadRuntime="25200000">
<className>com.yourCompany.adapter.MyAdapter.MyAdapterClass
</className>
</params>
<destinationInfo
className="com.hp.ucmdb.discovery.probe.tasks.BaseDestinationDa
ta">
<!-- check -->
<destinationData name="adapterId"
description="">${ADAPTER.adapter_id}</destinationData>
<destinationData name="attributeValues"
description="">${SOURCE.attribute_values}</destinationData>
<destinationData name="credentialsId"
description="">${SOURCE.credentials_id}</destinationData>
<destinationData name="destinationId"
description="">${SOURCE.destination_id}</destinationData>
</destinationInfo>
<resultMechanism isEnabled="true">
<autoDeleteCITs isEnabled="true">
<CIT>link</CIT>
<CIT>object</CIT>
</autoDeleteCITs>
</resultMechanism>
</taskInfo>
<adapterInfo>
<adapter-capabilities>
```

```
<support-federated-query>
<!--<supported-classes/> <!—see the section about supported
classes-->
<topology>
<pattern-topology /> <!—or <one-node-topology> -->
</topology>
</support-federated-query>
<!--<support-replicatioin-data>
<source>
<changes-source/>
</source>
<target/>
</adapter-capabilities>
<default-mapping-engine/>
<queries />
<removedAttributes />
<full-population-days-interval>-1</full-population-days-
interval>
</adapterInfo>
<inputClass>destination_config</inputClass>
<protocols />
<parameters>
<!--The description attribute may be written in simple text or
HTML.-->
<!--The host attribute is treated as a special case by UCMDB-->
<!--and will automatically select the probe name (if possible)-
->
<!--according to this attribute’s value.-->
<parameter name="credentialsId" description="Special type of
property, handled by UCMDB for credentials menu" type="integer"
display-name="Credentials ID" mandatory="true" order-index="12"
/>
<parameter name="host" description="The host name or IP address
of the remote machine" type="string" display-name="Hostname/IP"
mandatory="false" order-index="10" />
<parameter name="port" description="The remote machine's
connection port" type="integer" display-name="Port"
mandatory="false" order-index="11" />
</parameters>
```

```
<parameter name="myatt" description="is my att true?"
type="string" display-name="My Att" mandatory="false" order-
index="15" valid-values="True;False"/>True</parameters>
<collectDiscoveredByInfo>true</collectDiscoveredByInfo>
<integration isEnabled="true">
<category >My Category</category>
</integration>
<overrideDomain>${SOURCE.probe_name}</overrideDomain>
<inputTQL>
<resource:XmlResourceWrapper
xmlns:resource="http://www.hp.com/ucmdb/1-0-
0/ResourceDefinition" xmlns:ns4="http://www.hp.com/ucmdb/1-0-
0/ViewDefinition" xmlns:tql="http://www.hp.com/ucmdb/1-0-
0/TopologyQueryLanguage">
<resource xsi:type="tql:Query" group-id="2" priority="low" is-
live="true" owner="Input TQL" name="Input TQL">
<tql:node class="adapter_config" id="-11" name="ADAPTER" />
<tql:node class="destination_config" id="-10" name="SOURCE" />
<tql:link to="ADAPTER" from="SOURCE" class="fcmdb_conf_
aggregation" id="-12" name="fcmdb_conf_aggregation" />
</resource>
</resource:XmlResourceWrapper>
</inputTQL>
<permissions />
</pattern>
```

XML 태그에 대한 자세한 내용은 "XML 구성 태그 및 속성"(207페이지)을 참조하십시오.

4. 지원되는 클래스 정의

***getSupportedClasses()* 메서드를 구현하거나 패턴 XML 파일을 사용하여 지원되는 클래스나 어댑터 코드를 정의합니다.**

```
<supported-classes>
  <supported-class name="HistoryChange" is-derived="false" is-reconciliation-
supported="false" federation-not-supported="false" is-id-reconciliation-supported="false">
  <supported-conditions>
    <attribute-operators attribute-name="change_create_time">
      <operator>GREATER</operator>
      <operator>LESS</operator>
      <operator>GREATER_OR_EQUAL</operator>
      <operator>LESS_OR_EQUAL</operator>
```

```
      <operator>CHANGED_DURING</operator>
    </attribute-operators>
  </supported-conditions>
</supported-class>
```

name	CI 유형의 이름입니다.
is-derived	이 정의에 상속되는 모든 하위 항목을 포함할지 여부를 지정합니다.
is-reconciliation-supported	이 클래스를 조정에 사용할지 여부를 지정합니다.
is-id-reconciliation-supported	이 클래스를 ID 조정에 사용할지 여부를 지정합니다.
federation-not-supported	연합을 위해 이 CIT를 허용하지 않을지(특정 CIT, 예를 들어 연합을 위해서만 정의된 CIT 차단) 여부를 지정합니다.
<supported-conditions>	각 특성에 대해 지원되는 조건을 지정합니다.

5. 어댑터 구현

정의된 기능에 따라 올바른 어댑터 구현 클래스를 선택합니다. 어댑터 구현 클래스는 정의된 기능에 따라 적합한 인터페이스를 구현합니다.

어댑터가 **getTopologyWithReconciliationData**를 구현하고 어댑터 기능에 시작점으로 사용할 수 있는 기능이 포함된 경우 어댑터는 또한 조건 없이(유형만) 조정 데이터를 포함한 요청 토폴로지를 지원해야 합니다. 이 경우 어댑터는 검색된 결과의 전체 조정 데이터를 반환해야 합니다.

global_id에 따라 어댑터 조정 지원을 정의할 수 있습니다. 이 경우 **global_id**는 어댑터에서 지원되는 클래스의 조정 특성 일부로 정의해야 합니다. **global_id**에 따라 어댑터 조정 지원이 정의된 경우 **getTopologyWithReconciliationData()**는 조정 개체 속성의 일부로 **global_id**를 반환해야 합니다. UCMDB는 CIT의 연합 결과 조정에 식별 규칙 대신 **global_id**를 사용합니다.

DataAdapterEnvironment 인터페이스는 연합 API의 일부입니다. 이 인터페이스는 데이터 어댑터의 환경을 나타냅니다. 여기에는 어댑터가 작동하는 데 필요한 환경 API가 포함되어 있습니다. DataAdapterEnvironment 인터페이스에 대한 자세한 내용은 "DataAdapterEnvironment 인터페이스"(209페이지)를 참조하십시오.

6. 조정 규칙 정의 또는 매핑 엔진 구현

어댑터가 연합 TQL 쿼리를 지원하는 경우 매핑 엔진을 정의하는 두 가지 옵션이 있습니다.

- CMDB의 매핑에 대한 내부 조정 규칙을 사용하는 기본 매핑 엔진을 사용합니다. 이 엔진을 사용하려면 **<default-mapping-engine>** XML 태그를 비워 두십시오.
- 매핑 엔진 인터페이스를 구현하고 나머지 어댑터 코드가 있는 JAR을 배치하여 직접 매핑 엔진을

작성합니다. 이렇게 하려면 XML 태그 **<default-mapping-engine>com.yourcompany.map.MyMappingEngine</default-mapping-engine>**을 사용합니다.

7. 구현에 필요한 Jar을 클래스 경로에 추가

   클래스를 구현하려면 코드 편집기 클래스 경로에 federation_api.jar 파일을 추가합니다.

8. 어댑터 배포

   어댑터 패키지를 배포합니다. 일반적으로 패키지를 배포하는 방법에 대한 자세한 내용은 *HP Universal CMDB 관리 안내서*에서 패키지 관리자를 참조하십시오.

   패키지에는 다음 엔터티가 포함되어야 합니다.

   - **새 CIT 정의(선택 사항):**
   - **어댑터가 UCMDB에 아직 없는 새 CI 유형을 지원하는 경우에만 사용됩니다.**
   - **새 CIT 정의는 패키지의** class **폴더에 있습니다.**
   - **새 데이터 유형 정의(선택 사항):**
   - **새 CIT에 새 데이터 유형이 필요한 경우에만 사용됩니다.**
   - **새 데이터 유형 정의는 패키지의** typedef **폴더에 있습니다.**
   - **새 유효한 관계 정의(선택 사항):**
   - **어댑터가 연합 TQL을 지원하는 경우에만 사용됩니다.**
   - **새 유효한 관계 정의는 패키지의** validlinks **폴더에 있습니다.**
   - **패턴 구성 XML 파일은 패키지의** discoveryPatterns **폴더에 있어야 합니다.**
   - **설명자.** 패키지 정의를 정의합니다.
   - **패키지의 컴파일된 클래스(보통 jar 파일)을 adapterCode\<adapter id> 폴더 아래에 배치합니다.**

     참고: adapter id **폴더 이름은 어댑터 구성의 값과 같은 값을 갖습니다.**

   - **직접 구성 파일을 만드는 경우에는 패키지의 파일을 adapterCode\<adapter id> 폴더 아래에 배치해야 합니다.**

9. 어댑터 업데이트

   어댑터의 파일 중 바이너리가 아닌 파일은 어댑터 관리 모듈에서 변경할 수 있습니다. 어댑터 관리 모듈에서 구성 파일을 변경하면 어댑터가 새 구성으로 다시 로드됩니다.

패키지의 파일(바이너리 파일 및 바이너리가 아닌 파일 모두)을 편집한 다음 패키지 관리자를 사용하여 패키지를 재배포하는 방법으로 업데이트할 수도 있습니다. 자세한 내용은 *HP Universal CMDB 관리 안내서*에서 "패키지를 배포하는 방법"을 참조하십시오.

샘플 어댑터 만들기

이 예는 샘플 어댑터를 만드는 방법을 보여 줍니다. 이 작업에는 다음 단계가 포함됩니다.

- "어댑터 논리 선택"(207페이지)
- "프로젝트 로드"(207페이지)

1. 어댑터 논리 선택

   어댑터를 구현할 때 구현의 조건 논리(속성 조건, ID 조건, 조정 조건 및 링크 조건)를 처리하는 방법을 선택해야 합니다.

   a. 전체 데이터를 어댑터 메모리로 가져와 필요한 CI 인스턴스를 선택하거나 필터링할 수 있도록 합니다.

   b. 모든 조건을 데이터 원본 언어로 변환하여 데이터를 필터링하거나 선택할 수 있도록 합니다. 예:

      ◦ 조건을 SQL 쿼리로 변환합니다.

      ◦ 조건을 Java API 필터 개체로 변환합니다.

   c. 원격 서비스에서 일부 데이터를 필터링하고 어댑터에서 나머지 데이터를 선택하고 필터링합니다.

   MyAdapter 예에서는 *a* 옵션의 논리가 사용됩니다.

2. 프로젝트 로드

   C:\hp\UCMDB\UCMDBServer\tools\adapter-dev-kit\SampleAdapters 폴더에서 파일을 복사한 후 추가 정보 파일의 지침을 따릅니다.

   > **참고:** 대용량 데이터 집합이 있는 어댑터를 사용하는 경우 연합 성능을 향상시키기 위해 캐싱 및 인덱싱을 사용해야 할 수도 있습니다.

   온라인 Javadoc 문서는 다음 위치에서 볼 수 있습니다.

   C:\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\DBAdapterFramework_JavaAPI\index.html

XML 구성 태그 및 속성

id="newAdapterIdName"	어댑터의 실제 이름을 정의합니다. 로그 및 폴더 조회에 사용됩니다.

displayName="New Adapter Display Name"		어댑터의 표시 이름을 UI에 표시되는 대로 정의합니다.
<className>...</className>		Java 클래스를 구현하는 어댑터의 인터페이스를 정의합니다.
<category >My Category</category>		어댑터의 범주를 정의합니다.
<parameters>		새 통합 포인트를 설정할 때 UI에 사용할 수 있는 구성의 속성을 정의합니다.
	name	속성의 이름(주로 코드에서 사용)입니다.
	description	속성의 표시 힌트입니다.
	type	문자열 또는 정수(부울에 대한 문자열이 있는 유효한 값 사용)입니다.
	display-name	UI의 속성 이름입니다.
	mandatory	이 구성 속성이 사용자에 대해 필수인지 여부를 지정합니다.
	order-index	속성의 순서를 지정합니다(작은 값이 위로).
	valid-values	가능한 유효 값이 `;' 문자로 구분된 목록(예: valid-values="Oracle;SQLServer;MySQL" 또는 valid-values="True;False")입니다.
<adapterInfo>		어댑터의 정적 설정 및 기능의 정의를 포함합니다.
	<support-federated-query>	이 어댑터를 연합 가능한 어댑터로 정의합니다.
	<start-point-adapter>	이 어댑터가 TQL 쿼리 계산의 시작점인지 여부를 지정합니다.
	<one-node-topology>	쿼리를 하나의 연합 쿼리 노드로 연합할 수 있습니다.
	<pattern-topology>	복잡한 쿼리를 연합할 수 있습니다.
	<support-replicatioin-data>	데이터 밀어넣기 흐름 및 채우기 흐름을 실행하는 기능을 정의합니다.
	<source>	이 어댑터를 채우기 흐름에 사용할 수 있습니

		다.
	<push-back-ids>	CI의 글로벌 ID를 테이블의 global_id 열에 다시 밀어넣습니다(orm.xml에서 정의해야 함). **FcmdbPluginPushBackIds** 플러그인을 구현하면 이 동작을 다시 정의할 수 있습니다.
	<changes-source>	이 어댑터를 변경 내용 채우기 흐름에 사용할 수 있습니다.
	<instance-based-data>	이 태그는 어댑터가 인스턴스 기반 채우기 흐름을 지원할 것을 정의합니다.
	<target>	이 어댑터를 데이터 밀어넣기 흐름에 사용할 수 있습니다.
	<default-mapping-engine>	어댑터의 매핑 엔진을 정의할 수 있습니다. 기본적으로 어댑터는 기본 매핑 엔진을 사용합니다. 다른 매핑 엔진을 사용하려면 매핑 엔진의 구현 클래스 이름을 입력합니다().
	<removedAttributes>	결과에서 특정 특성을 제거합니다.
	<full-population-days-interval>	차등 작업 대신 전체 채우기 작업을 실행할 때('x'일마다)를 지정합니다. 변경 내용 흐름과 함께 에이징 메커니즘을 사용합니다.
	<adapter-settings>	어댑터의 설정 목록입니다.
	<list.attributes.for.set>	이전 값을 다시 정의하는 특성을 결정합니다(있는 경우).

DataAdapterEnvironment 인터페이스

OutputStream openResourceForWriting(String resourceName) throws FileNotFoundException;

이 메서드는 쓰기 위해 지정된 이름의 리소스를 엽니다. 이 메서드는 통합을 위한 영구 데이터를 저장하는 데 사용됩니다. Java 메서드를 사용하여 파일을 로드하려고 하는 대신 이 메서드를 사용해야 합니다. 사용자는 스트림에 쓰기를 마쳤을 때 스트림이 닫혔는지 확인해야 합니다. close()/flush()는 리소스를 저장합니다. 이 메서드는 런타임 리소스를 만들지만 어댑터 패키지에 함께 제공된 파일은 덮어쓸 수 없습니다.

매개 변수

- **resourceName:** 검색할 리소스의 이름입니다. 이 이름은 동일한 어댑터의 모든 통합에서 고유해야 합니다.

반환 값

쓰려는 스트림을 반환합니다.

예외

- 이 메서드는 리소스 유형이 파일이고 파일이 없는 경우, 리소스가 일반 파일이 아닌 디렉터리인 경우 또는 일부 다른 이유로 인해 읽으려는 리소스를 열 수 없는 경우 *FileNotFoundException*을 발생시킵니다.
- 이 메서드는 보안 관리자가 있고 해당 *checkRead* 메서드가 파일에 대한 액세스를 거부하는 경우 *SecurityException*을 발생시킵니다.

InputStream openResourceForReading(String resourceName) throws FileNotFoundException;

이 메서드는 읽기 위해 지정된 이름의 리소스를 엽니다. 이 메서드는 통합을 위한 영구 데이터를 읽는 데 사용됩니다. Java 메서드를 사용하여 파일을 로드하려고 하는 대신 이 메서드를 사용해야 합니다. 사용자는 스트림 읽기를 마쳤을 때 스트림이 닫혔는지 확인해야 합니다. 이 메서드는 먼저 어댑터 패키지에 함께 제공된 파일을 로드하려고 시도합니다. 이 파일을 찾을 수 없으면 *DataAdapterEnvironment.openResourceForWriting(String)*에서 런타임에 만든 리소스를 로드하려고 합니다. 런타임 리소스는 프로브 및 서버의 JMX를 사용하여 볼 수 있습니다.

매개 변수

- **resourceName:** 검색할 리소스의 이름입니다. 이 이름은 동일한 어댑터의 모든 통합에서 고유해야 합니다.

반환 값

읽을 스트림을 반환합니다.

예외

- 이 메서드는 리소스 유형이 **파일**이고 파일이 없는 경우, 리소스가 일반 파일이 아닌 디렉터리인 경우 또는 일부 다른 이유로 인해 읽으려는 리소스를 열 수 없는 경우 *FileNotFoundException*을 발생시킵니다.
- 이 메서드는 보안 관리자가 있고 해당 *checkRead* 메서드가 파일에 대한 읽기 권한을 거부하는 경우 *SecurityException*을 발생시킵니다.

Properties openResourceAsProperties(String propertiesFile) throws IOException;

이 메서드는 지정된 이름의 리소스를 연 다음 *속성* 구조로 로드합니다. 이 메서드는 통합을 위한 영구 데이터를 읽는 데 사용됩니다. Java 메서드를 사용하여 **.properties** 파일을 로드하려고 하는 대신 이 메서드를 사용해야 합니다. 이 메서드는 먼저 어댑터 패키지에 함께 제공된 파일을 로드하려고 시도합니다.

이 파일을 찾을 수 없으면 *DataAdapterEnvironment.openResourceForWriting(String)*에서 런타임에 만든 리소스를 로드하려고 합니다. 런타임 리소스는 프로브 및 서버의 JMX를 사용하여 볼 수 있습니다.

매개 변수

- **propertiesFile:** 검색할 리소스의 이름입니다. 이 이름은 동일한 어댑터의 모든 통합에서 고유해야 합니다.

반환 값

속성에 표시된 파일 컨텐츠를 반환합니다.

예외

- 이 메서드는 리소스 유형이 **파일**이고 파일이 없는 경우, 리소스가 일반 파일이 아닌 디렉터리인 경우 또는 일부 다른 이유로 인해 읽으려는 리소스를 열 수 없는 경우 *FileNotFoundException*을 발생시킵니다.
- 이 메서드는 보안 관리자가 있고 해당 *checkRead* 메서드가 파일에 대한 읽기 권한을 거부하는 경우 *SecurityException*을 발생시킵니다.
- 이 메서드는 속성 파일을 *Properties* 개체로 변환하지 못한 경우 *IOException*을 발생시킵니다.

String openResourceAsString(String resourceName) throws IOException;

이 메서드는 지정된 이름의 리소스를 연 다음 문자열로 로드합니다. 이 메서드는 통합을 위한 영구 데이터를 읽는 데 사용됩니다. Java 메서드를 사용하여 파일을 로드하려고 하는 대신 이 메서드를 사용해야 합니다.

이 메서드는 먼저 어댑터 패키지에 함께 제공된 파일을 로드하려고 시도합니다. 이 파일을 찾을 수 없으면 *DataAdapterEnvironment.openResourceForWriting(String)*에서 런타임에 만든 리소스를 로드하려고 합니다. 런타임 리소스는 프로브 및 서버의 JMX를 사용하여 볼 수 있습니다.

매개 변수

- **resourceName:** 검색할 리소스의 이름입니다. 이 이름은 동일한 어댑터의 모든 통합에서 고유해야 합니다.

반환 값

문자열 형식으로 표시된 파일 컨텐츠를 반환합니다.

예외

- 이 메서드는 리소스 유형이 **파일**이고 파일이 없는 경우, 리소스가 일반 파일이 아닌 디렉터리인 경우 또는 일부 다른 이유로 인해 읽으려는 리소스를 열 수 없는 경우 *FileNotFoundException*을 발생시킵니다.
- 이 메서드는 보안 관리자가 있고 해당 *checkRead* 메서드가 파일에 대한 읽기 권한을 거부하는 경우 *SecurityException*을 발생시킵니다.
- 이 메서드는 I/O 오류가 발생하는 경우 *IOException*을 발생시킵니다.

public void saveResourceFromString(String relativeFileName, String value) throws IOException;

이 메서드는 문자열을 받은 다음 리소스로 저장합니다. 이 메서드는 통합을 위한 영구 데이터를 저장하는 데 사용됩니다. Java 메서드를 사용하여 파일을 저장하려고 하는 대신 이 메서드를 사용해야 합니다. 이 메서드는 문자열을 스트림으로 변환하고 이 스트림을 리소스에 저장합니다. 이 메서드는 런타임 리소스를 만들지만 어댑터 패키지에 함께 제공된 파일은 덮어쓸 수 없습니다. 런타임 리소스는 프로브 및 서버의 JMX를 사용하여 볼 수 있습니다.

매개 변수

- **relativeFileName:** 검색할 리소스의 이름입니다. 이 이름은 동일한 어댑터의 모든 통합에서 고유해야 합니다.
- **value:** 리소스로 저장할 문자열입니다.

예외

이 메서드는 I/O 오류가 발생하는 경우 IOException을 발생시킵니다.

boolean resourceExists(String resourceName);

이 메서드는 지정된 리소스 이름이 있는지 확인합니다. 이 메서드는 어댑터 패키지에 함께 제공된 파일을 찾은 다음 *DataAdapterEnvironment.openResourceForWriting(String)*에서 런타임에 만든 리소스를 찾습니다.

매개 변수

- **resourceName:** 검색할 리소스의 이름입니다. 이 이름은 동일한 어댑터의 모든 통합에서 고유해야 합니다.

반환 값

*resourceName*이 있는 경우 **True**를 반환합니다.

boolean deleteResource(String resourceName);

이 메서드는 영구 데이터에서 지정된 리소스를 삭제합니다. 이 메서드는 런타임 리소스를 삭제하지만 어댑터 패키지에 함께 제공된 파일은 삭제할 수 없습니다. 런타임 리소스는 프로브 및 서버의 JMX를 사용하여 볼 수 있습니다.

매개 변수

- **resourceName:** 삭제할 리소스의 이름입니다. 이 이름은 동일한 어댑터의 모든 통합에서 고유해야 합니다.

반환 값

리소스를 삭제한 경우 **True**를 반환합니다.

Collection<문자열> listResourcesInPath(String path);

이 메서드는 지정된 리소스 경로에 있는 리소스 목록을 검색합니다. 이 메서드는 어댑터 패키지에 함께 제공된 파일을 찾은 다음 *DataAdapterEnvironment.openResourceForWriting(String)*에서 런타임에 만든 리소스를 찾습니다. 런타임 리소스는 프로브 및 서버의 JMX를 사용하여 볼 수 있습니다.

매개 변수

- **path:** 리소스 경로입니다(예: "META-INF/myfiles/").

반환 값

경로에 있는 리소스 목록을 반환합니다.

DataAdapterLogger getLogger();

어댑터가 사용할 로거를 검색합니다. 이 로거는 어댑터의 이벤트를 기록하는 데 사용됩니다.

반환 값

DataAdapter가 사용하는 로거를 반환합니다.

DestinationConfig getDestinationConfig();

이 메서드는 통합의 대상 구성을 검색합니다. 이 구성에는 통합을 위한 모든 연결 및 실행 중인 설정이 포함됩니다.

반환 값

어댑터의 DestinationConfig를 반환합니다.

int getChunkSize();

이 메서드는 이 통합에 대해 요청된 채우기 청크 크기를 검색합니다.

반환 값

채우기 청크 크기를 반환합니다.

int getPushChunkSize();

이 메서드는 이 통합에 대해 요청된 밀어넣기 청크 크기를 검색합니다.

반환 값

밀어넣기 청크 크기를 반환합니다.

ClassModel getLocalClassModel();

이 메서드는 로컬 UCMDB의 클래스 모델에 대한 정보를 쿼리하기 위한 클래스 모델을 검색합니다. 이 메서드는 업데이트된 ClassModel을 가져옵니다. ClassModel 개체가 반환된 후에는 클래스 모델 변경 내용에 대해 업데이트되지 않습니다. 업데이트된 클래스 모델을 검색하려면 이 메서드를 다시 사용하여 검색합니다.

반환 값

UCMDB의 클래스 모델을 반환합니다.

CustomerInformation getLocalCustomerInformation();

이 메서드는 어댑터를 실행 중인 고객에 대한 고객 정보를 검색합니다.

반환 값

어댑터를 실행 중인 고객에 대한 고객 정보를 반환합니다.

Object getSettingValue(String name);

이 메서드는 특정 어댑터 설정을 검색합니다.

매개 변수

name: 설정의 이름입니다.

반환 값

개체 설정 값을 반환합니다.

Map<문자열, 개체> getAllSettings();

이 메서드는 모든 어댑터 설정을 검색합니다.

반환 값

어댑터 설정을 반환합니다.

boolean isMTEnabled();

이 메서드는 서버 환경에서 MT(다중 테넌트)를 지원하는지 여부를 확인합니다.

반환 값

서버 환경에서 MT를 지원하는 경우 **true**를 반환하고 그렇지 않으면 **false**를 반환합니다.

String getUcmdbServerHostName();

이 메서드는 로컬 UCMDB 서버 호스트 이름을 반환합니다.

반환 값

로컬 UCMDB 서버 호스트 이름을 반환합니다.

7장: 밀어넣기 어댑터 개발

이 장의 내용:

밀어넣기 어댑터 개발 및 배포

일반 밀어넣기 어댑터는 UCMDB 데이터를 외부 데이터 저장소(데이터베이스 및 타사 응용 프로그램)로 밀어 넣는 통합을 빠르게 개발할 수 있는 공통 플랫폼을 제공합니다. 일반 밀어넣기 어댑터는 데이터를 밀어넣는 데 사용되는 프로토콜에 따라 분류됩니다. 일반 XML 밀어넣기 어댑터를 사용하는 XML을 통한 밀어넣기에 대한 자세한 내용은 "일반 XML 밀어넣기 어댑터 SQL 쿼리"(228페이지)를 참조하십시오. 일반 웹 서비스 밀어넣기 어댑터를 사용하는 웹 서비스를 통한 밀어넣기에 대한 자세한 내용은 "일반 웹 서비스 밀어넣기 어댑터"(228페이지)를 참조하십시오.

일반 밀어넣기 어댑터를 기반으로 하는 사용자 지정 통합을 개발하려면 다음 사항이 필요합니다.

- 적절한 일반 밀어넣기 어댑터 템플릿 파일에서 새 어댑터 패키지 작성. 자세한 내용은 "어댑터 패키지 빌드"(217페이지)를 참조하십시오.
- UCMDB CI 링크 유형과 외부 데이터 항목 간 매핑. 매핑은 XML로 저장되고 각 외부 데이터 저장소로 사용자 지정됩니다. 자세한 내용은 "매핑 만들기"(220페이지)를 참조하십시오.
- 데이터 항목을 외부 데이터 저장소로 밀어 넣는 Jython 스크립트. 자세한 내용은 "Jython 스크립트 작성"(223페이지)을 참조하십시오.
- 추가 어댑터별 단계(예: XML 밀어넣기 어댑터의 경우 쓰려는 파일의 경로 선택, 웹 서비스 밀어넣기 어댑터의 경우 데이터 수신기 만들기)

어댑터 패키지 빌드

MDR별 밀어넣기 어댑터를 새로 만들려면 일반 어댑터의 사본을 만든 다음 편집하여 특정 밀어넣기 대상에 대한 어댑터가 되도록 사용자 지정해야 합니다.

일반 어댑터 패키지는 다음 두 위치 중 한 곳에 있습니다.

- 일반 XML 밀어넣기 어댑터: **hp\UCMDB\UCMDBServer\content\adapters\push-adapter.zip**
- 일반 웹 서비스 어댑터: **hp\UCMDB\UCMDBServer\content\adapters\web-service-push-adapter.zip**

일반 밀어넣기 어댑터에서 새 밀어넣기 어댑터를 만들려면 다음을 수행합니다.

1. 작업 폴더에 선택한 패키지 zip 파일의 압축을 풉니다.
2. 이름 바꾸기 및 교체 단계를 준비하기 위해 다음 디렉터리를 검토합니다.
   - **adapterCode: C:\hp\UCMDB\UCMDBServer\runtime\fcmdb\CodeBase** 디렉터리에 배포된 디렉터리가 포함되어 있습니다. 여기에 배포된 Jar 파일은 프로브를 자동으로 다시 시작하지 않으며 프로브의 CLASSPATH에 자동으로 표시되지 않습니다.
   - **discoveryConfigFiles:** 어댑터의 매핑 정의가 포함되어 있으며 올바른 Jython 스크립트(**push.properties**)를 가리킵니다.
   - **discoveryPatterns:** UCMDB 서버에 배포된 어댑터의 XML 정의가 포함되어 있습니다.
   - **discoveryScripts:** 타사 데이터 저장소에 대한 연결을 설정하고 데이터를 밀어넣는 데 사용할 어댑터의 Jython 스크립트가 포함되어 있습니다.
   - **discoveryResources:** 웹 서비스의 Java 통합 클래스가 포함된 **UCMDBDataReceiver.jar**이 포함되어 있습니다.

   참고: 이 패키지를 배포하면 프로브가 다시 시작되어 이 **.jar**이 프로브의 CLASSPATH에 포함됩니다. 패키지 배포 이외의 작업은 필요하지 않습니다.
3. 압축을 푼 어댑터 디렉터리 구조 내에서 다음 변경을 수행합니다.
   a. **discoveryConfigFiles\<푸시_어댑터_이름>:** "PushAdapter" 또는 "XMLtoWebService" 디렉터리 이름을 새 밀어넣기 어댑터의 이름(예: "myPushAdapter")으로 바꿉니다.
   b. **discoveryConfigFiles\<푸시_어댑터_이름>\push.properties: push.properties** 파일에서 다음을 수행합니다.
      - **jythonScript.name**의 이름을 새 밀어넣기 어댑터에서 사용할 Jython 스크립트의 이름(예: **pushToMyService.py**)으로 업데이트합니다.
      - 매핑 파일의 이름을 새 밀어넣기 어댑터에서 사용할 이름(예: myPushAdapter_mappings)으로 업데이트합니다. **.xml** 확장자는 자동으로 입력되므로 추가하지 마십시오.
   c. **discoveryPatterns\<푸시 어댑터 이름>.xml:** 이 파일의 이름을 새 어댑터의 정의 XML 파일 이

름(예: **my_push_adapter.xml**)으로 바꿉니다.

d. **discoveryPatterns\<푸시_어댑터>.xml:** 이 파일을 다음과 같이 업데이트합니다.

- XML 요소 **<pattern>**의 경우: id 및 description 특성을 적절히 설정합니다. 예:

  <pattern id="PushAdapter" xsi:noNamespaceSchemaLocation="../../Patterns.xsd" description="Discovery Pattern Description" schemaVersion="9.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  다음과 같이 변경합니다.

  <pattern id="MyPushAdapter" displayLabel="My Push Adapter" xsi:noNamespaceSchemaLocation="../../Patterns.xsd" description="Discovery Pattern Description" schemaVersion="9.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

- XML 요소 **<parameters>**의 경우: 어댑터의 필요에 따라 하위 요소를 업데이트합니다. 기본적으로 밀어넣기 어댑터를 정의하는 데는 다음 하위 요소가 사용됩니다. 이러한 값은 어댑터를 구성한 후에 통합 스튜디오에서 통합 포인트를 정의할 때 할당됩니다. 매개 변수 목록을 업데이트하여 필수 연결 특성을 매개 변수 목록에 반영합니다. **probeName** 특성은 제거하지 마십시오.
  - **host:** 웹 서비스를 호스팅하는 서버 이름입니다.
  - **port:** 수신 대기 UCMDB 데이터 수신기 서비스의 포트입니다.
  - **Web Service Push Adapter: uri** - 데이터 수신기의 서비스 끝점 주소를 구성하는 URL의 나머지 부분입니다.
  - **probeName:** 밀어넣기 작업이 실행되는 Data Flow Probe를 정의합니다.
- XML 요소 **<integration>**의 경우: 하위 요소 **<category>**의 값을 Generic 이외의 다른 값으로 업데이트합니다. 기본적으로 일반 범주에 속한 통합 어댑터는 통합 스튜디오에 표시되지 않습니다. 타사 데이터 저장소와 통합하려는 경우 이 값을 "Third Party"로 설정합니다. HP BTO 제품과 통합하려는 경우에는 이 값을 "HP BTO Products"로 설정합니다.

e. **adapterCode\PushAdapter:** 이 폴더의 이름을 이전 단계에서 사용된 어댑터 ID로 바꿉니다(예: **adapterCode\MyPushAdapter**).

f. **discoveryScripts\<Jython_밀어넣기_스크립트>.py: push.properties jythonScript.name** 속성에 정의된 것과 동일한 이름의 파일을 만듭니다. **discoveryScript** 파일에는 CI 및 외부 Oracle 데이터베이스의 링크를 삽입하는 스크립트가 있습니다. **discoveryScripts\pushScript.py**를 작성한 스크립트로 바꿉니다. 자세한 내용은 "Jython 스크립트 작성"(223페이지)을 참조하십시오. 스크립트의 이름을 바꾸면 **adapterCode\<어댑터 ID>\push.properties**의 **jythonScript.name** 속성이 이에 따라 업데이트됩니다.

- XML 밀어넣기 어댑터: **pushScript.py**
- 웹 서비스 밀어넣기 어댑터: **XMLtoWebService.py**

g. **tql\<통합_TQL>:** 일반 패키지와 마찬가지로 통합 TQL의 TQL XML 정의를 이 디렉터리에 배치합니다. 어댑터 패키지를 배포하면 이 폴더의 모든 TQL이 배포됩니다.

h. **discoveryConfigFiles\<밀어넣기_어댑터_이름>\mappings:** 통합에서 사용할 TQL에 대해 XML 매핑 파일을 만듭니다. 밀어넣기 어댑터는 매핑 파일의 변환을 통합 TQL의 결과에 적용한

다음 Ad-hoc 작업의 세 매개 변수(**addResult**, **updateResult** 및 **deleteResult**)에 있는 데이터를 Data Flow Probe로 보냅니다.

i. **adapterCode\<어댑터 ID>\mappings: mappings.xml** 파일을 준비한 매핑 파일로 바꿉니다. 자세한 내용은 "매핑 만들기"(220페이지)를 참조하십시오.

XML 밀어넣기 어댑터: 이 매핑 예는 sql_queries 파일의 ORACLE에 생성된 테이블의 예에 해당합니다.

각 TQL 메서드에 매핑 파일을 사용하려면 해당 TQL의 이름을 각 XML 파일에 할당한 후 그 뒤에 **.xml**을 추가합니다. 이 경우 현재 TQL 이름에 대한 특정 매핑 파일이 없으면 기본적으로 **mappings.xml** 파일이 사용됩니다. **adapterCode\<어댑터 ID>\push.properties**에서 **mappingFile.default** 속성을 변경하여 기본 매핑 파일의 이름을 수정할 수 있습니다.

4. 위 내용을 모두 변경한 후에 위 3단계에 지정된 폴더 및 파일을 선택하여 **.zip** 파일을 만듭니다(예: **my_Push_Adapter.zip**).
5. 패키지 관리자(**관리 > 패키지 관리자**로 이동)를 통해 새로 만든 **.zip** 파일을 UCMDB 서버에 배포합니다.
6. **데이터 흐름 관리 > 통합 스튜디오**에서 통합 포인트를 만들고 통합 포인트가 사용하는 통합 TQL을 정의합니다. 자동 데이터 밀어넣기에 대한 일정을 설정합니다.

문제 해결

새 밀어넣기 어댑터를 작성하는 절차에는 완전하고 올바른 이름 바꾸기 및 교체 작업이 필요합니다. 오류가 있는 경우 어댑터에 영향을 미칠 수 있습니다. 패키지의 압축을 푼 다음 UCMDB 패키지로 사용할 수 있도록 올바르게 다시 압축해야 합니다. 기본 패키지를 예제로 참조하십시오. 일반적인 오류에는 다음이 포함됩니다.

- ZIP 파일의 패키지 디렉터리 위에 다른 디렉터리 포함.
  해결 방법: discoveryResources, **adapterCode** 등의 패키지 디렉터리와 동일한 디렉터리에 패키지를 ZIP으로 압축합니다. ZIP 파일에서 이 디렉터리 위에 다른 디렉터리 수준을 포함하지 마십시오.
- 디렉터리, 파일 또는 파일의 문자열 이름 바꾸기 생략.
  해결 방법: 이 섹션에 나와 있는 지침을 매우 주의하여 따르십시오.
- 디렉터리, 파일 또는 파일의 문자열 이름 바꾸기 맞춤법 오류.
  해결 방법: 이름 바꾸기 절차를 시작한 후에 중간에 이름 지정 규칙을 변경하지 마십시오. 이름을 변경해야 할 경우 거꾸로 되돌아가서 이름을 수정하려 하면 오류가 발생할 위험이 높으므로 이 대신 처음부터 다시 시작하십시오. 또한 문자열을 수동으로 바꾸는 대신 찾기 및 바꾸기를 사용하면 오류 위험을 줄일 수 있습니다.
- 특히 **discoveryResources** 및 **adapterCode** 디렉터리에서 다른 어댑터와 동일한 파일 이름을 갖는 어댑터 배포.
  해결 방법: 매핑 파일이 동일한 UCMDB 환경에서 다른 어댑터와 동일한 이름을 사용하지 못하도록 하는, 알려진 문제가 있는 UCMDB 버전을 사용 중일 수 있습니다. 중복된 이름을 갖는 패키지를 배포하려고 할 경우 패키지 배포에 실패합니다. 이러한 파일이 서로 다른 디렉터리에 있는 경우에도 이 문제가 발생할 수 있습니다. 또한 이 문제는 패키지 내에서 중복되는지 또는 이전에 배포된 다른 패키지와 중복되는지 여부에 관계없이 발생할 수 있습니다.

이제 방금 배포한 새 어댑터를 사용하여 통합 스튜디오에서 새 밀어넣기 어댑터 작업을 만들 수 있습니다.

밀어넣기 어댑터의 TQL 모범 사례

1. TQL 및 보기 트리에 폴더 구조를 만든 다음 새 TQL과 보기를 모두 유지합니다. 이름 지정 규칙을 사용합니다.
2. TQL이 작지 않은 경우 먼저 가장 유사한 TQL을 복사합니다.
3. 한 번에 한 항목씩 변경합니다. 각 내용을 변경한 후에는 저장하고 테스트한 후 미리 보기를 수행합니다. 결과가 요구 사항을 준수할 때까지 반복합니다.

매핑 만들기

원시 TQL 결과 데이터는 UCMDB 클래스 모델 스키마 형식으로 되어 있습니다. 소비자는 다른 데이터 모델을 사용할 수도 있습니다. 밀어넣기 어댑터는 데이터를 사용하기에 더 적합한 형식으로 변환하는 매핑 메커니즘을 제공합니다. 매핑은 직접, 명명 형식 변환에서 상위/하위 집계 및 참조 기능에 이르는 직접 및 복합 변환을 모두 수행합니다.

매핑 사양에 대해서는 "매핑 파일 참조"(247페이지) 섹션에 나와 있습니다. 참조를 사용하여 매핑 파일을 만드십시오.

참고: 어댑터 속성 파일은 매핑 파일의 이름을 참조합니다. 어댑터 구성 파일에서 어댑터는 어댑터의 이름을 사용하여 폴더 구조를 구현합니다. 패키지 관리자에 필요한 고유성을 유지하기 위해 어댑터를 구현할 때는 이 폴더의 이름을 바꿉니다.

매핑 파일 작성

1. 기본 매핑 파일로 시작합니다.
2. 어댑터를 배포하고 한 번 실행합니다.
3. 결과를 검토합니다.
4. 변경해야 할 사항을 식별합니다.
5. 이전 단계에서 식별한 사항을 변경합니다. 다음 목록은 변경 순서를 안내합니다.
   a. 위쪽의 변형되지 않은 섹션부터 시작합니다. 각 내용을 변경한 후에 어댑터가 실행되는지 확인합니다.
   b. 원본 CI 섹션을 TQL 결과의 UCMDB 이름으로 변경합니다.
   c. 먼저 키를 매핑합니다.
   d. 그런 다음 직접 매핑을 모두 추가합니다.
   e. 복잡한 매핑을 추가합니다.
   f. 링크 매핑을 추가합니다.

매핑된 데이터가 사용하기에 적합해질 때까지 2-5단계를 반복합니다. 새 밀어넣기 어댑터를 만들 적절한 일반 어댑터 패키지를 선택합니다.

매핑 파일은 모든 유형의 밀어넣기 어댑터에 대해 동일한 방식으로 작동합니다. 일반 XML 밀어넣기 어댑터는 매핑된 결과를 파일에 씁니다. 일반 웹 서비스 밀어넣기 어댑터는 XML 결과를 데이터 수신기로 보냅니다. 자세한 내용은 "일반 웹 서비스 밀어넣기 어댑터"(228페이지)를 참조하십시오.

매핑 파일 준비

참고: mappings.xml 파일을 만들지 않고 매핑없이 CMDB에 있는 모든 CI 및 관계를 검색할 수 있습니다. 그러면 해당 특성과 함께 모든 CI 및 관계가 반환됩니다.

매핑 파일을 준비하는 방법은 다음 두 가지입니다.

- 하나의 전역 매핑 파일을 준비할 수 있습니다.

  모든 매핑이 **mappings.xml**이라는 하나의 파일에 배치됩니다.
- 각 밀어넣기 쿼리에 대해 별도의 파일을 준비할 수 있습니다.

  각 매핑 파일은 **<쿼리 이름>.xml**이라고 합니다.

자세한 내용은 "매핑 파일 스키마"(249페이지)를 참조하십시오.

이 작업에는 다음 단계가 포함됩니다.

- "mappings.xml 파일 만들기"(221페이지)
- "CI 매핑"(221페이지)
- "링크 매핑"(222페이지)

1. mappings.xml 파일 만들기

   매핑 파일 구조는 다음과 같이 만들어집니다(기존 파일을 템플릿으로 사용).

```
<?xml version="1.0" encoding="UTF-8"?>
<integration>
   <info>
      <source name="UCMDB"  versions="9.x" vendor="HP" >
      <!-- for example: -->
      <target name="Oracle" versions="11g" vendor="Oracle" >
   </info>
   <targetcis>
      <!--- CI Mappings --->
   </targetcis>
   <targetrelations>
      <!--- Link Mappings --->
   </ targetrelations>
</integration>
```

2. CI 매핑

다음 두 가지 방법으로 CMDB CI 유형을 매핑합니다.

- 해당 유형의 CI와 상속된 모든 유형이 동일한 방법으로 매핑되도록 CI 유형을 매핑합니다.

```
<source_ci_type_tree name="node" mode="update_else_insert">
  <apioutputseq>1</apioutputseq>
  <target_ci_type  name="host">
    <targetprimarykey>
      <pkey>name</pkey>
    </targetprimarykey
    <target_attribute name=" name" datatype="STRING">
      <map type="direct" source_attribute="name" >
    </target_attribute>
    <!-- more target attributes --->
  </target_ci_type>
</source_ci_type_tree>
```

- 해당 유형의 CI만 처리되도록 CI 유형을 매핑합니다. 다음 두 가지 방법 중 하나를 사용하여 상속된 유형도 매핑해야 상속된 유형의 CI가 처리됩니다.

```
<source_ci_type name="node" mode="update_else_insert">
  <apioutputseq>1</apioutputseq>
  <target_ci_type  name="host">
    <targetprimarykey>
      <pkey>name</pkey>
    </targetprimarykey
    <target_attribute name=" name" datatype="STRING">
      <map type="direct" source_attribute="name" >
    </target_attribute>
    <!-- more target attributes --->
  </target_ci_type>
</source_ci_type>
```

직접 매핑된 CI 유형(**source_ci_type_tree**를 사용하여 매핑되는 상위 중 하나)은 상위 맵을 해당 **source_ci_type_tree** 또는 **source_ci_type**에 표시하여 다시 정의할 수도 있습니다.

가능한 경우 **source_ci_type_tree**를 사용하는 것이 좋습니다. 그렇지 않으면 매핑 파일에 나타나지 않는 CI 유형의 결과 CI가 Jython 스크립트에 전송되지 않습니다.

3. 링크 매핑

다음 두 가지 방법으로 링크를 매핑합니다.

- 해당 유형의 링크와 상속된 모든 링크가 동일한 방법으로 매핑되도록 링크를 매핑합니다.

```
<source_link_type_tree name="dependency" target_link_type="dependency"
mode="update_else_insert" source_ci_type_end1="webservice" source_ci_type_
end2="sap_gateway">
```

```
  <target_ci_type_end1 name="webservice" >
  <target_ci_type_end2 name="sap_gateway" >
    <target_attribute name="name" datatype="STRING">
      <map type="direct" source_attribute="name" >
    </target_attribute>
</source_link_type_tree>
```

- 해당 유형의 링크만 처리되도록 링크를 매핑합니다. 다음 두 가지 방법 중 하나를 사용하여 상속된 유형도 매핑해야 상속된 유형의 링크가 처리됩니다.

```
<link source_link_type="dependency" target_link_type="dependency" mode="update_
else_insert" source_ci_type_end1="webservice" source_ci_type_end2="sap_gateway">
    <target_ci_type_end1 name="webservice" >
    <target_ci_type_end2 name="sap_gateway" >
    <target_attribute name="name" datatype="STRING">
      <map type="direct" source_attribute="name" >
  </target_attribute>
</link>
```

Jython 스크립트 작성

매핑 스크립트는 일반 Jython 스크립트이므로 Jython 스크립트의 규칙을 따라야 합니다. 자세한 내용은 "Jython 어댑터 개발"(36페이지)을 참조하십시오.

이 스크립트는 **DiscoveryMain** 함수를 포함해야 합니다. 이 함수는 성공 시 빈 **OSHVResult** 또는 **DataPushResults** 인스턴스를 반환할 수 있습니다.

오류를 보고하려면 스크립트에서 다음과 같이 예외를 발생시켜야 합니다.

```
raise Exception('Failed to insert to remote UCMDB using TopologyUpdateService. See log of the remote UCMDB')
```

DiscoveryMain 함수에서는 외부 응용 프로그램에서 밀어 넣거나 삭제할 데이터 항목을 다음과 같은 방법으로 가져올 수 있습니다.

```
# get add/update/delete result objects (in XML format) from the Framework
addResult = Framework.getTriggerCIData('addResult')
updateResult = Framework.getTriggerCIData('updateResult')
deleteResult = Framework.getTriggerCIData('deleteResult')
```

외부 응용 프로그램에 대한 클라이언트 개체는 다음과 같은 방법으로 가져올 수 있습니다.

```
oracleClient = Framework.createClient()
```

이 클라이언트 개체는 어댑터가 프레임워크를 통해 전달한 자격 증명 ID, 호스트 이름 및 포트 번호를 자동으로 사용합니다.

어댑터에 대해 정의한 연결 매개 변수를 사용해야 하는 경우(자세한 내용은 "어댑터 패키지 빌드"(217페이지)의 **discoveryPatterns\push_adapter.xml** 파일 편집 단계 참조) 다음 코드를 사용하십시오.

```
propValue = str(Framework.getDestinationAttribute('<Connection Property Name'))
```

예:

```
serverName = Framework.getDestinationAttribute('ip_address')
```

이 섹션에는 다음 내용도 포함됩니다.

- "매핑 결과 사용"(224페이지)
- "스크립트에서 연결 테스트 처리"(226페이지)

매핑 결과 사용

일반 밀어넣기 어댑터는 대상 시스템에서 추가하거나, 업데이트하거나, 삭제할 데이터를 나타내는 XML 문자열을 만듭니다. Jython 스크립트는 이 XML을 분석한 다음 대상에 대한 작업을 추가하거나, 업데이트하거나, 삭제합니다.

Jython 스크립트가 받는 추가 작업의 XML에서 개체 및 링크의 mamId 특성은 항상 유형, 특성 또는 기타 정보가 원격 시스템의 스키마로 변경되기 이전의 원래 개체 또는 링크의 UCMDB 식별자입니다.

업데이트 또는 제거 작업의 XML에서 각 개체 또는 링크의 mamId 특성에는 이전 동기화에서 Jython 스크립트가 반환한 같은 ExternalId의 문자열 표시가 포함됩니다.

XML에서 CI의 id 특성에는 외부 ID로 cmdbId가 포함되거나 또는 CI가 스크립트로 전송될 때 CI에 ExternalId가 있는 경우 해당 CI의 ExternalId가 포함됩니다. 링크의 end1Id 및 end2Id 필드는 각 링크 끝에 대해 외부 id로 cmdbId 또는 링크 끝에 있는 CI가 스크립트로 전송될 때 CI에 ExternalId가 있는 경우 해당 링크 끝의 ExternalId가 포함됩니다.

Jython 스크립트에서 CI가 처리될 때, 스크립트의 반환 값은 CI의 CMDB ID와 지정된 id(스크립트의 각 CI에 지정된 id) 간의 매핑입니다. CI를 처음으로 밀어넣는 경우 해당 CI의 XML에 있는 id가 CMDB id입니다. CI를 처음으로 밀어넣는 경우가 아니면, 해당 CI id는 처음으로 밀어넣을 때 스크립트의 해당 CI에 지정된 id와 같습니다.

다음과 같이 CI XML 스크립트에서 id를 검색합니다.

1. XML의 CI 요소의 id 속성에서 ID를 검색합니다. 예: id = objectElement.getAttributeValue('id')
2. XML에서 id를 검색한 후 특성(string)에서 id를 복원합니다. 예: objectId = CmdbObjectID.Factory.restoreObjectID(id)
3. 이전 단계에서 받은 objectId가 CMDB id인지 확인합니다. 스크립트에 의해 지정된 새 id가 objectId에 있는지 확인하여 이 작업을 수행합니다. 지정된 새 id가 있는 경우 반환된 id는 CMDB id가 아닙니다. 예:
   newId = objectId.getPropertyValue(<스크립트에 지정된 id 특성의 이름>)

   newId가 null이면 XML로 반환된 id가 CMDB id입니다.
4. id가 CMDB id일 경우(즉, newId가 null) 다음을 수행합니다(id가 CMDB id가 아닐 경우 5단계로 이동).
   a. 새 id가 포함된 해당 CI의 속성을 만듭니다. 예: propArray =[TypesFactory.createProperty('<스크립트에 지정된 id 특성의 이름>', '<새 id>')]

b. 해당 CI의 externalId를 만듭니다. 예:
cmdbId = extI.getPropertyValue('internal_id')
className = extI.getType()
externalId = ExternalIdFactory.createExternalCiId(className, propArray)

c. CMDB id를 새로 만든 externalId에 매핑합니다. (다음 단계에서는 해당 매핑을 어댑터에 반환합니다.) 예: objectMappings.put(cmdbId, externalId)

d. 모든 CI 및 링크가 매핑되면 다음을 수행합니다.
updateResult = DataPushResultsFactory.createDataPushResults(objectMappings, linkMappings);
return updateResult

5. id가 새 id일 경우(즉, newId가 null이 아님), externalId는 newId입니다.

다음과 같이 각 CI 및 링크에 대한 밀어넣기 상태를 보고할 수도 있습니다.

1. updateStatus = ReplicationActionDataFactory.createUpdateStatus();

   여기서 updateStatus는 CI 및 링크의 상태가 포함된 UpdateStatus 클래스의 인스턴스입니다.

2. reportCIStatus 또는 reportRelationStatus 메서드를 호출하여 updateStatus에 상태를 추가합니다.

   예:

   status = ReplicationActionDataFactory.createStatus(Severity.FAILURE, 'Failed', ERROR_CODE_CI, errorParams,Action.ADD);

   updateStatus.reportCIStatus(externalId, status);

   여기서 ERROR_CODE_CI는 어댑터 **properties.errors** 파일에 나타나는 오류 메시지 수이고 (**properties.errors** 파일에 대한 자세한 내용은 "오류 작성 규칙"(61페이지) 참조), errorParams에는 메시지에 전달할 매개 변수가 포함됩니다. 자세한 내용은 **ReplicationActionDataFactory** javadoc를 참조하십시오.

3. 다음과 같이 상태를 포함한 밀어넣기 결과를 만듭니다.

   updateResult = DataPushResultsFactory.createDataPushResults(objectMappings, linkMappings, updateStatus);

   return updateResult

XML 결과의 예

```
<root>
 <data>
  <objects>
   <Object mode="update_else_insert" name="UCMDB_UNIX" operation="add"
mamId="0c82f591bc3a584121b0b85efd90b174"
id="HiddenRmiDataSource%0Aunix%0A1%0Ainternal_
id%3DSTRING%3D0c82f591bc3a584121b0b85efd90b174%0A">
     <field name="NAME" key="false" datatype="char" length="255">UNIX5</field>
```

```
    <field name="DATA_NOTE" key="false" datatype="char" length="255"></field>
   </Object>
  </objects>
  <links>
    <link targetRelationshipClass="TALK" targetParent="unix" targetChild="unix" operation="add"
mode="update_else_insert"
mamId="265e985c6ec51a8543f461b30fa58f81"
id="end1id%5BHiddenRmiDataSource%0Aunix%0A1%0Ainternal_
id%3DSTRING%3D41372a1cbcaba27b214b84a2ec9eb535%0A%5D%0Aend2id%
5BHiddenRmiDataSource%0Aunix%0A1%0Ainternal_
id%3DSTRING%3D0c82f591bc3a584121b0b85efd90b174%0A%5D%0AHiddenRmi
DataSource%0Atalk%0A1%0Ainternal_
id%3DSTRING%3D265e985c6ec51a8543f461b30fa58f81%0A">
    <field name="DiscoveryID1">41372a1cbcaba27b214b84a2ec9eb535</field>
    <field name="DiscoveryID2">0c82f591bc3a584121b0b85efd90b174</field>
    <field name="end1Id">HiddenRmiDataSource%0Aunix%0A1%0Ainternal_
id%3DSTRING%3D41372a1cbcaba27b214b84a2ec9eb535%0A</field>
    <field name="end2Id">HiddenRmiDataSource%0Aunix%0A1%0Ainternal_
id%3DSTRING%3D0c82f591bc3a584121b0b85efd90b174%0A</field>
    <field name="NAME" key="false" datatype="char" length="255">TALK4</field>
    <field name="DATA_NOTE" key="false" datatype="char" length="255"></field>
    </link>
  </links>
 </data>
</root>
```

참고: datatype="BYTE"**의 경우 반환된 결과 값은** new String([the byte array attribute])**으로 생성된 문자열입니다.** byte[] object**는** <받은 문자열>.getBytes()**를 사용하여 재구성할 수 있습니다. 서버와 프로브 간에 기본 로캘이 다른 경우 서버의 기본 로캘에 따라 다시 만들기가 수행됩니다.**

스크립트에서 연결 테스트 처리

Jython 스크립트를 호출하여 외부 응용 프로그램과의 연결을 테스트할 수 있습니다. 이 경우 testConnection **대상 특성은** true**입니다. 이 특성은 다음과 같은 방법으로 프레임워크에서 가져올 수 있습니다.**

```
testConnection = Framework.getTriggerCIData('testConnection')
```

연결 테스트 모드에서 실행하는 경우 외부 응용 프로그램과의 연결을 설정할 수 없으면 스크립트에서 예외를 발생시켜야 합니다. 그렇지 않고 연결에 성공하면 **DiscoveryMain** 함수가 빈 **OSHVResult**를 반환해야 합니다.

차등 동기화 지원

밀어넣기 어댑터에서 차등 동기화를 지원하려면 **DiscoveryMain** 함수가 **DataPushResults** 인터페이스를 구현하는 개체를 반환해야 합니다. 이 인터페이스에는 Jython 스크립트가 XML에서 받는 ID와 Jython 스크립트가 원격 컴퓨터에서 만드는 ID 간의 매핑이 포함됩니다. 후자의 ID는 **ExternalId** 유형입니다.

CMDB에 있는 CI의 ID를 매개 변수로 수신하는 **ExternalIdUtil.restoreExternal** 명령은 CMDB에 있는 CI의 ID에서 외부 ID를 복원합니다. 예를 들어, 이 명령을 사용하여 차등 동기화를 수행하는 동안 양쪽 끝 중 하나가 벌크에 포함되지 않는(이미 동기화되어서) 상황에서 링크를 수신할 수 있습니다.

밀어넣기 어댑터의 기반이 되는 Jython 스크립트의 **DiscoveryMain** 메서드가 빈 **ObjectStateHolderVector** 인스턴스를 반환하면 이 어댑터에서 차등 동기화를 지원하지 않게 됩니다. 즉, 차등 동기화 작업을 실행할 때도 사실상 전체 동기화가 수행됩니다. 따라서 동기화할 때마다 CMDB에 모든 데이터가 추가되므로 원격 시스템에서 데이터를 업데이트하거나 제거할 수 없습니다.

> **중요:** 버전 9.00 또는 9.01에서 만든 기존 어댑터에서 차등 동기화를 구현하는 경우 버전 9.02 이상에서 push-adapter.zip 파일을 사용하여 어댑터 패키지를 다시 만들어야 합니다. 자세한 내용은 "어댑터 패키지 빌드"(217페이지)를 참조하십시오.

이 작업을 수행하면 밀어넣기 어댑터에서 차등 동기화를 수행할 수 있습니다.

Jython 스크립트가 두 개의 Java 맵이 포함된 **DataPushResults** 개체를 반환합니다. 포함된 Java 맵 하나는 개체 ID 매핑에 대한 맵(키와 값이 ExternalCiId 유형의 개체)이고 또 하나는 링크 ID에 대한 맵(키와 값이 ExternalRelationId 유형의 개체)입니다.

- Jython 스크립트에 다음 **from** 문을 추가합니다.

  ```
  from com.hp.ucmdb.federationspi.data.query.types import ExternalIdFactory
  from com.hp.ucmdb.adapters.push import DataPushResults
  from com.hp.ucmdb.adapters.push import DataPushResultsFactory
  from com.mercury.topaz.cmdb.server.fcmdb.spi.data.query.types import ExternalIdUtil
  ```

- **DataPushResultsFactory** 기본 클래스를 사용하여 **DiscoveryMain** 함수에서 **DataPushResults** 개체를 가져옵니다.

  ```
  # Create the UpdateResult object
  updateResult = DataPushResultsFactory.createDataPushResults(objectMappings, linkMappings);
  ```

- 다음 명령을 사용하여 **DataPushResults** 개체에 대한 Java 맵을 만듭니다.

  ```
  # Prepare the maps to store the mappings if IDs
  objectMappings = HashMap()
  linkMappings = HashMap()
  ```

- **ExternalIdFactory** 클래스를 사용하여 다음 ExternalId ID를 만듭니다.
  - CMDB에서 가져온 개체 또는 링크(예: 추가 작업의 모든 CI를 CMDB에서 가져옴)의 ExternalId:

    externaCIId = ExternalIdFactory.createExternalCmdbCiId(ciType, ciIDAsString)

    externalRelationId = ExternalIdFactory.createExternalCmdbRelationId(linkType, end1ExternalCIId, end2ExternalCIId, linkIDAsString)

  - CMDB에서 가져오지 않은 개체 또는 링크(예: 모든 업데이트 및 제거 작업에 이러한 개체 포함)의 ExternalId:

    myIDField = TypesFactory.createProperty("systemID", "1")

    myExternalId = ExternalIdFactory.createExternalCiId(type, myIDField)

    참고: Jython 스크립트가 기존 정보 및 개체 또는 링크 ID 변경 내용을 업데이트한 경우 이전 외부 ID와 새 외부 ID 간의 매핑을 반환해야 합니다.

- **ExternalIdFactory** 클래스의 **restoreCmdbCiIDString** 또는 **restoreCmdbRelationIDString** 메서드를 사용하여 UCMDB에서 가져온 개체 또는 링크의 외부 ID에서 UCMDB ID 문자열을 검색합니다.
- **ExternalIdUtil** 클래스의 **restoreExternalCiId** 및 **restoreExternalRelationId** 메서드를 사용하여 업데이트 또는 제거 작업 XML의 mamId 특성 값에서 **ExternalId** 개체를 복원합니다.

  참고: ExternalId 개체는 사실상 속성의 배열입니다. 즉, **ExternalId** 개체를 사용하여 원격 시스템에서 데이터를 식별하는 데 필요할 수 있는 정보를 저장할 수 있음을 의미합니다.

일반 XML 밀어넣기 어댑터 SQL 쿼리

어댑터 패키지에서 **adapterCode > PushAdapter > sqlTablesCreation**에 있는 **sql_queries** 파일에는 어댑터를 테스트하기 위해 Oracle의 새 스키마에서 테이블을 만드는 데 필요한 쿼리가 포함되어 있습니다. 이러한 테이블은 adapterCode\<adapter ID>\mappings\mappings.xml 파일에 해당합니다.

참고: sql_queries 파일은 어댑터에 필요하지 않으며 예제일 뿐입니다.

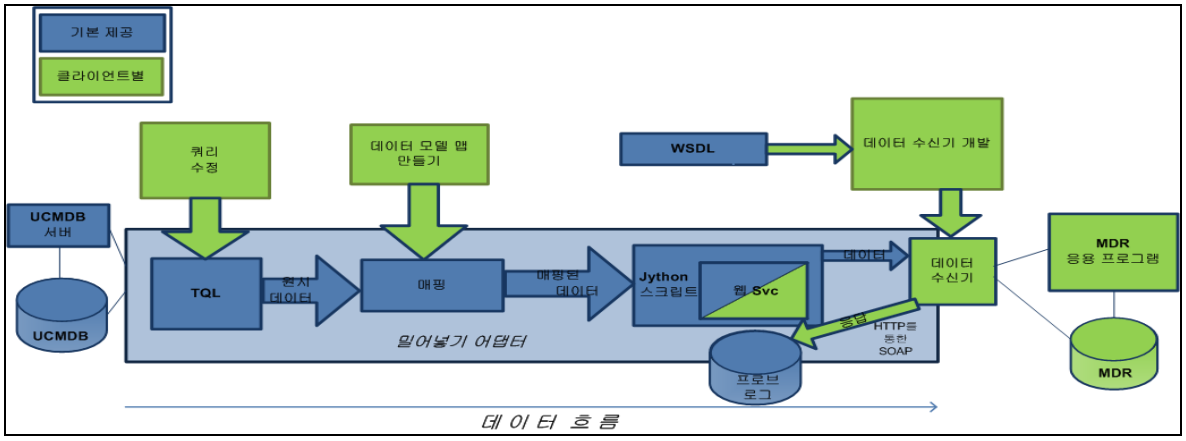
일반 웹 서비스 밀어넣기 어댑터

일반 웹 서비스 밀어넣기 어댑터는 쿼리 데이터가 포함된 SOAP 메시지를 웹 서비스 데이터 수신기로 밀어 넣는(UCMDB에서 시작) 기능을 제공합니다. 매핑된 결과는 표준 SOAP 메시지 형태로 HTTP POST 프로토콜을 통해 데이터 수신기로 전송됩니다. 데이터 수신기는 밀어넣기 어댑터에서 생성된 SOAP 메시지를 인식해야 합니다. 적절한 데이터 수신기를 쉽게 개발할 수 있도록 이 밀어넣기 어댑터에는 WSDL이 제공됩니다.

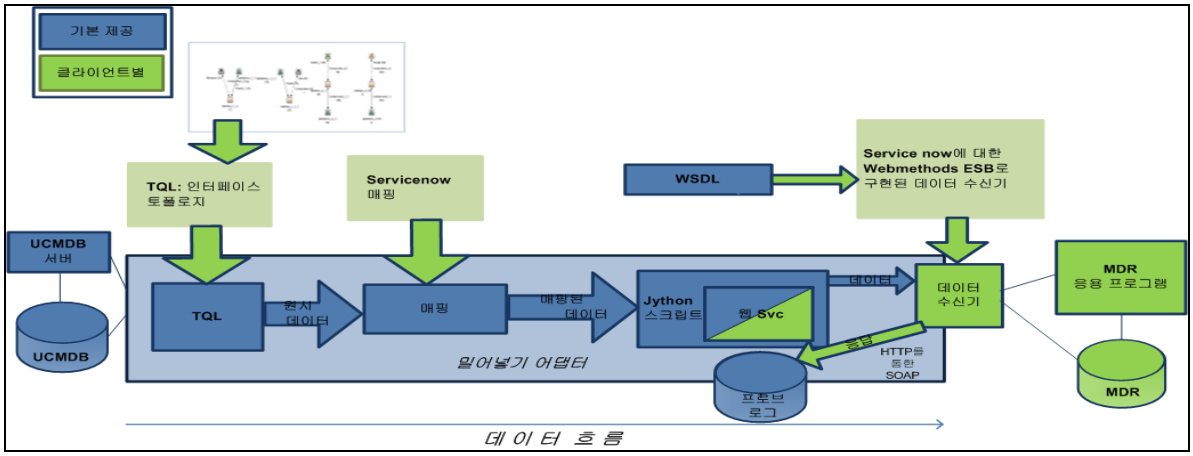
SOAP 메시지 응답 XML의 사용자 지정 처리는 Jython 스크립트에서 가능합니다.

들어오는 매핑된 데이터의 형식을 이해하기 위해 데이터 수신기 개발자는 매핑 파일의 개발자와 의사 소통해야 합니다. **.xsd**는 현재 이 버전의 웹 서비스 밀어넣기 어댑터에 제공되지 않으므로 원래 TQL과 적용된 매핑이 결합된 들어오는 데이터를 어느 정도 반영하여 데이터를 처리해야 합니다.

아래 다이어그램에는 데이터를 클라이언트로 밀어 넣는 웹 서비스 밀어넣기 어댑터 기능이 나와 있습니다. 녹색으로 표시된 항목은 특정 밀어넣기 대상에 대해 어댑터를 구현하기 위해 클라이언트가 사용자 지정하거나 제공하는 것입니다. 파란색으로 표시된 항목은 기본 구성 요소입니다.

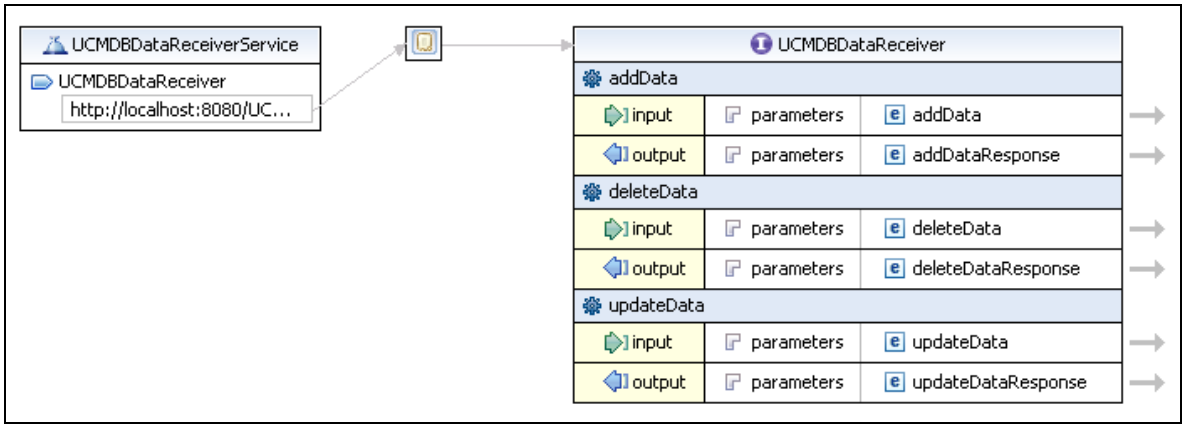


다음은 ESB(Enterprise Service Bus)를 사용하여 MDR별 밀어넣기 어댑터에 대해 일반 웹 서비스 밀어넣기 어댑터를 구현하는 예입니다.



WSDL

WSDL은 웹 서비스를 통해 UCMDB 밀어넣기 어댑터와 통신할 수 있는 데이터 수신기를 만드는 클라이언트 개발자에게 제공됩니다. **UCMDBDataReceiver.wsdl**은 UCDMB에서 데이터 수신기로 데이터를 전달하는 데 사용되는 SOAP 메시지에 대해 설명합니다. 다음은 WSDL의 설계 다이어그램입니다.



데이터 수신기(실제로 서버 또는 SOAP 용어로는 "서비스 끝점")는 UCMDB가 밀어 넣는 데이터 집합에 해당하는 세 가지 메서드, 즉 **addData**, **deleteData** 및 **updateData**를 구현해야 합니다. HTTP 헤더에는 보내려는 데이터의 유형을 나타내는 올바른 **SoapAction** 키워드가 포함되어 있습니다. 데이터 수신기는 비즈니스 논리를 구현하고 데이터를 처리하는 역할을 담당합니다.

기본 WSDL URL은 다음과 같습니다.

- http://localhost:8080/UCMDBDataReceiver/services/UCMDBDataReceiver?wsdl

데이터 수신기에서 구현한 URL은 다음과 유사할 수 있습니다.

- http://testWSPAserver:4444/MyCo.IT.SvcMgt.ws.us:provider/UCMDBDataReceiver?wsdl

웹 서비스의 URL은 끝에 있는 "?wsdl"을 뺀 WSDL URL과 동일합니다.

아래에는 WSDL의 원본이 나와 있습니다.

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions targetNamespace="http://ucmdb.hp.com"
xmlns:apachesoap="http://xml.apache.org/xml-soap" xmlns:impl="http://ucmdb.hp.com"
xmlns:intf="http://ucmdb.hp.com" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<!--WSDL created by Apache Axis version: 1.4 Built on Apr 22, 2006 (06:55:48 PDT)-->

    <wsdl:types>

        <schema elementFormDefault="qualified" targetNamespace="http://ucmdb.hp.com"
        xmlns="http://www.w3.org/2001/XMLSchema">

            <element name="addData">

                <complexType>

                    <sequence>

                    <element name="xmlAdded" type="xsd:string"/>

                    </sequence>
```

```
                </complexType>
            </element>
            <element name="addDataResponse">
                <complexType/>
            </element>
            <element name="deleteData">
                <complexType>
                    <sequence>
                    <element name="xmlDeleted" type="xsd:string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="deleteDataResponse">
                <complexType/>
            </element>
            <element name="updateData">
                <complexType>
                    <sequence>
                    <element name="xmlUpdate" type="xsd:string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="updateDataResponse">
                <complexType/>
            </element>
        </schema>
    </wsdl:types>

    <wsdl:message name="addDataRequest">
        <wsdl:part element="impl:addData" name="parameters">
        </wsdl:part>
```

```
</wsdl:message>
<wsdl:message name="deleteDataResponse">
    <wsdl:part element="impl:deleteDataResponse" name="parameters">
    </wsdl:part>
</wsdl:message>
<wsdl:message name="updateDataResponse">
    <wsdl:part element="impl:updateDataResponse" name="parameters">
    </wsdl:part>
</wsdl:message>
<wsdl:message name="deleteDataRequest">
    <wsdl:part element="impl:deleteData" name="parameters">
    </wsdl:part>
</wsdl:message>
<wsdl:message name="addDataResponse">
    <wsdl:part element="impl:addDataResponse" name="parameters">
    </wsdl:part>
</wsdl:message>
<wsdl:message name="updateDataRequest">
    <wsdl:part element="impl:updateData" name="parameters">
    </wsdl:part>
</wsdl:message>
<wsdl:portType name="UCMDBDataReceiver">
    <wsdl:operation name="addData">
        <wsdlsoap:operation soapAction="addDataRequest"/>
        <wsdl:input message="impl:addDataRequest" name="addDataRequest">
        </wsdl:input>
        <wsdl:output message="impl:addDataResponse" name="addDataResponse">
        </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="deleteData">
        <wsdlsoap:operation soapAction="deleteDataRequest"/>
```

```
            <wsdl:input message="impl:deleteDataRequest" name="deleteDataRequest">
            </wsdl:input>
            <wsdl:output message="impl:deleteDataResponse" name="deleteDataResponse">
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="updateData">
            <wsdlsoap:operation soapAction="updateDataRequest"/>
            <wsdl:input message="impl:updateDataRequest" name="updateDataRequest">
            </wsdl:input>
            <wsdl:output message="impl:updateDataResponse"
            name="updateDataResponse">
            </wsdl:output>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="UCMDBDataReceiverSoapBinding" type="impl:UCMDBDataReceiver">
        <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"
        />
        <wsdl:operation name="addData">
            <wsdl:input name="addDataRequest">
                <wsdlsoap:body use="literal" />
            </wsdl:input>
            <wsdl:output name="addDataResponse">
                <wsdlsoap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="deleteData">
            <wsdl:input name="deleteDataRequest">
                <wsdlsoap:body use="literal" />
            </wsdl:input>
            <wsdl:output name="deleteDataResponse">
                <wsdlsoap:body use="literal" />
            </wsdl:output>
```

```
            </wsdl:operation>
            <wsdl:operation name="updateData">
                <wsdl:input name="updateDataRequest">
                    <wsdlsoap:body use="literal" />
                </wsdl:input>
                <wsdl:output name="updateDataResponse">
                    <wsdlsoap:body use="literal" />
                </wsdl:output>
            </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="UCMDBDataReceiverService">
            <wsdl:port binding="impl:UCMDBDataReceiverSoapBinding"
            name="UCMDBDataReceiver">
                <wsdlsoap:address location="http://localhost:8080/UCMDBDataReceiver/services/
                UCMDBDataReceiver"/>
            </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

응답 처리

데이터 수신기는 addDataResponse, deleteDataResponse 또는 updateDataResponse 구조로 문자열을 반환해야 합니다. 어댑터는 처리되지 않은 응답 데이터를 프로브의 probeMgr-adaptersDebug.log로 전달합니다. 수신기는 문자열 데이터를 반환할 수 있으며 응답은 SOAP 준수 XML로 래핑됩니다. Jython 스크립트에서 SOAPMessage 및 관련 Java 클래스를 사용하여 응답 메시지의 구문을 분석할 수 있습니다. 다음은 데이터 수신기에서 보내는 응답 메시지의 예입니다.

```
<2012-03-16 15:47:38,080> [INFO ] [Thread-110] - XMLtoWebService.py:addData received response:
<soapenv:Body xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<intf:addDataResponse xmlns:intf="http://ucmdb.hp.com">
    <xml>&lt;result&gt;&lt;status&gt;error&lt;/status&gt;
    &lt;message&gt;Error publishing config item changes&lt;/message&gt;
    &lt;/result&gt;</xml>
</intf:addDataResponse>
</soapenv:Body>
```

위에 표시된 메시지는 **<Error publishing config item changes>**라는 오류 메시지이지만, 컨텐츠는 데이터 수신기가 응답하도록 설계된 어느 것이든 될 수 있습니다. 여기서 응답은 오류 메시지이며, 이것은 단순히 의도된 것으로 설계자는 밀어넣기 어댑터가 성공 또는 실패를 나타내는 것을 응답으로 예상한다고 설명합니다. 컨텐츠는 성공적으로 추가된 모든 CI의 조정 ID 또는 특정 CI의 경우 오류 메시지일 수 있습니다. GWSPA의 사용자 지정에는 응답 메시지의 구문 분석 및 일부 CI 다시 보내기, 기타 로깅 수행 등의 작업이 포함될 수 있습니다.

WSDL 테스트

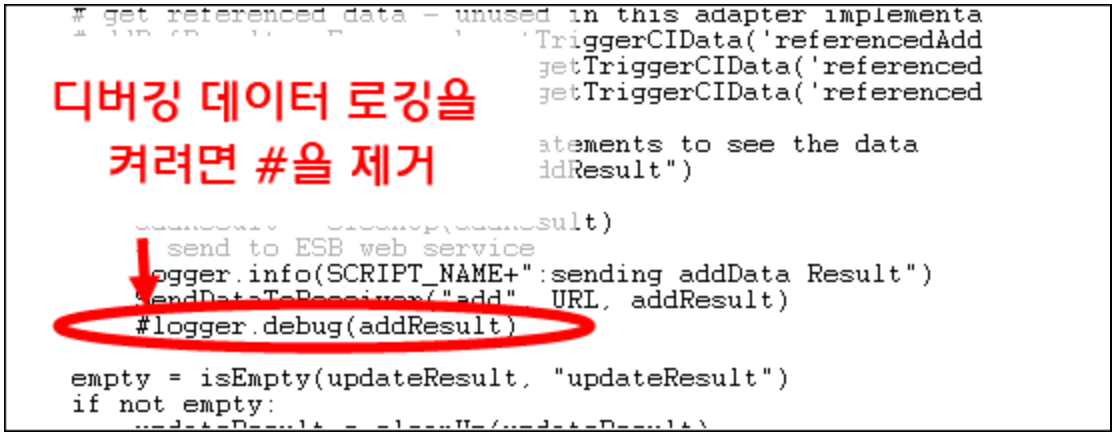
SOAPUI Eclipse 플러그인은 개발 중에 웹 서비스 레이어를 테스트하는 데 사용됩니다. SOAPUI를 사용하면 웹 서비스를 사용자 지정하는 데 도움이 될 수 있습니다. SOAPUI는 SOAP 메시지 작성, 보내기 및 받기를 테스트할 수 있는 IDE(통합 개발 환경)를 제공합니다. SOAPUI 퍼스펙티브에서 230-234페이지의 WSDL은 다음과 같은 샘플 메시지를 생성했습니다.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ucm="http://ucmdb.hp.com">
    <soapenv:Header/>
    <soapenv:Body>
        <ucm:addData>
            <ucm:xmlAdded>?</ucm:xmlAdded>
        </ucm:addData>
    </soapenv:Body>
</soapenv:Envelope>
```

위 **xmlAdded** 요소의 "**?**"는 웹 서비스 밀어넣기 어댑터 통합에서 제공한 데이터의 위치입니다.

결과 관찰

밀어넣기 어댑터가 디버그 이외의 모드에서 정상적으로 작동할 경우 데이터는 최종 결과가 기록될 때까지 파일에 기록되지 않습니다. (중간 TQL 결과와 매핑된 데이터 결과는 일반적으로 로그 파일에 표시되지 않습니다.) 그러나 아래 표시된 대로 DiscoveryMain 섹션에서 **logger.debug** 문의 주석 처리를 제거("#" 문자 제거)하여 프로브의 디버그 파일에 결과를 쓸 수 있습니다.



logger 문이 다른 이전 줄 및 다음 줄과 동일한 열에서 시작하도록 하십시오. Jython은 들여쓰기에 민감하므로 모든 줄의 들여쓰기가 올바르지 않으면 스크립트가 실패합니다.

여기서 프로브의 디버그 로그 파일 **probeMgr-adaptersDebug.log**는 다음과 같은 출력 내용을 표시합니다.

<2011-12-07 14:02:23,019> [INFO] [Thread-273] - XMLtoWebService.py started

<2011-12-07 14:02:23,019> [DEBUG] [Thread-273] - ESB Push parameters:

<2011-12-07 14:02:23,019> [DEBUG] [Thread-273] - Wshost=harpy.trtc.com

<2011-12-07 14:02:23,019> [DEBUG] [Thread-273] - WShostport=5555

<2011-12-07 14:02:23,019> [DEBUG] [Thread-273] - WSuri=ws/DtITServiceManagement.esla.v1.ws.provider:UMDBDataReceiver

<2011-12-07 14:02:23,019> [INFO] [Thread-273] - URL is http://harpy.trtc.com:5555/ws/DtITServiceManagement.esla.v1.ws.provider:UMDBDataReceiver

<2011-12-07 14:02:23,035> [DEBUG] [Thread-273] - Connected to http://harpy.trtc.com:5555/ws/DtITServiceManagement.esla.v1.ws.provider:UMDBDataReceiver

<2011-12-07 14:02:23,035> [ERROR] [Thread-273] - sending results

<2011-12-07 14:02:23,035> [DEBUG] [Thread-273] - <?xml version="1.0" encoding="UTF-8"?>

<root>

<data>

<objects>

<Object mode="" name="u_imp_ip_switch" operation="add" mamId="9e8c2f6bdfe4b7d0864c79e70833902c">

<field name="Correlation ID" key="true" datatype="char" length="">9e8c2f6bdfe4b7d0864c79e70833902c</field>

```
                <field name="name" key="false" datatype="char" length="">nma_
                09sw</field>
                <field name="location" key="false" datatype="char" length="" />
                <field name="u_chassis_vendor_type" key="false" datatype="char"
                length="">ciscoCat2960-24TT</field>
                <field name="serial_number" key="false" datatype="char" length="" />
                <field name="ram" key="false" datatype="char" length="" />
                <field name="os_version" key="false" datatype="char" length="" />
            </Object>
```

Jython 스크립트 수정

XMLtoWebService.py

웹 서비스 밀어넣기 어댑터가 사용하는 Jython 스크립트는 XML 밀어넣기 어댑터와 매우 비슷합니다. 이 스크립트는 어댑터에 포함된 **UCMDBDataReceiver.jar**을 사용합니다. 이 스크립트는 **SendDataToReceiver()** 메서드를 구현합니다. **SendDataToReceiver()**는 다음 세 가지 매개 변수를 사용합니다.

1. 수행(add, update 또는 delete)
2. 데이터 수신기의 URL
3. 데이터

예를 들어 add 블록은 **SendDataToReceiver("add", URL, addResult)**와 같습니다.

모든 웹 서비스와 SOAP 레이어가 래핑됩니다. URL은 UCMDB 데이터 수신기의 서비스 끝점 주소입니다. 이 URL은 "?wsdl" 접미사를 통해 wsdl을 가져오는 데 사용되는 URL과 동일합니다.

아래에는 Jython 스크립트의 원본이 나와 있습니다. 웹 서비스 통합 래퍼 줄은 녹색으로 강조 표시되어 있습니다.

```
#####################################

# script: XMLtoWebService.py

#####################################

# This jython script accepts TQL data results (adds, updates, and deletes) from the Integration
adapter.

# and sends it to a web service. The web service is called UCMDBDataReceiver.

# A web service client of this name must be addressable at the URL provided by the parameters.

# The SendDataToReceiver.jar exposes the SendDataToReceiver function, as well as the service
locator.
```

```
# examples of the service locator are in the testconnection section.
# regular expressions
import re
# logging
import logger
# web service interface
from com.hp.ucmdb import SendDataToReceiver
from com.hp.ucmdb.SendDataToReceiver import locateService
from com.hp.ucmdb.SendDataToReceiver import SendData
##################################################
########     VARIABLES          ##########
##################################################
SCRIPT_NAME = "XMLtoWebService.py"
logger.info(SCRIPT_NAME+" started")
def cleanUp(str):

        # replace mode=""
        str = re.sub("mode=\"\w+\"\s+", "", str)

        # replace mamId with id
        str = re.sub("\smamId=\"", " id=\"", str)

        # replace empty attributes
        str = re.sub("[\n|\s|\r]*<field name=\"\w+\" datatype=\"\w+\" />", "", str)

        # replace targetRelationshipClass with name
        str = re.sub("\stargetRelationshipClass=\"", " name=\"", str)

        # replace Object with object with name
        str = re.sub("<Object mode=\"", "<object mode=\"", str)
        str = re.sub("<Object operation=\"", "<object operation=\"", str)
```

```
        str = re.sub("<Object name=\"", "<object name=\"", str)
        str = re.sub("</Object>", "</object>", str)

        # replace field to attribute
        str = re.sub("<field name=\"", "<attribute name=\"", str)
        str = re.sub("</field>", "</attribute>", str)

        #logger.debug("String = %s" % str)
        #logger.debug("cleaned up")

        return str
def isEmpty(xml, type = ""):
        objectsEmpty = 0
        linksEmpty = 0

        m = re.findall("<objects />", xml)
        if m:
                #logger.warn("\t[%s] No objects found" % type)
                objectsEmpty = 1

        m = re.findall("<links />", xml)
        if m:
                #logger.warn("\t[%s] No links found" % type)
                linksEmpty = 1

        if objectsEmpty and linksEmpty:
                return 1
        return 0
##################################################
########    MAIN            ###########
##################################################
```

```
def DiscoveryMain(Framework):
        #fix this for web service export
        errMsg = "UCMDBDataReceiver Service not found."
        testConnection = Framework.getTriggerCIData("testConnection")
        # Get Web Service Push variables
        WShostName = Framework.getTriggerCIData("Host Name")
        WShostport = Framework.getTriggerCIData("Protocol Port")
        WSuri = Framework.getTriggerCIData("URI")

        logger.info(SCRIPT_NAME+":ESB Push parameters:")
        logger.info("Host Name="+WShostName)
        logger.info("Protocol Port="+WShostport)
        logger.info("URI="+WSuri)
        URL = "http://"+WShostName+":"+WShostport+"/"+WSuri
        logger.info("URL="+URL)
        if testConnection == 'true':
                # locate the service
                test_receiver = SendDataToReceiver()
                locator = test_receiver.locateService(URL)
                #locator = locateService(URL)
                if(locator):
                        logger.info(SCRIPT_NAME+":Test connection was successful")
                        return
                else:
                        raise Exception, errMsg
                        return
        # do same thing here if not just a test connection -
        receiver = SendDataToReceiver()
        locator = receiver.locateService(URL)
        if(locator):
                logger.info(SCRIPT_NAME+":Connected to "+URL)
```

```
else:
        logger.error(SCRIPT_NAME+":no locator")
        raise Exception, errMsg
        return

# get add/update/delete result objects from the Framework
addResult = Framework.getTriggerCIData('addResult')
updateResult = Framework.getTriggerCIData('updateResult')
deleteResult = Framework.getTriggerCIData('deleteResult')
logger.debug(deleteResult)

# get referenced data - unused in this adapter implementation
#addRefResult = Framework.getTriggerCIData('referencedAddResult')
#updateRefResult = Framework.getTriggerCIData('referencedUpdateResult')
#deleteRefResult = Framework.getTriggerCIData('referencedDeleteResult')
# uncomment out the logger statements to see the data
empty = isEmpty(addResult, "addResult")
if not empty:
        addResult = cleanUp(addResult)
        # send to ESB web service
        logger.info(SCRIPT_NAME+":sending addData Result")
        rcvr = SendDataToReceiver()
        resp = rcvr.SendData("add", URL, addResult)
        logger.info(SCRIPT_NAME+":addData received response:"+resp)
        #logger.debug(addResult)
empty = isEmpty(updateResult, "updateResult")
if not empty:
        updateResult = cleanUp(updateResult)
        # send to ESB web service
        #logger.debug(updateResult)
        logger.info(SCRIPT_NAME+":sending updateData Result")
```

```
        rcvr = SendDataToReceiver()
        resp = rcvr.SendData("update", URL, updateResult)
        logger.info(SCRIPT_NAME+":received response:"+resp)


empty = isEmpty(deleteResult, "deleteResult")
if not empty:
        deleteResult = cleanUp(deleteResult)
        # send to ESB web service
        #logger.debug(deleteResult)
        logger.info(SCRIPT_NAME+":sending deleteData Result")
        rcvr = SendDataToReceiver()
        resp = rcvr.SendData("delete", URL, deleteResult)
        logger.info(SCRIPT_NAME+":received response:"+resp)
logger.info(SCRIPT_NAME+" ended")
```

응답 메시지 처리 사용자 지정

데이터 수신기는 응답 또는 원하는 상태가 포함된 문자열을 반환해야 합니다. 웹 서비스 밀어넣기 어댑터는 기본적으로 응답을 프로브의 정보 수준 로그로 전달합니다. 응답 메시지는 반환된 응답 문자열이 포함된 SOAP 형식의 XML입니다. 수신기는 그룹화되었거나 개별 오류 또는 성공 메시지 등의 모든 데이터를 반환할 수 있습니다. 추가 처리가 필요한 경우 어댑터의 Jython 스크립트를 통해 응답을 처리할 수 있습니다. Java 프로그래밍은 필요하지 않습니다.

다음을 사용하여 보낸 반환 응답 메시지의 예는 그 아래에 나와 있습니다.

```
// stub example for building your own UCMDBDataData Receiver
public class UCMDBDataReceiver {


public String addData (String xmlAdd){
System.out.println(xmlAdd); // do something with the data
// send back a response message based on what you did
String tr = new String("a test response from addData!");
return tr;
}
```

즉, 아래와 같습니다.

```
<soapenv:Body xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
    <addDataResponse xmlns="http://ucmdb.hp.com">
        <addDataReturn>a test response from addData!</addDataReturn>
    </addDataResponse>
</soapenv:Body>
```

데이터 수신기 수정

Java 클라이언트는 **UCMDBDataReceiver.jar**에 포함된 클래스를 구현하고 Jython과 동일한 방식으로 웹 서비스를 호출할 수 있습니다. 또한 래핑되지 않은 메서드를 호출할 수도 있습니다. **UCMDBDataReceiver.jar** 클래스에 대해서는 Javadoc가 있습니다. 아래 원본 코드는 이러한 필수 메서드를 사용하여 데이터를 SOAP 메시지로 래핑한 후 HTTP를 통해 수신기로 보내는 방법을 보여 줍니다.

이 프로세스는 **UCMDBDataReceiverServiceLocator** 개체를 만든 다음 **UCMDBDataReceiverEndPointAddress**를 데이터 수신기의 URL에 할당하는 단계로 진행됩니다.

데이터를 보내기 위해 로케이터의 **getUCMDBDataReceiver** 메서드가 호출되어 **UCMDBDataReceiver** 개체를 만듭니다. **UCMDBDataReceiver** 개체는 실제로 추가/변경/삭제 데이터를 보내는 메서드를 구현합니다. 각 요청 유형을 처리하기 위한 세 가지 동일한 코드 블록이 있습니다.

SendDataToReceiver 클래스의 원본 코드는 아래에 나열되어 있습니다. 강조 표시된 개체와 메서드는 필수적으로 사용해야 할 요소입니다.

```
/**
* Test SendData for the UCMDB Data Receiver for the UCMDB Web Service Push Adapter
*/
package com.hp.ucmdb;
import com.hp.ucmdb.SendDataToReceiver;
/**
* TestSendData can be used to verify the SOAP classes are working.
* TestSendData creates a SendDataToReceiver class and invokes its SendData method.
* a response String is returned.
* The test URL is typically appended with "?wsdl" to get the WSDL of the service.
*/
public class TestSendData {
    /**
    * @param args - test SOAP message.
```

```
*  optional arguments [0] a test string [1] a service endpoint URL of a Data Receiver.
* the default URL is sent the incoming argument as a test message.
* the default URL is "http://localhost:8080/UCMDBDataReceiver/services/UCMDBDataReceiver".
* If any errors are encountered, TestClient will attempt to throw exceptions.
*/
public static void main(String[] args) {
        // use test message if supplied, otherwise supply a default test string
        String teststring = new String("Test SOAP message from UCMDBDataReceiver
        TestSendData.");
        if(args.length > 0) {
                teststring = args[0];
        }
        // use test URL if supplied, otherwise supply the default URL
        String URL = new String("");
        if(args.length > 1) {
                URL = args[1];
        }
        // return response
        String response = new String("");
        // perform the tests
        try {
                if(URL.equals("")) {
                        UCMDBDataReceiverServiceLocator locator = new
                        UCMDBDataReceiverServiceLocator();
                        UCMDBDataReceiver receiver = locator.getUCMDBDataReceiver();
                        URL = locator.getUCMDBDataReceiverAddress();
                        System.out.println("TestClient: tested
                        URL="+locator.getUCMDBDataReceiverAddress());
                        System.out.println("TestClient: receiver="+receiver.toString());
                }
                SendDataToReceiver sdtr = new SendDataToReceiver();
                // this sends a test push and gets a response message
```

```
                response = sdtr.SendData("add", URL, args[0]);
                System.out.println("Response received was:"+response);
            } catch(Exception e){
                System.out.println("TestClient: Remote Error:");
                e.printStackTrace();
            }
        }
}
```

원본 코드는 또한 다른 클래스의 **UCMDBDataReceiver.jar** 파일에도 포함되어 있습니다.

- TestClient.java
- UCMDBDataReceiver.java
- UCMDBDataReceiverProxy.java
- UCMDBDataReceiverService.java
- UCMDBDataReceiverServiceLocator.java
- UCMDBDataReceiverSoapBindingStub.java

원본은 Eclipse IDE에서 생성된 후 수정되었습니다. UCMDB 코드의 대부분은 SOAP 사양 및 UCMDB 데이터 수신기와 일치하도록 자동 생성되었으므로 이 코드를 수정할 때는 주의해야 합니다.

Javadoc

일반 웹 서비스 밀어넣기 어댑터에는 전체 주석 처리된 **javadoc**가 제공됩니다. **javadoc**는 docs 폴더 **javadoc**에 포함되어 있습니다. **index.html**부터 시작합니다. 개요 페이지에서는 SDK의 모든 클래스 및 메서드에 대한 설명서에 액세스할 수 있습니다.

모든 클래스

- **SendDataToReceiver:** 웹 서비스 래퍼의 API입니다.
- **TestClient:** 클라이언트를 테스트하여 서비스 끝점에 대한 연결을 확인합니다.
- **UCMDBDataReceiver:** 웹 서비스 래퍼입니다.

나머지는 웹 서비스 작성기에 의해 자동으로 생성됩니다.

- UCMDBDataReceiverProxy
- UCMDBDataReceiverService
- UCMDBDataReceiverServiceLocator
- UCMDBDataReceiverSoapBindingStub

개요

원본 코드 예제를 포함한 SDK의 기본 사용법에 대해서는 패키지의 설명서에 설명되어 있습니다. 이 **javadoc**는 UCMDB 웹 서비스 밀어넣기 어댑터용입니다. Jython 또는 Java에서 API를 호출할 수 있습니다.

SDK는 **TestClient**, **SendDataToReceiver** 등의 두 가지 원본 샘플을 제공합니다. **TestClient**는 응답하는 로컬 클라이언트에 대해 매우 제한된 테스트를 제공합니다. **SendDataToReceiver**는 데이터를 웹 서비스로 보내는 데 사용되는 기본 클래스입니다.

먼저 이 SDK(주로 동봉된 WSDL)를 사용하여 UCMDB 데이터 수신기를 구현하여 이 웹 서비스와 통신합니다. 그런 다음 이 SDK를 사용하여 UCMDB에 밀어넣기 어댑터를 만들어 UCMDB TQL 결과 데이터를 데이터 수신기에 밀어 넣습니다. 아래에서는 Jython 및 Java 구현을 모두 포함하여 이 API의 기본 사용법에 대해 설명합니다.

SendDataToReceiver() 구현

SendDataToReceiver()는 다음과 같이 단일 메서드를 사용하여 모든 기능을 래핑합니다.

- Jython: SendDataToReceiver("add",yourURL,"Hello!")
- Java: SendDataToReceiver("add",yourURL,"Hello!");

또는 아래 표시된 대로 **SendDataToReceiver** 개체를 만든 다음(예: 다른 설정을 조작하려는 경우) 별도로 **SendData** 메서드를 호출합니다.

- Jython:

  rcvr = SendDataToReceiver()

  responseMsg = rcvr.SendData(“add”, yourURL, “Hello!”)

- Java:

  SendDataToReceiver rcvr = new SendDataToReceiver();

  String responseMsg = rcvr.SendData(“add”, yourURL, “Hello!”);

또는 한 번에 하나씩 해야 할 경우에는 다음과 같이 할 수 있습니다.

1. 아래와 같이 새 **UCMDBDataReceiverServiceLocator()** 개체 x를 만든 다음 나중에 개체의 끝점 주소를 설정합니다.
   - **Jython:**

     x = UCMDBDataReceiverServiceLocator()

     x.setUCMDBDataReceiverEndPointAddress(URL)

   - **Java :**

     UCMDBDataReceiverServiceLocator x = new UCMDBDataReceiverServiceLocator();

     x. setUCMDBDataReceiverEndPointAddress(URL);

2. 그런 후 다음을 사용하여 UCMDBDataReceiver를 만듭니다.
   - **Jython:** y = x.getUCMDBDataReceiver()
   - **Java:** UCMDBDataReceiver y = x.getUCMDBDataReceiver();

3. 그런 후 다음과 같이 SOAP 웹 서비스를 통해 데이터를 보냅니다.
   - **Jython:**
     - y.addData(yourData)
     - 또는 y.updateData(yourData)
     - 또는 y.deleteData(yourData)
   - **Java:**
     - y.addData(yourData);
     - 또는 y.updateData(yourData);
     - 또는 y.deleteData(yourData);
4. 연결을 테스트해야 할 수 있습니다. 테스트에 성공할 경우 동일한 로케이터 개체를 다시 사용하여 데이터 전송에 사용할 **UCMDBDataReceiver**를 반환할 수 있습니다.

클래스에는 소멸자가 포함되어 있지 않으며 메모리 관리를 수행하지 않습니다.

매핑 파일 참조

매핑 사용

변환된 XML 출력의 각 대상 특성에 대해 매핑을 만들어야 합니다. 매핑은 데이터를 가져올 위치와 방법을 지정합니다. 데이터가 UCMDB의 다른 해당 특성에 있는 경우에는 직접 매핑이 사용됩니다.

여러 특성에서 데이터를 끌어오거나 UCMDB CI의 하위 또는 상위 CI의 특성에서 특성을 끌어오려면 다른 복잡한 매핑이 필요할 수 있습니다. 아래의 매핑 스키마는 가능한 모든 매핑을 보여 줍니다.

매핑 파일은 UCMDB의 어떤 CI/관계 유형이 대상 데이터 저장소의 어떤 CI/관계 유형으로 매핑되는지를 정의하는 XML 파일입니다. 형식에 대해서는 아래에서 자세히 설명합니다. 매핑 파일은 밀어 넣을 CI 및 관계 유형을 제어할 뿐 아니라 정확히 어떤 특성을 밀어 넣을지 제어합니다.

대상 MDR로 밀어 넣을 각 특성에 대해 매핑 항목이 있습니다. 각 매핑 항목은 원시 UCMDB 밀어넣기 데이터에 있는 하나 이상의 특성으로 구성될 수 있습니다. 매핑 항목은 대상 MDR로 밀어 넣을 데이터의 최종 구조 및 이름 지정을 완전히 세부적으로 제어할 수 있게 해 줍니다.

직접 매핑

매핑은 데이터 모델을 변환합니다(이 경우 UCMDB를 밀어넣기 대상 MDR로 변환). UCMDB 특성과 대상 간의 관계가 1:1 관계인 경우 이름과 유형만 다르기 때문에 변환 작업이 간단할 수 있습니다.

대부분의 특성 매핑은 직접 매핑입니다. 예를 들어 서버 이름 "ServerX"는 UCMDB에서 길이가 50이고 유형이 **string**이며 특성 이름이 **primary_server_name**인 **unix** 유형의 CI로 표시될 수 있습니다. 대상 MDR의 데이터 모델은 최대 길이가 250이고 유형이 **char[]**이며 특성 이름이 **hostname**인 CI 유형 **linux**와 동일한 논리적 엔터티를 지정할 수 있습니다. 직접 매핑은 이와 같은 모든 유형의 변환 작업을 수행할 수 있습니다.

다음은 직접 매핑의 예입니다.

```
<target_attribute name="dns_domain" datatype="char">
<map type="direct" source_attribute="domain_name" />
</target_attribute>
```

이 직접 매핑은 UCMDB 특성 **dns_domain**을 대상 데이터 모델의 **domain_name** 특성에 매핑합니다.

실제 데이터 유형을 사용해야 하는 경우가 아니면 실제 데이터 유형에 관계없이 **char** 데이터 유형을 사용합니다.

복잡한 매핑

좀 더 복잡한 매핑을 통해 다음과 같은 변환을 추가로 수행할 수 있습니다.

- 여러 CI의 특성 값을 하나의 대상 CI에 매핑
- 하위 CI(**container_f** 또는 포함된 관계가 있는 CI)의 특성을 대상 데이터 저장소의 상위 CI에 매핑(예: 대상 호스트 CI에서 **Number of CPUs** 값 설정). 또 다른 예로는 UCMDB에 있는 호스트 CI의 모든 메모리 CI의 메모리 크기 값을 더하여 대상 호스트 CI에서 **Total Memory** 값을 설정하는 것을 들 수 있습니다.
- 상위 CI(**container_f** 또는 포함된 관계가 있는 CI)의 특성을 대상 데이터 저장소의 CI에 매핑(예: UCMDB에 있는 소프트웨어 CI의 포함 호스트에서 값을 가져와서 **설치된 소프트웨어** CI라는 대상 특성에서 **Container Server** 값 설정).

다음은 쉼표 문자로 구분된 두 개의 원본 특성을 사용하여 대상 특성 "**os**"를 만드는 복잡한 매핑의 예입니다.

```
<target_attribute name="os" datatype="char">
    <map type="compoundstring">
        <source_attribute name="discovered_os_name" />
        <constant value="," />
        <source_attribute name="host_osinstalltype" />
    </map>
</target_attribute>
```

링크 방향 바꾸기

UCMDB에는 원본 간에 구조가 다른 데이터가 포함되어 있을 수 있습니다. 예를 들어 IpAddress CI와 Interface CI 간의 관계는 **parent**일 수 있으며 이러한 관계는 HP Network Node Manager 통합에서 발생할 수 있습니다. 또는 Universal Discovery에서 일반적으로 생성되는 **containment** 링크일 수 있습니다. 또한 이러한 링크의 방향은 서로 반대 방향입니다.

현재는 매핑 파일에서 링크의 방향을 반대로 바꿀 수 없습니다. **_end1** 및 **_end2** 변수를 뒤바꾸면 변환된 XML에서 데이터의 순서가 바뀌거나 원본 데이터에 링크가 누락됩니다.

이 문제를 해결할 수 있는 한 가지 가능한 방법은 다음과 같이 엔리치먼트 규칙을 정의하는 것입니다.

1. 엔리치먼트의 TQL 부분은 밀어넣기 어댑터에서 사용하는 TQL의 하위 집합입니다. 이 TQL은 특히 변환된 XML에서 기대되는 것과 반대 방향의 링크를 모두 선별합니다.

2. 엔리치먼트 부분은 올바른 방향 및 원하는 유형의 새 링크를 정의합니다.
3. 엔리치먼트가 활성화된 다음 올바른 링크를 만듭니다.
4. 통합 작업 TQL은 이제 원본 링크가 아닌 향상된 링크를 참조합니다.
5. 그런 다음 밀어넣기 어댑터의 <link> 매핑에서도 향상된 링크를 참조하고 유형 및 방향이 일관된 링크 집합을 생성합니다.

매핑 파일 스키마

요소 이름 및 경로	설명	특성
integration	파일의 매핑 컨텐츠를 정의합니다. 시작 줄 및 주석을 제외하고 파일에서 가장 바깥쪽 블록이어야 합니다.	
info (integration)	통합할 데이터 저장소에 대한 정보를 정의합니다.	
source (integration > info)	원본 데이터 저장소에 대한 정보를 정의합니다.	1. **이름:** type **설명:** 원본 데이터 저장소의 이름입니다. **필수 여부:** 필수 **유형:** 문자열 2. **이름:** versions **설명:** 원본 데이터 저장소의 버전입니다. **필수 여부:** 필수 **유형:** 문자열 3. **이름:** vendor **설명:** 원본 데이터 저장소의 벤더입니다. **필수 여부:** 필수 **유형:** 문자열
target (integration > info)	대상 데이터 저장소에 대한 정보를 정의합니다.	1. **이름:** type **설명:** 원본 데이터 저장소의 이름입니다. **필수 여부:** 필수 **유형:** 문자열 2. **이름:** versions **설명:** 원본 데이터 저장소의 버전입니다. **필수 여부:** 필수 **유형:** 문자열 3. **이름:** vendor

요소 이름 및 경로	설명	특성
		설명: 원본 데이터 저장소의 벤더입니다. **필수 여부:** 필수 **유형:** 문자열
targetcis (integration)	모든 CIT 매핑의 컨테이너 요소입니다.	
source_ci_type_tree (integration > targetcis)	원본 CIT와 원본에서 상속하는 모든 CI 유형을 정의합니다.	1. **이름:** name **설명:** 원본 CIT의 이름입니다. **필수 여부:** 필수 **유형:** 문자열 2. **이름:** mode **설명:** 현재 CI 유형에 필요한 업데이트 유형입니다. **필수 여부:** 필수 **유형:** 다음 문자열 중 하나입니다. a. **insert:** CI가 아직 없는 경우에만 사용합니다. b. **update:** CI가 있는 경우에만 사용합니다. c. **update_else_insert:** CI가 있는 경우 업데이트하고, 그렇지 않으면 새 CI를 만듭니다. d. **ignore:** 이 CI 유형으로는 아무 작업도 수행하지 않습니다.
source_ci_type (integration > targetcis)	원본에서 상속하는 CI 유형 없이 원본 CIT를 정의합니다.	1. **이름:** name **설명:** 원본 CIT의 이름입니다. **필수 여부:** 필수 **유형:** 문자열 2. **이름:** mode **설명:** 현재 CI 유형에 필요한 업데이트 유형입니다. **필수 여부:** 필수 **유형:** 다음 문자열 중 하나입니다. a. **insert:** CI가 아직 없는 경우에만 사용합니다. b. **update:** CI가 있는 경우에만 사용합니다. c. **update_else_insert:** CI가 있는 경우 업데이트하고, 그렇지 않으면 새 CI를 만듭니다. d. **ignore:** 이 CI 유형으로는 아무 작업도 수

요소 이름 및 경로	설명	특성
		행하지 않습니다.
target_ci_type (integration > targetcis > source_ci_type -또는- integration > targetcis > source_ci_type_tree)	대상 CIT를 정의합니다.	1. **이름:** name **설명:** 대상 CI 유형 이름입니다. **필수 여부:** 필수 **유형:** 문자열 2. **이름:** schema **설명:** 이 CI 유형을 대상에 저장하는 데 사용할 스키마의 이름입니다. **필수 여부:** 필수 아님 **유형:** 문자열 3. **이름:** namespace **설명:** 대상에서 이 CI 유형의 네임스페이스를 나타냅니다. **필수 여부:** 필수 아님 **유형:** 문자열
targetprimarykey (integration > targetcis > source_ci_type) -또는- (integration > targetcis > source_ci_type_tree -또는- (integration > targetrelations > link) -또는- (integration > targetrelations > source_link_type_tree)	대상 CIT 기본 키 특성을 식별합니다.	
pkey (integration > targetcis> source_ci_type > targetprimarykey -또는-	기본 키 특성 하나를 식별합니다. 모드가 **update** 또는 **insert_else_update**인 경우에만 필요합니다.	

요소 이름 및 경로	설명	특성
integration > targetcis > source_ci_type_tree > targetprimarykey -또는- (integration > targetrelations > link > targetprimarykey) -또는- integration > targetrelations > source_link_type_tree > targetprimarykey)		
target_attribute (integration > targetcis> source_ci_ type -또는- integration > targetcis > source_ci_type_tree -또는- integration > targetrelations > link -또는- integration > targetrelations > source_link_type_tree)	대상 CIT의 특성을 정의합니다.	1. **이름:** name **설명:** 대상 CIT의 특성 이름입니다. **필수 여부:** 필수 **유형:** 문자열 2. **이름:** datatype **설명:** 대상 CIT 특성의 데이터 유형입니다. **필수 여부:** 필수 **유형:** 문자열 3. **이름:** length **설명:** 문자열/문자 데이터 유형의 경우 대상 특성의 정수 크기입니다. **필수 여부:** 필수 아님 **유형.** 정수 4. **이름**. option **설명.** 값에 적용할 변환 함수입니다. **필수 여부.** False **유형.** 다음 문자열 중 하나입니다. a. **uppercase** –대문자로 변환 b. **lowercase** –소문자로 변환 이 특성이 비어 있으면 변환 함수가 적용되지 않습니다.
map (integration > targetcis > source_ci_	원본 CIT의 특성 값을 가져오는 방법을 지정합니다.	1. **이름**. type **설명.** 원본 값과 대상 값 사이의 매핑 유형입니다.

요소 이름 및 경로	설명	특성
type > target_attribute -또는- integration > targetcis > source_ci_type_tree > target_attribute) -또는- (integration > targetrelations > link > target_attribute -또는- integration > targetrelations > source_link_type_tree > target_attribute)		**필수 여부.** 필수 **유형.** 다음 문자열 중 하나입니다. a. **direct** –원본 특성의 값에서 대상 특성의 값으로 일대일 매핑을 지정합니다. b. **compoundstring** –하위 요소가 문자열 하나에 조인되고 대상 특성 값이 설정됩니다. c. **childattr** –하위 요소가 하나 이상의 하위 CIT의 특성입니다. 하위 CIT는 **composition** 또는 **containment** 관계가 있는 CIT로 정의됩니다. d. **constant** –정적 문자열 2. **이름**. value **설명.** type=**constant**에 대한 상수 문자열입니다. **필수 여부.** type=**constant**일 때만 필요합니다. **유형.**문자열 3. **이름**. attr **설명.** type=**direct**에 대한 원본 특성 이름입니다. **필수 여부.** type=**direct**일 때만 필요합니다. **유형.**문자열
aggregation (integration > targetcis > source_ci_type > target_attribute > map -또는- integration > targetcis > source_ci_type_tree > target_attribute > map -또는- (integration > targetrelations > link > target_attribute >	원본 CI의 하위 CI 특성 값을 대상 CI 특성에 매핑할 하나의 값으로 결합하는 방법을 지정하며, 선택 사항입니다.	**이름:** type **설명.** 집계 함수의 유형입니다. **필수 여부:** 필수 **유형.** 다음 문자열 중 하나입니다. • **csv** –포함된 모든 값을 쉼표로 구분된 목록(숫자 또는 문자열/문자)에 연결합니다. • **count** –포함된 모든 값의 개수를 숫자로 반환합니다. • **sum** –포함된 모든 숫자 값의 합계를 반환합니다. • **average** –포함된 모든 값의 평균을 숫자로 반환합니다. • **min** –포함된 값의 최소값을 숫자/문자로 반환합니다. • **max** –포함된 값의 최대값을 숫자/문자로 반환합니다.

요소 이름 및 경로	설명	특성
map -또는- integration > targetrelations > source_link_type_tree > target_attribute > map) 맵의 유형이 **childattr**일 때만 유효		
source_child_ci_type (integration > targetcis> source_ci_type > target_attribute > map -또는- integration > targetcis > source_ci_type_tree > target_attribute > map -또는- (integration > targetrelations > link > target_attribute > map -또는- integration > targetrelations > source_link_type_tree > target_attribute > map) 맵의 유형이 **childattr**일 때만 유효	하위 특성을 가져오는 연결된 CI를 지정합니다.	1. **이름**. name **설명.** 하위 CI의 유형입니다. **필수 여부.** 필수 **유형.**문자열 2. **이름.** source_attribute **설명.** 매핑되는 하위 CI의 특성입니다. **필수 여부.** 동일한 경로에 있는 **childAttr** 집계 유형이 count가 아닐 경우에만 필요합니다. **유형.**문자열
validation (integration > targetcis > source_ci_type > target_attribute > map	특성 값에 기반한 원본 CI의 하위 CI를 필터링으로 제외할 수 있습니다. 집계 하위 요소와 함께 사용하여 대상 CIT의 특성	1. **이름**. minlength **설명.** 주어진 값보다 짧은 문자열을 제외합니다. **필수 여부:** 필수 아님 **유형.** 정수

요소 이름 및 경로	설명	특성
-또는- integration > targetcis > source_ci_type_tree > target_attribute > map -또는- (integration > targetrelations > link > target_attribute > map -또는- integration > targetrelations > source_link_type_tree > target_attribute > map) 맵의 유형이 **childatt**일 때만 유효	값에 매핑되는 하위 특성을 면밀히 파악할 수 있으며, 선택 사항입니다.	2. **이름**. maxlength **설명.** 주어진 값보다 긴 문자열을 제외합니다. **필수 여부:** 필수 아님 **유형.** 정수 3. **이름**. minvalue **설명.** 지정된 값보다 작은 숫자를 제외합니다. **필수 여부:** 필수 아님 **유형.** 숫자 4. **이름**. maxvalue **설명.** 지정된 값보다 큰 숫자를 제외합니다. **필수 여부:** 필수 아님 **유형.** 숫자
targetrelations (integration)	모든 관계 매핑의 컨테이너 요소이며, 선택 사항입니다.	
source_link_type_tree (integration > targetrelations)	원본에서 상속되는 유형 없이 원본 관계 유형을 대상 관계에 매핑합니다. **targetrelation**이 있는 경우에만 필수입니다.	1. **이름:** name **설명.** 원본 관계 이름입니다. **필수 여부:** 필수 **유형.**문자열 2. **이름:** target_link_type **설명.** 대상 관계 이름입니다. **필수 여부:** 필수 **유형.**문자열 3. **이름:** nameSpace **설명:** 대상에 만들 링크의 네임스페이스입니다. **필수 여부:** 필수 아님 **유형:** 문자열 4. **이름:** mode **설명:** 현재 링크에 필요한 업데이트 유형입니다. **필수 여부:** 필수 **유형:** 다음 문자열 중 하나입니다.

요소 이름 및 경로	설명	특성
		• **insert** –CI가 아직 없는 경우에만 사용합니다. • **update** –CI가 있는 경우에만 사용합니다. • **update_else_insert** –CI가 있는 경우 업데이트하고, 그렇지 않으면 새 CI를 만듭니다. • **ignore** –이 CI 유형으로는 아무 작업도 수행하지 않습니다. 5. **이름:** source_ci_type_end1 **설명:** 원본 관계의 End1 CI 유형입니다. **필수 여부:** 필수 **유형:** 문자열 6. **이름:** source_ci_type_end2 **설명:** 원본 관계의 End2 CI 유형입니다. **필수 여부:** 필수 **유형:** 문자열
link (integration > targetrelations)	원본 관계를 대상 관계에 매핑합니다. **targetrelation**이 있는 경우에만 필수입니다.	1. **이름:** source_link_type **설명:** 원본 관계 이름입니다. **필수 여부:** 필수 **유형:** 문자열 2. **이름:** target_link_type **설명:** 대상 관계 이름입니다. **필수 여부:** 필수 **유형:** 문자열 3. **이름:** nameSpace **설명:** 대상에 만들 링크의 네임스페이스입니다. **필수 여부:** 필수 아님 **유형:** 문자열 4. **이름:** mode **설명:** 현재 링크에 필요한 업데이트 유형입니다. **필수 여부:** 필수 **유형:** 다음 문자열 중 하나입니다. • **insert** –CI가 아직 없는 경우에만 사용합니다.

요소 이름 및 경로	설명	특성
		• **update** –CI가 있는 경우에만 사용합니다. • **update_else_insert** –CI가 있는 경우 업데이트하고, 그렇지 않으면 새 CI를 만듭니다. • **ignore** –이 CI 유형으로는 아무 작업도 수행하지 않습니다. 5. **이름:** source_ci_type_end1 **설명:** 원본 관계의 End1 CI 유형입니다. **필수 여부:** 필수 **유형:** 문자열 6. **이름:** source_ci_type_end2 **설명:** 원본 관계의 End2 CI 유형입니다. **필수 여부:** 필수 **유형:** 문자열
target_ci_type_end1 (integration > targetrelations > link -또는- integration > targetrelations > source_link_type_tree)	대상 관계의 End1 CI 유형입니다.	1. **이름:** name **설명:** 대상 관계의 End1 CI 유형 이름입니다. **필수 여부:** 필수 **유형:** 문자열 2. **이름:** superclass **설명:** End1 CI 유형의 슈퍼 클래스 이름입니다. **필수 여부:** 필수 아님 **유형:** 문자열
target_ci_type_end2 (integration > targetrelations > link -또는- integration > targetrelations > source_link_type_tree)	대상 관계의 End2 CI 유형입니다.	1. **이름:** name **설명:** 대상 관계의 End2 CI 유형 이름입니다. **필수 여부:** 필수 **유형:** 문자열 2. **이름:** superclass **설명:** End2 CI 유형의 슈퍼 클래스 이름입니다. **필수 여부:** 필수 아님 **유형:** 문자열

매핑 결과 스키마

요소 이름 및 경로	설명	특성
root	결과 문서의 루트입니다.	
data (root)	데이터 자체의 루트입니다.	
objects (root > data)	업데이트할 개체의 루트 요소입니다.	
Object (root > data > objects)	단일 개체 및 모든 해당 특성에 대한 업데이트 작업을 설명합니다.	1. **이름:** name **설명:** CI 유형의 이름입니다. **필수 여부:** 필수 **유형:** 문자열 2. **이름:** mode **설명:** 현재 CI 유형에 필요한 업데이트 유형입니다. **필수 여부:** 필수 **유형:** 다음 문자열 중 하나입니다. a. **insert** –CI가 아직 없는 경우에만 사용합니다. b. **update** –CI가 있는 경우에만 사용합니다. c. **update_else_insert** –CI가 있는 경우 업데이트하고, 그렇지 않으면 새 CI를 만듭니다. d. **ignore** –이 CI 유형으로는 아무 작업도 수행하지 않습니다. 3. **이름:** operation **설명:** 이 CI에 대해 수행할 작업입니다. **필수 여부:** 필수 **유형:** 다음 문자열 중 하나입니다. a. **add** –CI를 추가해야 합니다. b. **update** –CI를 업데이트해야 합니다. c. **delete** –CI를 삭제해야 합니다. 값을 설정하지 않으면 기본값인 **추가**가 사용됩니다. 4. **이름:** mamId **설명:** 원본 CMDB의 개체 ID입니다. **필수 여부:** 필수

요소 이름 및 경로	설명	특성
		유형: 문자열
field (root > data > objects> Object -또는- root > data > links > link)	개체의 단일 필드 값을 설명합니다. 필드 텍스트는 필드에 있는 새 값이며, 필드에 링크가 포함되어 있는 경우 값은 끝 중 하나의 ID입니다. 각 끝 ID는 개체로 표시됩니다(<objects> 아래).	1. **이름:** name **설명:** 필드의 이름입니다. **필수 여부:** 필수 **유형:** 문자열 2. **이름:** key **설명:** 이 필드가 개체의 키인지 여부를 지정합니다. **필수 여부:** 필수 **유형:** 부울 3. **이름:** datatype **설명:** 필드의 유형입니다. **필수 여부:** 필수 **유형:** 문자열 4. **이름:** length **설명:** 문자열/문자 데이터 유형의 경우 대상 특성의 정수 크기입니다. **필수 여부:** 필수 아님 **유형:** 정수
links (root > data)	업데이트할 링크의 루트 요소입니다.	1. **이름:** targetRelationshipClass **설명:** 대상 시스템의 관계(링크) 이름입니다. **필수 여부:** 필수 **유형:** 문자열 2. **이름:** targetParent **설명:** 링크의 첫 번째 끝 유형입니다(상위). **필수 여부:** 필수 **유형:** 문자열 3. **이름:** targetChild **설명:** 링크의 두 번째 끝 유형입니다(하위). **필수 여부:** 필수 **유형:** 문자열 4. **이름:** mode **설명:** 현재 CI 유형에 필요한 업데이트 유형입니다. **필수 여부:** 필수

요소 이름 및 경로	설명	특성
		유형: 다음 문자열 중 하나입니다. a. **insert** –CI가 아직 없는 경우에만 사용합니다. b. **update** –CI가 있는 경우에만 사용합니다. c. **update_else_insert** –CI가 있는 경우 업데이트하고, 그렇지 않으면 새 CI를 만듭니다. d. **ignore** –이 CI 유형으로는 아무 작업도 수행하지 않습니다. 5. **이름:** operation **설명:** 이 CI에 대해 수행할 작업입니다. **필수 여부:** 필수 **유형:** 다음 문자열 중 하나입니다. a. add –CI를 추가해야 합니다. b. update –CI를 업데이트해야 합니다. c. delete –CI를 삭제해야 합니다. 값을 설정하지 않으면 기본값인 **추가**가 사용됩니다. 6. **이름:** mamId **설명:** 원본 CMDB의 개체 ID입니다. **필수 여부:** 필수 **유형:** 문자열

사용자 지정

이 섹션에서는 밀어넣기 어댑터에 대한 공통된 사용자 지정 유형의 일부 기본 절차를 설명합니다.

특성 추가

1. TQL 결과에 특성이 포함되어 있는지 확인합니다.
2. 매핑 파일의 올바른 CI 매핑 섹션에 특성 매핑을 추가합니다.
3. 데이터 수신기가 데이터의 추가 특성을 받도록 준비되어 있는지 확인합니다.

특성 제거

특성을 제거하려면 매핑 파일에서 특성을 제거합니다. 또한 결과에서 더 이상 사용되지 않거나 조건부 노드로 사용되지 않는 경우 TQL에서도 특성을 제거해야 합니다.

CI 유형 추가

1. TQL에 CI 유형을 추가합니다.
2. CI 유형과 해당 특성 데이터가 TQL 결과에 나타나는지 확인합니다(계산 및 미리 보기 사용).

3. 매핑 파일에 CI 유형의 매핑을 추가합니다. 새 CI 유형을 빠르게 만들려면 다른 CI 유형의 매핑을 복사합니다.
4. 복사한 XML의 이름 및 특성 매핑을 새 CI 유형 및 특성에 맞게 수정합니다. 사용 가능한 매핑 유형을 보려면 "매핑 파일 참조"(247페이지)를 참조하십시오.

CI 유형 제거

1. TQL에서 CI 유형을 제거합니다.
2. 매핑 파일에서 해당 CI 유형에 대한 매핑 섹션을 제거합니다.

링크 추가

1. 데이터에 두 개의 끝 CI가 있는지 확인합니다.
2. 추가해야 할 링크가 실제로 유효한 링크인지 확인합니다(CI 유형 관리자에서 확인).
3. 매핑 xml의 관계 섹션에 링크 요소를 추가합니다.

링크 제거

1. 매핑 파일에서 제거할 링크의 링크 섹션을 제거합니다.
2. 가능한 경우 TQL에서 링크를 제거합니다(TQL의 효율성 또는 기능에 영향을 미치지 않는 경우).

8장: 일반 어댑터 개발

이 장의 내용:

인스턴스 동기화

밀어넣기 및 채우기 일반 어댑터 작업은 인스턴스 데이터를 사용합니다. *인스턴스* 및 *루트*의 개념에 대한 자세한 내용은 "인스턴스 기반 채우기 흐름"(186페이지) 및 "일반 어댑터를 사용하여 데이터 밀어넣기 수행"(262페이지)을 참조하십시오.

일반 어댑터를 사용하여 데이터 밀어넣기 수행

데이터 밀어넣기에서는 기존의 강화된 일반 밀어넣기 어댑터 프레임워크에서 XML 스키마를 약간 변경하여 사용합니다.

참고: 일반 어댑터는 인스턴스 모드에서 작동합니다(즉, 단일 CI 유형에서는 작동하지 않지만 주요 루트 CI로 함께 그룹화된 CI 컬렉션에서는 작동). 자세한 내용은 "인스턴스 기반 채우기 흐름"(186페이지)을 참조하십시오.

양방향 매핑 의미 체계를 수용하기 위해 필요한 XML 스키마 변경은 다음과 같습니다.

- **<targetcis>** 태그가 **<target_entities>**로 이름이 변경되었습니다.
- **<source_instance_type>** 태그가 **<source_instance>**로 이름이 변경되었습니다.

- **<target_ci_type>** 태그가 **<target_entity>**로 이름이 변경되었습니다.
- **<for-each-source-ci>** 태그가 **<for-each-source-entity>**로 이름이 변경되었습니다.
- 헤더 **versions** 특성이 **version**으로 이름이 변경되었으며 더 이상 소수점이 필요 없습니다.

이 섹션에서는 일반 어댑터 프레임워크를 사용하여 데이터 밀어넣기에 대한 정보를 제공합니다.

밀어넣기 개요

일반 어댑터는 TQL 쿼리 결과를 나타내는 데이터 구조에서 작동합니다. 일반 어댑터 프레임워크 위에 작성된 각 어댑터는 이 데이터 구조를 처리하고 필요한 대상으로 밀어넣습니다.

데이터 구조의 이름은 **ResultTreeNode (RTN)**입니다. RTN은 어댑터의 매핑 파일과 TQL 쿼리 결과에 따라 만들어집니다. 일반 어댑터 프레임워크에 사용되는 쿼리는 루트 기반이어야 합니다. 즉, 쿼리에 요소 이름이 **root**인 쿼리 노드 하나가 포함되어 있거나 접두사 **root**로 시작하는 관계 요소가 하나 이상 포함되어 있어야 합니다. 이 CI 또는 관계는 쿼리의 루트 요소로 사용됩니다. 자세한 내용은 *HP Universal CMDB 데이터 흐름 관리 안내서*에서 데이터 밀어넣기를 참조하십시오.

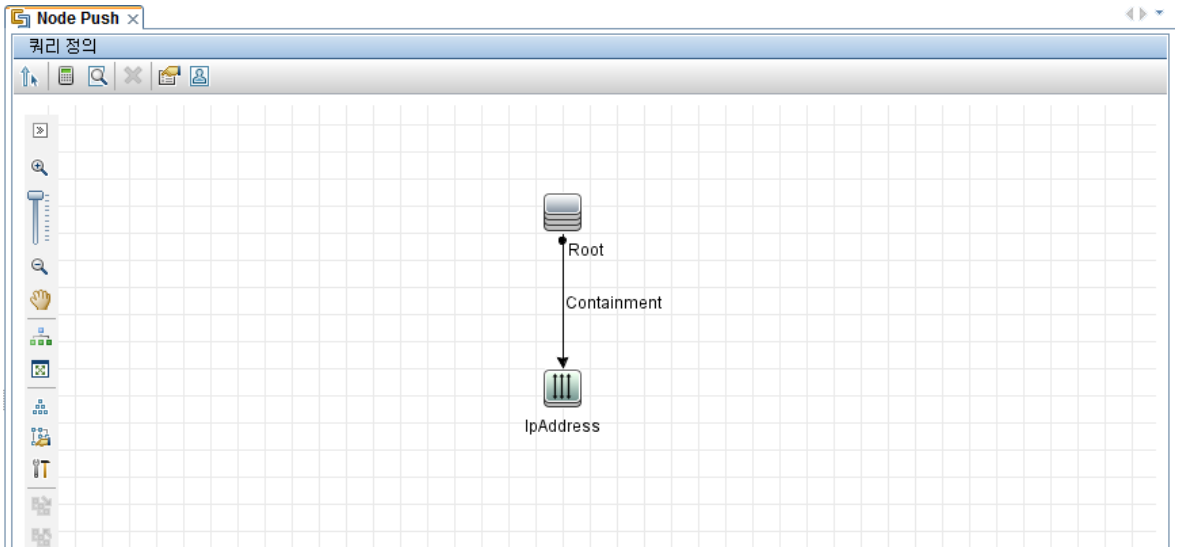
강화된 밀어넣기 어댑터를 개발하는 두 가지 기본 단계는 다음과 같습니다.

1. PushAdapterConnector 인터페이스 구현 –이 인터페이스는 추가, 업데이트, 삭제할 데이터를 RTN 목록으로 받아 대상에 밀어넣기를 수행합니다.
2. 매핑 파일 만들기 –매핑 파일이 TQL 결과의 CI 및 특성을 매핑하여 RTN 구조 만들기를 결정합니다.

매핑 파일

다음 예는 매핑 파일을 만드는 방법을 보여줍니다.

이 예제에서는 노드 및 IP 주소 밀어넣기를 시뮬레이션합니다. 여기서는 다음과 같이 **Node Push**라는 TQL 쿼리를 만듭니다.

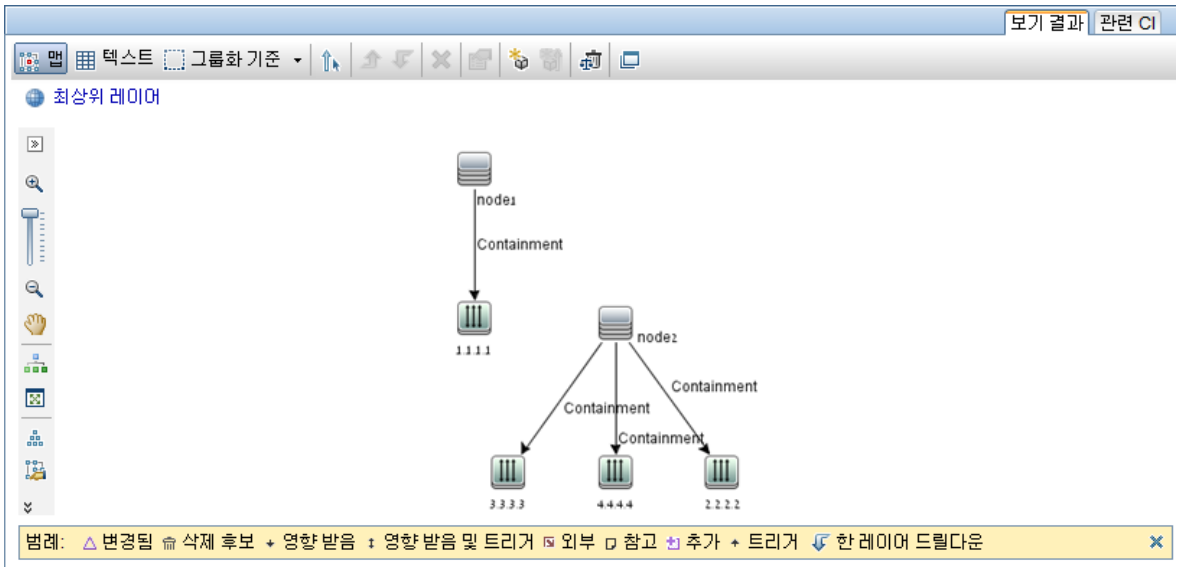


매핑 파일에는 두 개의 대상 CI 유형, 즉 **Computer**와 **IP**를 만듭니다. Computer에는 변수 하나와 특성 두 개가 있습니다. IP에는 특성 하나가 있습니다.

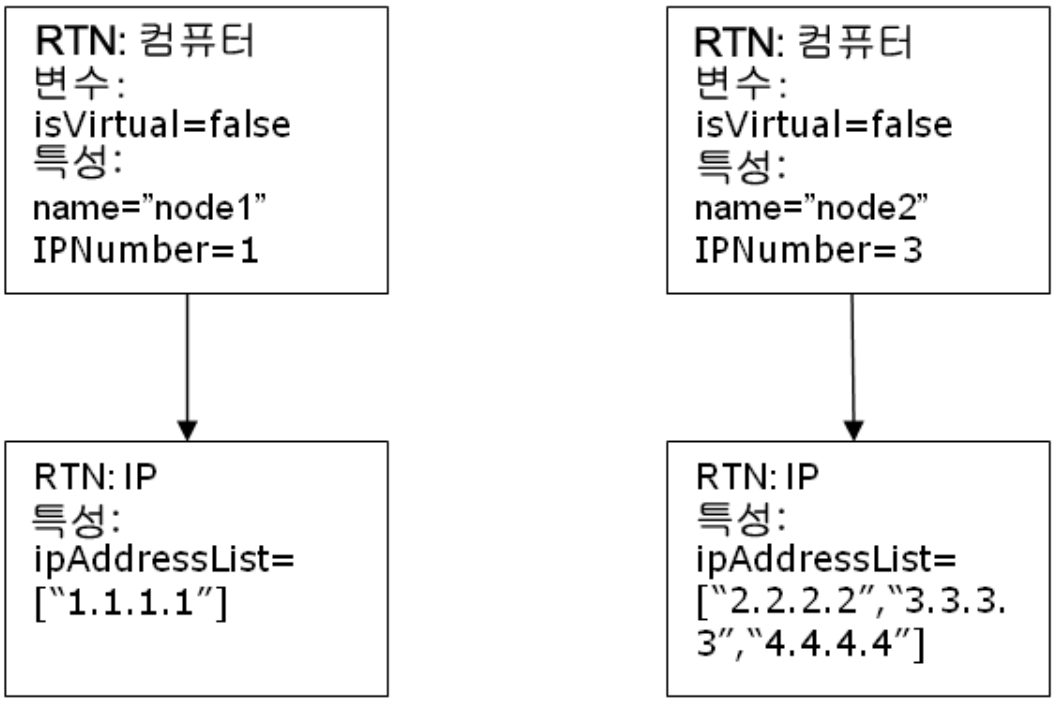
다음은 매핑 XML 파일입니다.

```
<?xml version="1.0" encoding="UTF-8"?>
<integration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../generic-adapter.xsd">
    <info>
        <source name="UCMDB" version="10.20" vendor="HP"/>
        <target name="PushProduct" version="9.3" vendor="HP"/>
    </info>

    <import>
        <scriptFile path="mappings.scripts.PushFunctions"/>
    </import>

    <target_entities>
        <source_instance query-name="Node Pusy" root-element-name="Root">
            <target_entity name="Computer" is-valid="(Root['root_iscandidatefordeletion'] == null) ? true : !Root['root_iscandidatefordeletion']">
                <variable name="isVirtual" datatype="BOOLEAN" value="PushFunctions.isVirtual(Root['root_class'])"/>
                <target_mapping name="name" datatype="STRING" value="Root['name']"/>
                <target_mapping name="ipNumber" datatype="INTEGER" value="Root.IpAddress.size()"/>
                <target_mapping name="description" datatype="STRING" value="PushFunctions.getDescription(isVirtual)"/>
                <target_entity name="IP">
                    <target_mapping name="IpAddressList" datatype="STRING_LIST" value="Root.IpAddress*.getAt('name')"/>
                </target_entity>
            </target_entity>
        </source_instance>
    </target_entities>

</integration>
```

쿼리 결과는 다음과 같이 표시됩니다.



다음은 이 매핑 파일에 따라 작성된 RTN 목록입니다.



각각의 루트 인스턴스는 매핑 파일을 사용하여 별도로 매핑됩니다. 따라서 이 예에서 PushAdapterConnector는 두 RTN 루트의 목록을 받습니다.

참고: 이전의 밀어넣기 어댑터는 CI 유형에 대한 일반 매핑을 만들 수 있었습니다. 새 밀어넣기 어댑터 매핑은 TQL 쿼리별로 실행됩니다. **x**라는 쿼리를 사용하는 밀어넣기 작업을 실행하는 동안, 어댑터는 관련 매핑 파일을 찾습니다(특성: query-name=x).

Groovy 스크립트 언어를 사용하여 매핑 파일의 값을 계산할 수 있습니다. 자세한 내용은 "Groovy Traveler"(266페이지)를 참조하십시오.

Groovy Traveler

다음과 같은 방식으로 TQL 쿼리 결과에 액세스합니다.

- **Root[attr]**는 Root 요소의 **attr** 특성을 반환합니다.
- **Root.Query_Element_Name**은 TQL Query_Element_Name에 이름이 지정되어 있으며 현재 루트 CI에 링크된 CI 인스턴스의 목록을 반환합니다.
- **Root.Query_Element_Name[2][attr]**은 현재 루트 CI에 링크된 세 번째 Query_Element_Name의 **attr** 특성을 반환합니다.
- **Root.Query_Element_Name*.getAt(attr)**는 TQL에서 이름이 Query_Element_Name이며 현재 루트 CI에 링크된 CI 인스턴스의 **attr** 특성 목록을 반환합니다.

Groovy Traveler에서 다음과 같은 추가 특성에도 액세스할 수 있습니다.

- **cmdb_id** –CI 또는 관계의 UCMDB ID를 문자열로 반환합니다.
- **external_cmdb_id** –CI 또는 관계의 외부 ID를 문자열로 반환합니다.
- **Element_type** –CI 또는 관계의 요소 유형을 문자열로 반환합니다.

import 태그:

```
<import>
<scriptFile path="mappings.scripts.PushFunctions"/>
</import>
```

이것은 매핑 파일에서 모든 groovy 스크립트에 대해 가져오기를 선언하는 것을 의미합니다. 이 예에서 **PushFunctions**는 몇 개의 정적 함수가 포함된 groovy 스크립트 파일이며 매핑 중에 액세스할 수 있습니다(value=" PushFunctions.foo()")

source_instance_type

TQL별로 매핑이 수행되며, query-name 값은 현재 매핑의 관련된 TQL입니다. '*'는 이 매핑 파일이 **Node Push** 접두사로 시작하는 모든 TQL 쿼리와 연결되어 있음을 의미합니다.

```
<source_instance_type query-name="Node Push*" root-element-name="Root">
```

source_instance_type 태그는 매핑되는 루트 요소를 지정합니다.

root-element-name은 TQL에 있는 루트 이름과 정확하게 일치해야 합니다.

target_entity

이 태그는 RTN을 만드는 데 사용됩니다.

name 특성은 target_entity 이름을 나타냅니다(name=Computer).

is-valid 특성은 매핑하는 동안 계산되는 Boolean 값이며 현재 target_ci가 올바른지 여부를 결정합니다. 잘못된 target_entities는 RTN에 추가되지 않습니다. 이 예제에서는 UCMDB에서 **root_ iscandidatefordeletion** 특성이 true인 target_entity 인스턴스를 만들지 않으려고 합니다.

target_entity에는 매핑 중에 계산되는 변수를 사용할 수 있습니다.

```
<variable name="vSerialNo" datatype="STRING" value="Root['serial_number']"/>
```

변수 **vSerialNo**는 현재 루트의 **serial_number** 값을 가져옵니다.

RTN의 특성은 **target_mapping** 태그를 사용하여 만듭니다. **value** 필드의 groovy 스크립트 실행 결과가 RTN 특성에 할당됩니다.

```
<target_mapping name="SerialNo" datatype="STRING" value="vSerialNo"/>
```

SerialNo는 **vSerialNo** 변수의 값을 할당합니다.

다음과 같이 target_entity를 다른 target_entity의 하위 항목으로 정의할 수 있습니다.

```
<target_entity name="Portfolio">
<variable name="vSerialNo" datatype="STRING" value="Root['global_id']"/>
<target_mapping name="CMDBId" datatype="STRING" value="globalId"/>
<target_entity name=Asset">
<target_mapping name="SerialNo" datatype="STRING" value="vSerialNo"/>
</target_entity>
</target_entity>
```

Portfolio라는 RTN에는 **Asset**이라는 하위 RTN이 생깁니다.

for-each-source-entity

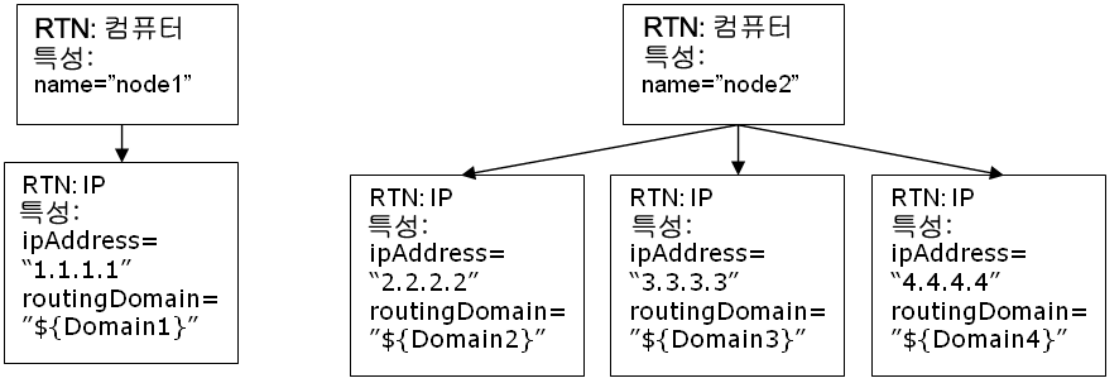
이 태그는 루트 인스턴스의 특정 CI를 나열합니다. 다음과 같은 필드가 있습니다.

- **source-entities=" "** –대상 CI가 만들어지는 CI의 목록입니다. 이 목록은 Groovy Traveler에서 **Root.IpAddress** 필드로 정의됩니다.
- **count-index=" "** –루프의 현재 반복에서 CI 인덱스가 들어가는 변수입니다.
- **var-name=" "** –루프의 현재 반복에 있는 CI의 이름입니다.

예로 든 매핑 파일을 수정해 보겠습니다.

```
<target_entity name="Computer">
    <target_mapping name="name" datatype="STRING" value="Root['name']"/>
    <for-each-source-entity count-index="i" var-name="currIP" source-entities="Root.IpAddress">
        <target_entity name="IP">
            <target_mapping name="ipAddress" datatype="STRING" value="Root.IpAddress[i]['name']"/>
            <target_mapping name="routingDomain" datatype="STRING" value="currIP['routing_domain']"/>
        </target_entity>
</target_entity>
```

이 매핑 파일에 따라 작성되는 RTN 목록은 다음과 같습니다.



dynamic_mapping

이 태그는 RTN 구조를 만드는 동안 대상 데이터 저장소에서 데이터 매핑을 만드는 기능을 추가합니다.

예: UCMDB에서 **Node.name**에 관련된 **id** 열과 **name** 열이 있는 **Computer**라는 테이블이 포함된 데이터베이스가 대상인 경우를 가정해 보겠습니다. 두 열은 모두 고유합니다. 데이터베이스에는 Computer 테이블의 **parentID**에 대한 참조 키가 있는 **IP**라는 테이블도 있습니다. 'dynamic_mapping'은 name과 id를 <name,id>로 저장하는 맵을 만들 수 있습니다. Adapter는 이 맵을 기반으로 ID를 컴퓨터에 대응시키고 IP 테이블의 **parentID** 특성에 올바른 값을 밀어넣을 수 있습니다. RTN을 만드는 동안 이 맵을 사용하여 parentID 특성에 값을 할당할 수 있습니다.

매핑은 **map_property**에 따라 결정됩니다. dynamic_mapping은 각 청크에 대해 한 번씩 실행됩니다.

```
<dynamic_mapping name="IdByName " keys-unique="true">
```

name 특성은 맵의 이름을 나타냅니다. **keys-unique** 특성은 키가 고유한지 여부를 나타냅니다(각 키는 값 또는 값 집합에 매핑).

이 예에 사용된 맵은 이름이 **IdByName**이고 고유한 키가 있습니다. 스크립트에서 맵에 액세스하려면 다음 명령을 실행합니다.

```
DynamicMapHolder.getMap('IdByName')
```

그러면 해당 맵에 대한 참조가 반환됩니다.

map_property 태그는 매핑을 기반으로 하는 속성을 만듭니다.

예:

```
<map_property property-name="SQLQuery" datatype="STRING"
property-value="SELECT name, id FROM Computer"/>
```

이 예에서 속성 이름은 **SQLQuery**이고 값은 맵을 만드는 SQL 문입니다. PushConnector 인터페이스에 대한 **retrieveUniqueMapping** 및 **retrieveNonUniqueMapping** 메서드 구현은 반환된 맵의 실제 구현을 결정합니다.

글로벌 변수

다음과 같은 글로벌 변수에서 매핑 파일의 Groovy 스크립트에 액세스할 수 있습니다.

- **Topology** -유형: Topology. 현재 청크의 토폴로지 인스턴스입니다.
- **QueryDefinition** - 유형: QueryDefinition. 현재 TQL의 쿼리 정의 인스턴스입니다.
- **OutputCI** -유형: ResultTreeNode. 현재 트리 매핑에 있는 루트 요소의 RTN입니다.
- **ClassModel** -유형: ClassModel. 클래스 모델의 인스턴스입니다.
- **CustomerInformation** -유형: CustomerInformation. 작업을 실행하는 고객에 대한 정보입니다.
- **Logger** -유형: DataAdapterLogger. 이 로거는 어댑터에서 UCMDB 로깅 프레임워크에 대한 로그를 쓰는 데 사용할 수 있습니다.

Groovy 스크립트 작성

이 섹션에서는 **PushFunctions.groovy** 파일을 만듭니다. 이 파일에는 루트 인스턴스를 매핑하는 동안 사용되는 정적 함수가 포함됩니다.

```
package mappings.scripts

public class PushFunctions {

    public static boolean isVirtual(def nodeRole){
        return isListContainsOne(def list, "MY_VM", "MY_SIMULATOR");
    }



public static String getDescription(boolean isVirtual){
        if(isVirtual){
            return "This is a VM";
        }
        else{
            return "This is physical machine";
        }
    }



private static boolean isListContainsOne(def list, ...stringList){
        //returns true if the list contains one of the values.
    }
}
```

PushAdapterConnector 인터페이스 구현

구현은 다음과 같은 기본 단계를 지원해야 합니다.

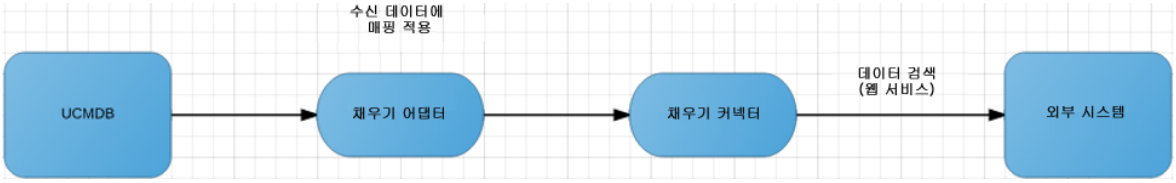
```
public class PushExampleAdapter implements PushAdapterConnector
{

public UpdateResult pushTreeNodes(PushConnectorInput input) throws DataAccessException{

// 1. build an UpdateResult instance - the UpdateResult is used to return mappings between the
sent ids to the actual ids that entered the data store.
// Also has an update status which allows to pass the status of data that was actually pushed,
detailed status reports on failed IDs, and actions actually performed on successful ids.
// 2. handle the data:
// a. handle data to add. Can be retrieved by: input.getResultTreeNodes.getDataToAdd();
//  b. handle data to update.
//  c. handle data to delete.
// 3. Return the Update result.
    }
public void start(PushDataAdapterEnvironment env) throws DataAccessException{
    // this method is called when the integration point created,
or when the adapter is reloaded
    //(i.e after changing one of the mapping files
    // and pressing 'save').

}



public void testConnection(PushDataAdapterEnvironment env) throws DataAccessException {
        // this method is called when pressing the 'test
connection' button in the
        //creation of the integration point.
        // For example if we push data to RDBMS this method
can create a connection
        //to the database and will run a dummy SQL statement.
        // If it fails it writes an error message to the log
and throws an exception.
    }




Map<Object, Object> retrieveUniqueMapping(MappingQuery mappingQuery){
//This method will create the map according to the given mappingQuery. It will be called in the
// mapping stage of the adapter execution, before the 'UpdateResult pushTreeNodes' method.
// This method is called when the 'keys-unique' attribute of the 'dynamic_mapping' tag is true.
}
Map<Object, Set<Object>> retrieveNonUniqueMapping(MappingQuery mappingQuery){
 // This method is called when the 'keys-unique' attribute of the 'dynamic_mapping' tag is false.
//  In this case a key can be mapped to several values.
```

```
}
}
```

일반 어댑터를 사용하여 데이터 채우기 수행

이 섹션에는 다음 항목이 포함됩니다.

채우기 프레임워크 아키텍처



메커니즘은 채우기 어댑터 프레임워크의 메커니즘과 동일합니다. 즉, 이 채우기 어댑터 프레임워크의 사용자가 매핑 파일 및 커넥터 구현을 제공하고 UCMDB 어댑터 패키지에서 이를 번들로 제공해야 합니다.

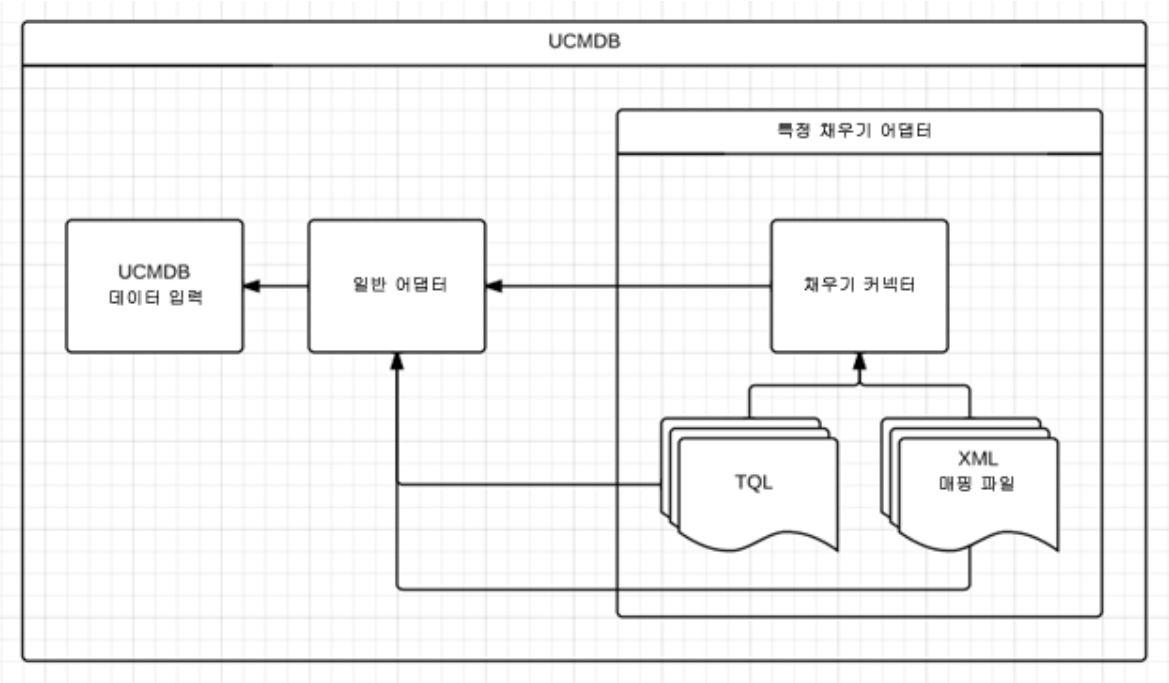
작업 흐름에는 다음 단계가 포함됩니다.

1. UCMDB 사용자가 UI에서 채우기 작업을 트리거합니다.
2. 명령을 채우기 어댑터로 보냅니다.
3. 채우기 어댑터에서 채우기 커넥터를 호출하고 데이터를 청크로 검색합니다.
4. 채우기 어댑터에서 각 청크의 데이터에 정의된 매핑을 적용한 후 UCMDB 서버로 전달합니다.

채우기에 관련된 기본 아티팩트

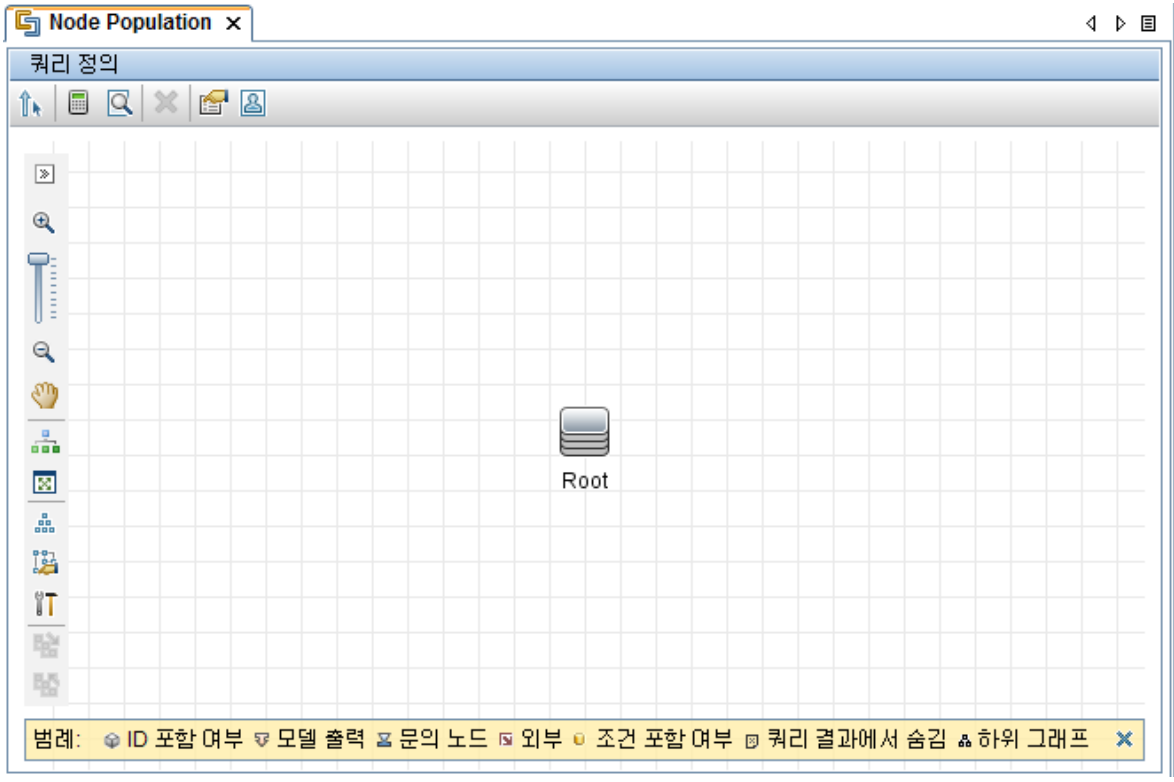
채우기에 관련된 기본 아티팩트는 다음과 같습니다.

- UCMDB에 채워질 데이터를 지정하는 TQL 쿼리
- 커넥터 반환 데이터를 UCMDB에 매핑하는 방법을 지정하는 XML 매핑 파일
- 필수 데이터
- 외부 시스템 데이터를 검색하고 이를 UCMDB 일반 어댑터에 반환하는 채우기 커넥터



채우기 TQL 쿼리

채우기 TQL 쿼리의 역할은 UCMDB에 채워질 데이터를 표시하는 것입니다. 예를 들어, 다음 그림의 TQL은 UCMDB의 Node 인스턴스를 가져오는 데 사용됩니다.



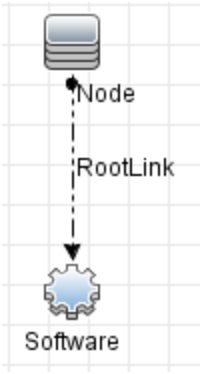
채우기 커넥터는 채우기 TQL 쿼리를 파악하고 외부 시스템에서 필요한 데이터를 제공합니다.

채우기 매핑 파일

XML 매핑 파일의 용도는 방향이 반대인 것을 제외하면 밀어넣기 작업의 용도와 같습니다. 이러한 매핑 파일은 커넥터에서 반환되는 데이터를 UCMDB 데이터에 매핑하는 방법을 설명합니다.

여기 제공된 정보는 채우기 매핑과 관련이 있으며, 강화된 일반 밀어넣기 어댑터에 대해서는 아직 언급되지 않습니다.

다음은 UCMDB 노드 및 실행 중인 소프트웨어에 대한 매핑 예입니다. 첫 번째 이미지는 노드 및 소프트웨어 채우기 TQL 쿼리를 보여 줍니다. 두 번째 이미지는 노드 및 소프트웨어 채우기 매핑을 보여 줍니다.




```
<target_entities>
    <!--The query name must match the one selected in the UI-->
    <source_instance query-name="Nodes And Software Population" root-element-name="PC">
        <!-- need to match case in UCMDB TQL -->
        <target_entity name="RootLink">
            <target_mapping name="name" datatype="STRING" value="PC['name'] + 'has' + PC.Programs[0]['name']"/>
        </target_entity>
        <target_entity name="Node" type="Util.getNodeType(PC)">
            <target_mapping name="name" datatype="STRING" value="PC['name']"/>
            <target_mapping name="description" datatype="STRING" value="PC['description']"/>
        </target_entity>
        <target_entity name="Software">
            <target_mapping name="name" datatype="STRING" value="PC.Programs[0]['name']"/>
            <target_mapping name="root_container_name" datatype="STRING" value="PC['name']"/>
            <target_mapping name="product_name" datatype="STRING" value="'vmware_hypervisor'"/>
        </target_entity>
    </source_instance>
</target_entities>
```

이러한 채우기 작업은 외부 시스템의 데이터를 ResultTreeNode(RTN) PC 양식으로 가져옵니다. ResultTreeNode API는 강화된 일반 밀어넣기 어댑터에서 도입되었으며, UCMDB 서버 **lib** 폴더에 있는 **push-interfaces.jar** 파일에 있습니다. 자세한 내용은 "일반 어댑터를 사용하여 데이터 밀어넣기 수행" (262페이지)을 참조하십시오.

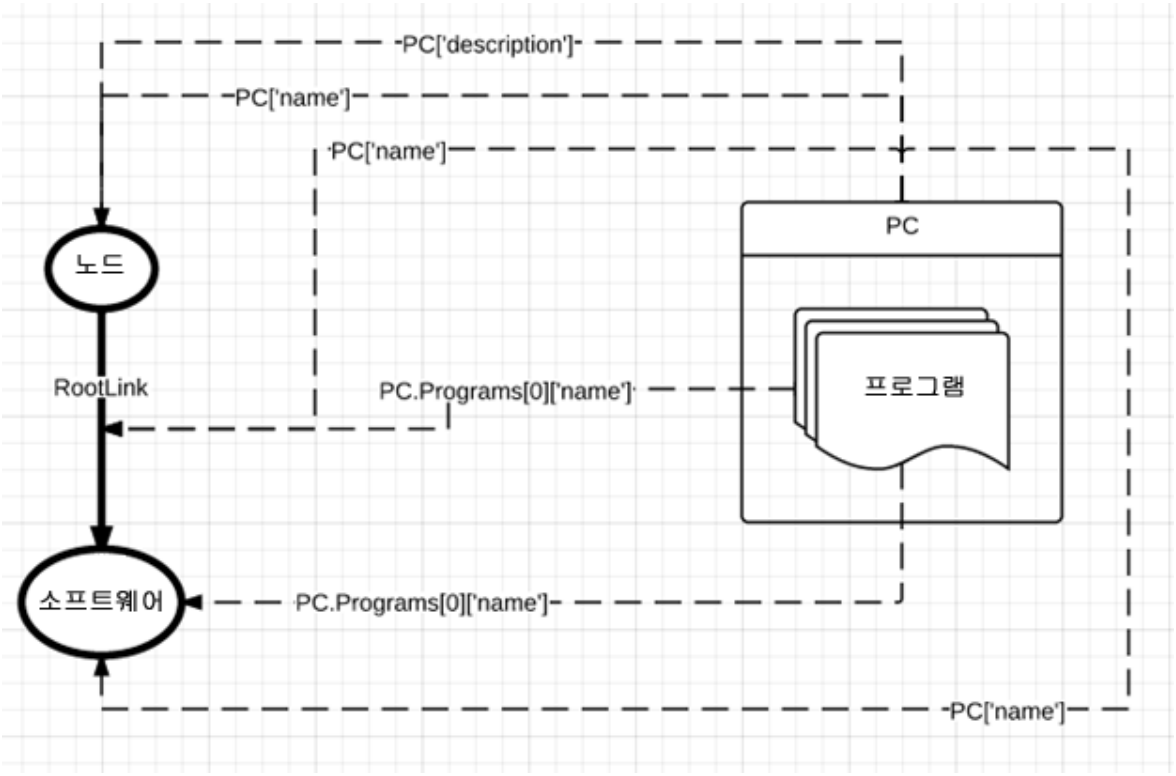
PC RTN에는 소프트웨어 유형 엔터티와 관련 특성이 있는 포함된 내장 프로그램 엔터티뿐만 아니라 특성 형식의 일반 노드 정보가 포함되어 있습니다.

하나의 PC 인스턴스는 UCMDB에 있는 다음 세 가지 엔터티에 매핑됩니다.

- 노드 CI
- 실행 중인 소프트웨어 CI
- Composition 링크 CI

PC 인스턴스의 형식에 대한 자세한 내용은 "채우기 요청 출력"(281페이지)을 참조하십시오.

다음 그림은 커넥터 데이터가 UCMDB 데이터에 매핑되는 방법을 보여 줍니다.



핵심 줄을 분석해 보겠습니다.

```
<!--The query name must match the one selected in the UI-->
<source_instance query-name="Nodes And Software Population" root-element-name="PC">
```

source_instance 정의는 엔터티를 UCMDB로 가져오고, 이러한 엔터티를 그룹화하는 UCMDB 토폴로지가 노드 및 소프트웨어 채우기 TQL 쿼리로 정의됨을 명시합니다. 또한 커넥터에서 반환되어 UCMDB 데이터를 만드는 데 사용될 데이터 구조는 **PC**라는 **ResultTreeNode**입니다.

```
<target_entity name="RootLink">
    <target_mapping name="name" datatype="STRING" value="PC['name'] + 'has' + PC.Programs[0]['name']"/>
</target_entity>
```

target_entity 태그는 새 UCMDB 엔터티가 여기서 시작하며, 이 엔터티가 노드 및 소프트웨어 채우기 TQL 쿼리의 **RootLink** 요소에 해당함을 명시합니다. 이 표시에는 새 엔터티의 UCMDB CI 유형도 포함됩니다.

생성될 **RootLink** 엔터티에는 하나의 특성, **name**이 있으며 해당 값은 **Computer_22 has MySQL Server**와 같습니다.

이 샘플 매핑에서는 수동 링크 채우기를 사용하지만 "자동 링크 채우기"(275페이지)에 설명된 자동 방식을 사용하는 것이 좋습니다.

채우기 유형 특성

```
<target_entity name="Node" type="Util.getNodeType(PC)">
```

노드 엔터티에는 **type** 특성이 있습니다. 이 **type**은 이 엔터티가 UCMDB에서 갖게 될 정확한 CI 유형을 나타냅니다. **type** 특성은 엔터티의 기본 생성 유형을 엔터티가 참조하는 TQL 요소(이 경우에는 Node)에서 가져오므로 필수가 아닙니다. 그러나 UCMDB 노드 CI 유형의 여러 인스턴스를 반환해야 하며 일부 인스턴스는 Windows이고 일부는 Unix인 경우 **type** 특성을 사용하여 정확한 UCMDB 생성 유형을 지정할 수 있습니다. 따라서 이 경우에는 PC 트리를 입력으로 수신하여 유효한 UCMDB CI 유형 식별자(Unix의 경우 "unix", Windows의 경우 "nt")를 반환하는 **getNodeType** 함수를 **Util** 함수 스크립트 파일 내부에 만듭니다.

> **참고: target_entity type** 특성은 채우기 흐름 컨텍스트에서만 사용할 수 있으며, 해당 값은 유효한 Groovy 식이어야 합니다.

소프트웨어 엔터티 만들기도 동일한 방식으로 설명할 수 있습니다.

자동 링크 채우기

"채우기 매핑 파일"의 매핑 샘플에서 채워진 링크를 명시적으로 매핑하는 데 필요한 사항을 확인했습니다. 아래 표시된 예와 같이 각 TQL 링크 요소에 대해 매핑된 대상 엔터티가 있어야 합니다.

```
<target_entity name="RootLink">
    <target_mapping name="name" datatype="STRING" value="PC['name'] + 'has' + PC.Programs[0]['name']"/>
</target_entity>
```

일반 어댑터 자동 링크 채우기 메커니즘을 사용하면 위 예에 표시된 것과 같이 더 이상 매핑 섹션을 사용하여 TQL 링크 요소를 매핑하지 않아도 됩니다. 프레임워크에서 속성이 비어 있는 링크 CI 인스턴스를 TQL에 지정된 유형으로 생성합니다. 채우기 TQL 쿼리에 있는 모든 링크에 대해 이 작업이 수행됩니다.

샘플 매핑의 결과로 링크 끝(end1 및 end2)으로 노드 및 실행 중인 소프트웨어 CI 인스턴스가 있는 유형 Composition의 링크 CI가 생성됩니다.

채우기 중인 링크에 다음이 필요한 경우 수동 링크 채우기를 사용해야 합니다.

- 동적 링크 유형(**type** 특성 사용)
- 링크 속성

수동 링크 채우기

일반 어댑터는 링크에 필요한 다음 세 가지 엔터티를 정의(매핑)하여 링크 채우기를 수행합니다.

- 링크 엔터티
- 링크의 End 1 엔터티
- 링크의 End 2 엔터티

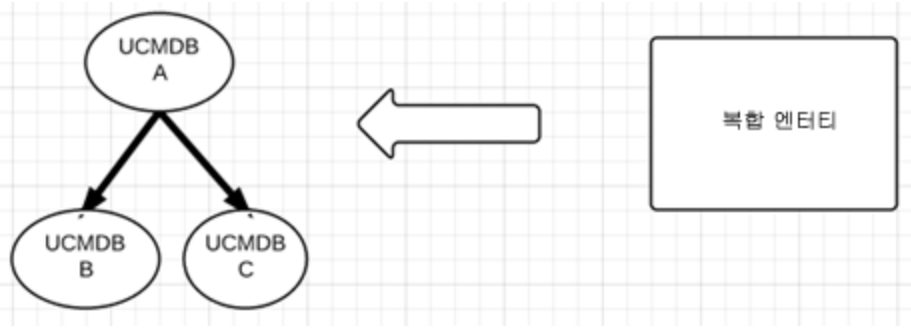
"채우기 매핑 파일"(273페이지)에 표시된 매핑의 예를 분석해 보겠습니다. 이 경우, UCMDB에서 세 가지 엔터티 유형인 노드, 실행 중인 소프트웨어 및 이들 간의 Composition 링크를 채웁니다. 링크(유형 Composition의 **RootLink** 링크)를 채워야 하므로 두 개의 링크 끝도 매핑해야 합니다. 따라서 TQL 쿼리를 보고 매핑이 필요한 엔터티가 노드(end 1) 및 소프트웨어(end 2)임을 확인합니다. 일반 어댑터 프레임워크는 TQL 쿼리에서 생성된 엔터티 요소 이름과 정의를 확인하여 링크 구조를 파악합니다. 채우기 작업에서는 노드 및 실행 중인 소프트웨어의 인스턴스도 가져와야 하므로 필요한 끝 매핑이 이미 적용됩니다.

링크 채우기 유형

링크 채우기 상황에는 다음 두 가지 유형이 있습니다.

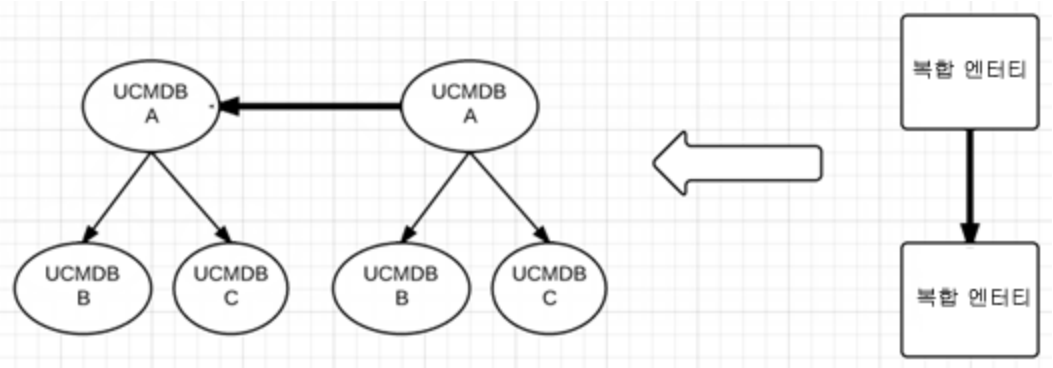
- 여러 개의 관련 UCMDB 엔터티에 있는 복잡한 외부 엔터티 분해

  이 경우 복잡한 외부 엔터티(예: PC)는 **Composition** 링크로 연결해야 하는 UCMDB **노드** 및 **실행 중인 소프트웨어** 유형으로 변환됩니다. 이러한 링크 유형은 UCMDB 컨텍스트에만 존재합니다.



- 복잡한 외부 엔터티 사이의 링크

  이 경우 두 개의 복잡한 외부 엔터티(예: PC) 사이의 링크를 모델링해야 합니다.



채우기 커넥터

채우기 커넥터는 외부 시스템 데이터를 검색합니다. 이 데이터는 설정된 API 형식(**ResultTreeNode**)으로 일반 어댑터에 전달된 다음 UCMDB 데이터 구조에 매핑되어 데이터 입력 프로세스를 통해 UCMDB에 삽입됩니다.

밀어넣기 커넥터와 유사하게 채우기 커넥터는 Java 및 Groovy 모두에서 구현할 수 있으며, 아래 그램에 표시된 채우기 커넥터 Java 인터페이스를 구현해야 합니다.

채우기 커넥터를 구성하려면 어댑터 구성 XML 파일에 다음 줄을 추가합니다.

```
<adapter-settings>
    <adapter-setting name="PopulationConnector.class.name">com.hp.ucmdb.connector.dummy.DummyPopulationConnector</adapter-setting>
```

```
public interface PopulationAdapterConnector extends GenericConnector, DynamicMappingConnector {

    /**
     * Executes a population request and provides the entities that will be populated in the format of
     * {@link com.hp.ucmdb.adapters.push.output.ResultTreeNode}
     * Used for both population and federation with the flow type flag which is present in the
     * {@link com.hp.ucmdb.adapters.population.connector.PopulationConnectorInput}
     *
     * @param input population request
     * @return a population response
     * @throws DataAccessException
     */
    PopulationConnectorOutput populate(PopulationConnectorInput input) throws DataAccessException;

    /**
     * Returns the collection of queries the current connector supports for population
     * Note: this method is independent to the adapter life cycle (i.e. start/shutdown methods) and must work at all times.
     * <p/>
     * The method also supports return of queries with their folders path: E.g.
     * "Folder/Secondary Folder/Query Name 1"
     * "Folder/Secondary Folder/Query Name 2"
     *
     * @param env the adapter environment
     * @return a collection of the supported queries
     */
    Collection<String> getPopulationQueries(DataAdapterEnvironment env);

    /**
     * Returns the collection of queries the current connector supports for federation
     * Note: this method is independent to the adapter life cycle (i.e. start/shutdown methods) and must work at all times.
     * <p/>
     * The method also supports return of queries with their folders path: E.g.
     * "Folder/Secondary Folder/Query Name 1"
     * "Folder/Secondary Folder/Query Name 2"
     *
     * @param env the adapter environment
     * @return a collection of the supported queries
     */
    Collection<String> getFederationQueries(DataAdapterEnvironment env);

    /**
     * Update target id (global id) for each source object.
     *
     * @param idMapping mapping between source id (external id) and target id (string)
     */
    void updateGlobalIDsFromTarget(FCmdbExtenalToTargetIdMappingSet idMapping);

    /**
     * This methods reports population queries resources used by the adapter.<br>
     * This allows editing these resources directly from the Integration Studio.
     *
     * @param input  the requested information
     * @param output the returned resources
     * @see com.hp.ucmdb.federationspi.adapter.resource.PushQueriesResourceLocator
     */
    void locatePopulationQueriesResources(DataAdapterEnvironment env, LocatePopulationQueriesResourcesInput input,
                                          LocatePopulationQueriesResourcesOutput output);

    /**
     * Returns the collection of classes the current adapter supports for query.<br>
     * Notes:<br>
     * 1. This method is independent of the adapter life cycle (i.e. start/shutdown methods).<br>
     * 2. If adapter configuration (xml) defines supported classes, this method doesn't need to be implemented (return null).<br>
     *
     * @param env the Adapter env
     * @return collection of the supported classes
     * @throws DataAccessException in case of an error
     */
    Collection<SupportedClassConfig> getSupportedClasses(DataAdapterEnvironment env);
}
```

첫 번째 메서드 populate는 외부 시스템에서 데이터를 검색하는 주요 커넥터 메서드입니다. 이 메서드는 채우기 TQL 쿼리를 입력으로 받아들여 일반 *ResultTreeNode* 형식으로 결과를 반환합니다. 자세한 내용

은 "일반 어댑터를 사용하여 데이터 밀어넣기 수행"(262페이지)을 참조하십시오. 커넥터는 주요 비즈니스 데이터와 함께 상태 및 청크 정보를 반환합니다.

두 번째 메서드 getSupportedQueries는 채우기 커넥터에서 지원하는 TQL을 나타냅니다.

세 번째와 네 번째 메서드는 더 개선된 사용 사례를 참조하는데, 채워진 데이터의 ID를 다시 밀어넣고 특정 쿼리에 대해 어댑터 내부에서 관련 채우기 리소스를 찾습니다. 이러한 API에 대한 자세한 내용은 **push-interfaces.jar** 파일을 참조하십시오.

채우기 요청 입력

채우기 요청은 UCMDB 채우기 쿼리를 설명하는 **QueryDefinition** 개체로 정의됩니다. 채우기 커넥터는 이 쿼리 개체를 읽고 외부 시스템의 쿼리 언어로 변환합니다.

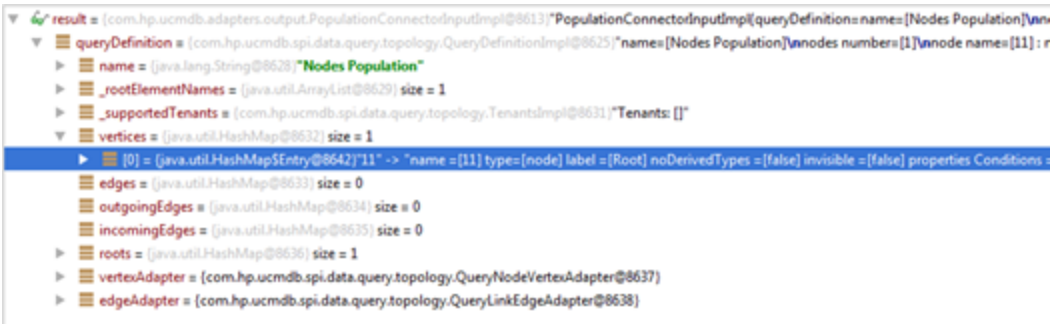
```
 * @author Sergiu Indrie
 * @since 10.20
 */
public interface PopulationConnectorInput {

    QueryDefinition getQueryDefinition();

    /**
     * Indicates the required {@link com.hp.ucmdb.adapters.push.output.ResultTreeNode} structure
     * that the population result must return.
     *
     * For the target mapping <target_mapping name="lMaxMemory" datatype="LONG" value="Root.VMWare_Host_Resource['vm_memory_limit']"/>
     * the resulting tree node should have the following structure: VMWare_Host_Resource will be a child of Root and vm_memory_limit will
     * be an attribute of VMWare_Host_Resource
     *
     * @return a map containing simplified result trees
     * @see PopulationConnectorOutput#getResultTreeNodes()
     */
    Map<String, ResultTreeNodeStructure> getResultTreeNodeStructure();

    /**
     * Returns the flow type of the operation.
     * Can be POPULATION or FEDERATION
     *
     * @return the flow type
     */
    FederationTopologyAdapterInput.FlowType getFlowType();

    /**
     * Returns the date from the last sync
     *
     * @return the date
     */
    Date getFromDate();
```

QueryDefinition 개체 이외에도 다음이 있습니다.

- **getResultTreeNodeStructure** –채우기 결과에서 반환해야 하는 필수 구조를 나타냅니다.
- **getFlowType** –커넥터에 대한 요청이 POPULATION 유형인지 또는 FEDERATION 유형인지 결정하는 데 사용됩니다.

getFromDate –마지막 동기화 날짜를 나타냅니다. 날짜가 null이면 FULL POPULATION이 실행되고 그렇지 않으면 Diff POPULATION이 실행됩니다(흐름 유형이 FEDERATION인 경우 getFromDate 메서드에서 항상 null을 반환함).

다음 그림은 채우기 요청 샘플입니다.



이 예제에서는 요청에 **Nodes Population** 쿼리가 포함되어 있습니다. 쿼리에 **Node** 유형의 TQL 요소 하나만 포함되어 있음을 확인할 수 있습니다.

ResultTreeNodeStructure

PopulationAdapterConnector를 구현하려면 UCMDB 채우기 TQL을 읽고, UCMDB에서 요청하는 사항을 파악하고, 외부 시스템 엔터티를 사용하여 결과를 제공해야 합니다. 예를 들어, UCMDB에서 외부 시스템에 있는 비즈니스 서비스 인스턴스에 관련된 모든 노드를 요청할 수 있으며, 컴퓨터에 상응하는 외부 시스템이 **Service** 엔터티와 관련된 **PC**일 수 있습니다. 따라서 채우기 커넥터에서 **Service** 인스턴스에 연결된 **PC** 인스턴스를 반환해야 합니다. 이 경우 매핑은 다음과 같습니다.

```
<target_entities>
    <source_instance query-name="Nodes And BS Population" root-element-name="PC">
        <!-- Node -->
        <target_entity name="Root">
            <target_mapping name="name" datatype="STRING" value="PC['name']"/>
        </target_entity>

        <!-- Business Service -->
        <target_entity name="BusinessService">
            <target_mapping name="name" datatype="STRING" value="PC.Service['name']"/>
            <target_mapping name="description" datatype="STRING" value="PC.Service['description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

이 경우 **Service** 인스턴스와 관련된 **PC** 인스턴스를 반환하기 위해 하위 노드로 **Service**가 포함된 **PC** RTN을 반환합니다. 그러나 다음과 같이 하위에 **PC**가 있는 **Service** RTN 형식으로 매핑을 만들도록 선택할 수도 있었습니다(매핑이 잘못됨).

```
<target_entities>
    <source_instance query-name="Nodes And BS Population" root-element-name="Service">
        <!-- Node -->
        <target_entity name="Root">
            <target_mapping name="name" datatype="STRING" value="Service.PC['name']"/>
        </target_entity>

        <!-- Business Service -->
        <target_entity name="BusinessService">
            <target_mapping name="name" datatype="STRING" value="Service['name']"/>
            <target_mapping name="description" datatype="STRING" value="Service['description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

따라서 채우기 커넥터 개발을 지원하려면 일반 어댑터에서 보낸 채우기 요청에 매핑 파일에서 사용된 데이터의 RTN 구조도 포함되어야 합니다. 이는 반환되는 RTN의 필수 형식을 구현 중인 커넥터에 표시합니다.

첫 번째 사례에서 **ResultTreeNodeStructure**는 다음과 같습니다.

PC
- name
  서비스
  - name
  - description

두 번째 사례에서 **ResultTreeNodeStructure**는 다음과 같습니다.

서비스
- name
- description
  PC
  - name

채우기 요청 출력

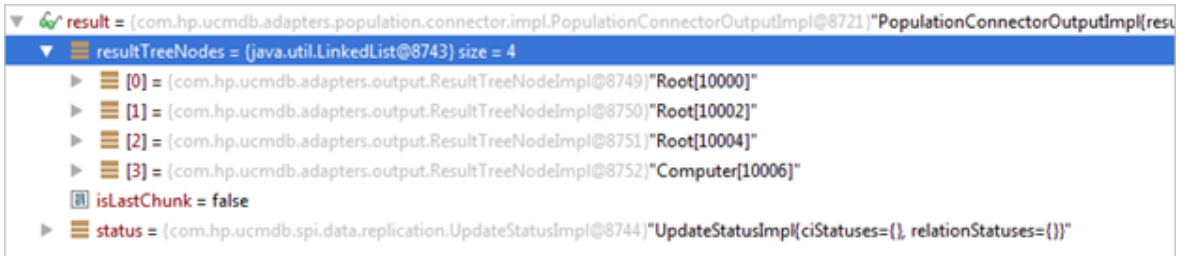
채우기 요청을 처리할 때 채우기 커넥터는 PopulationConnectorOutput**을 반환해야 합니다.**

```
public interface PopulationConnectorOutput {

    /**
     * The result trees representing the external entities in (@link com.hp.ucmdb.adapters.push.output.ResultTreeNode) format.
     *
     * @return a list of result trees or an empty list
     */
    List<ResultTreeNode> getResultTreeNodes();

    /**
     * Adds a result tree to the population output object.
     *
     * @param resultTreeNode the result tree to add
     */
    void addResultTreeNode(ResultTreeNode resultTreeNode);

    /**
     * This method indicates if the result received is the last chunk of data.
     *
     * @return true if this is the last chunk, false otherwise.
     */
    boolean isLastChunk();

    /**
     * Set whether this result object is the last chunk or not.
     */
    void setLastChunk(boolean isLastChunk);

    /**
     * Returns the status information about the population result.
     */
    UpdateStatus getStatus();

    /**
     * Set the population result status.
     */
    void setStatus(UpdateStatus status);

}
```

이 출력 개체에는 다음이 포함됩니다.

- ResultTreeNode **형식의 쿼리 데이터**
- **상태 정보(실패할 경우 필요)**
- **청크 정보**

다음 그림은 채우기 응답 샘플입니다.



위 응답에서 커넥터는 네 개의 데이터 인스턴스(UCMDB Node**에 해당), 비어 있는 상태(성공을 표시) 및 마지막 청크가 아님을 나타내는 플래그를 반환합니다.**

채우기 어댑터 모드

UCMDB 어댑터 프레임워크에서는 다음과 같이 두 가지 유형의 채우기 어댑터를 사용할 수 있습니다.

- 표준 채우기 어댑터
  - 어댑터 XML 파일에 **<changes-source/>** 태그가 없는 것이 특징입니다.
  - 항상 전체 쿼리 데이터를 외부 시스템에서 가져옵니다. 이 경우 UCMDB Probe Framework에서 연속된 두 실행의 차이를 확인합니다. Probe Framework는 지정된 쿼리의 이전 결과와 현재 결과를 비교하고 그 차이를 계산하여 해당 작업을 수행합니다. 전체 채우기는 현재의 쿼리 결과를 비교하지 않고 최종 결과로 처리하여 수행됩니다. 이 흐름은 채워진 데이터를 "시작 날짜"로 필터링하지 않는 것을 의미하는데, 이는 날짜로 필터링할 경우 데이터 비교가 무의미해지기 때문입니다.
- changes-source 채우기 어댑터
  - 다음과 같이 **<changes-source/>** 태그를 사용하여 어댑터 XML 파일에 구성됩니다.

```
<adapterInfo>
    <adapter-capabilities>
        <support-federated-query/>
        <support-replicatioin-data>
            <source>
                <changes-source/>
                <push-back-ids/>
                <re-populate/>
            </source>
            <target/>
        </support-replicatioin-data>
```

  - changes-source 어댑터는 연속된 두 실행의 차이를 계산합니다.

changes-source 어댑터를 사용할 때의 CI 삭제

changes-source 채우기 어댑터를 사용 중인 경우 해당 어댑터는 CI를 명시적으로 삭제합니다. 이는 올바른 Groovy 식을 수용하는 **is-deleted** 매핑 파일 XML 특성을 사용하여 수행됩니다.

예를 들어, 아래 표시된 매핑 파일에서 채우기 커넥터는 **Service** 인스턴스를 반환합니다. 인스턴스는 계속 유효하지만 해당 **Service** 인스턴스에 포함된 일부 CI는 삭제되었습니다. 해당 CI가 삭제되었음을 표시하려면 **BusinessService** 매핑에 **is-deleted** 특성을 사용해야 합니다.

```
<target_entities>
    <source_instance query-name="Nodes And BS Population" root-element-name="Service">
        <!-- Node -->
        <target_entity name="Root">
            <target_mapping name="name" datatype="STRING" value="Service.PC['name']"/>
        </target_entity>

        <!-- Business Service -->
        <target_entity name="BusinessService" is-deleted="Functions.isOlderThanThreeMonths()">
            <target_mapping name="name" datatype="STRING" value="Service['name']"/>
            <target_mapping name="description" datatype="STRING" value="Service['description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

명시적 외부 ID 매핑

채워진 데이터(CI)에 커넥터/어댑터 제어 **ExternalId**가 있어야 하는 상황이 있을 수 있습니다. 다음과 같은 매핑 구조를 사용하여 이를 수행합니다.

```
<target_entities>
    <!--The query name must match the one selected in the UI-->
    <source_instance query-name="Node with ID" root-element-name="Computer">
        <!-- need to match case in UCMDB TQL -->
        <target_entity name="Root">
            <!--This is how the RTN External ID is set-->
            <variable name="external_id_obj" datatype="STRING" value="Computer['external_id_obj']"/>
            <!--RTN Attributes-->
            <target_mapping name="name" datatype="STRING" value="Computer.Asset[0]['name']"/>
            <target_mapping name="description" datatype="STRING" value="Computer['name']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

이 경우 Root CI는 커넥터 수준에서 생성되어 Computer['external_id_obj']에 배치된 ExternalId로 채워집니다. ExternalId는 Groovy 스크립트를 사용하여 매핑 수준에서도 만들 수 있습니다.

참고: 외부 ID를 명시적으로 만드는 메커니즘은 target_entity **type** 특성을 다시 정의합니다. 따라서 외부 ID를 매핑 스크립트 파일을 사용하여 만들거나 커넥터 내부에서 만들면 **type** 특성이 무시되고, 채워진 CI의 마지막 UCMDB 유형은 **ExternalId** 개체에 설정된 UCMDB 유형이 됩니다.

글로벌 ID 다시 밀어넣기

외부 시스템에서도 UCMDB의 채워진 CI를 동기화 상태로 유지해야 하는 상황이 있습니다. 이 시나리오의 경우 일반 어댑터 프레임워크에서 다시 밀어넣기 ID를 사용하도록 할 수 있습니다. 이 기능을 사용하기 위해 UCMDB에 채워진 모든 CI에 콜백이 수행되어 각 CI에 대해 할당된 글로벌 ID를 채우기 어댑터에 알립니다.

이 기능을 사용하도록 설정하려면 아래 예에 표시된 줄을 어댑터 구성 XML 파일에 추가합니다.

```
<adapterInfo>
    <adapter-capabilities>

        <support-federated-query>
        <supported-classes>
            <supported-class is-derived="true" all-attributes-supported="true" name="node" is-reconciliation-supported="true"/>
            <supported-class is-derived="true" all-attributes-supported="true" name="business_service" is-reconciliation-supported="true"/>
            <supported-class is-derived="true" all-attributes-supported="true" name="incident" is-reconciliation-supported="true"/>
        </supported-classes>
        </support-federated-query>
        <support-replicatioin-data>
            <source>
                <push-back-ids/>
                <instance-based-data/>
                <population-queries-resources-locator/>
            </source>
            <target>
                <instance-based-data>true</instance-based-data>
                <push-queries-resources-locator/>
            </target>
        </support-replicatioin-data>
        <general-resources-locator/>
    </adapter-capabilities>
```

또한 다음과 같이 PopulationAdapterConnector **메서드를 구현해야 합니다.**

```
/**
* <b>Description:</b><br>
 * This Interface is used for implementing
 * an adapter to allow the definition of Integration Points.<br>
 * An adapter that implements this Interface will expose
 * ths ability to run Population flows.<br>
 * The only Population flow exposed by this adapter is the Simple Flow using {@link #getDataResult(com.hp.ucmdb.federationspi.adapter.federation.
 * which will return the entire data set of data each time, and compare it to the last data set (handled by framework).<p>
 * It is highly recommended to allow better link validation by implementing {@link com.hp.ucmdb.federationspi.adapter.ReportsLinks}<br>
 * If possible, it's recommended to implement the {@link PopulationChangesAdapter} instead,
 * allowing the adapter to have better performance and control.<p>
 *
 * @see com.hp.ucmdb.federationspi.data.query.topology.TopologyFactory
 * @see com.hp.ucmdb.federationspi.adapter.ChunkTopologyResultGetter
 * @see PopulationAdapter
 * @since 10.10
 *
*/
public interface PopulationAdapter extends BasicSourceDataAdapter{

    /**
     * Retrieves the calculated result of the given {@code QueryDefinition}.<br>
     * <li>
     * This method is called by the population framework.<br>
     * </li>
     * <p>
     * This method implementation may use the {@code UpdateStatus} to report warnings for specific CIs or Relations.
     * <p/>
     * When implementing this method, it should return the result by being aware to all the conditions
     * that appear on the {@code QueryDefinition} like: topology, cardinality, id conditions and property conditions.
     * </p><br>
     * When implementing this method, one must be aware that different flows may actually be used:<br>
     * 1) A topology only flow (a query definition with conditions or id conditions but with out any layout requested).<br>
     * 2) A layout only flow (a query definition with only ids condition and layout).<br>
     * 3) A full topology flow (a query with property conditions, id conditions and layout)<br>
     *
     * @param input contains the logged in user and queryDefinition that contains the topology, conditions and layout requested by the framework
     * @return {@link com.hp.ucmdb.federationspi.adapter.federation.FederationTopologyAdapterOutput} containing the query's calculated result.
     * @throws com.hp.ucmdb.federationspi.exception.DataAccessException
     */
    public FederationTopologyAdapterOutput getDataResult(FederationTopologyAdapterInput input) throws DataAccessException;

}
```

일반 어댑터를 사용하여 데이터 연합 수행

데이터 연합은 다음을 사용하여 수행됩니다.

연합 매핑 방법

연합 매핑은 UCMDB 연합 프레임워크에서 연합 요청 처리에 사용하는 하위 TQL을 매핑하여 수행됩니다. 일반적인 개념은 일반 어댑터에서 연합 요청을 수신할 때 다음이 발생한다는 것입니다.

1. 동적 연합 TQL 쿼리를 분석하고 연합 커넥터에서 제공하는 정적 연합 TQL 쿼리 목록과 비교합니다.
2. 정적 TQL 쿼리 일치가 수행됩니다. 이 TQL 쿼리는 지정된 연합 요청에 필요한 매핑을 식별하고, 연합 커넥터에 공급될 RTN 구조(커넥터에서 필요한 트리 노드 구조를 나타내는 Java 개체) 입력 인수를 만드는 데 사용됩니다(자세한 내용은 **push-interfaces.jar** 파일 참조).
3. 연합 호출을 TQL 인수와 함께 커넥터로 보냅니다.
4. 커넥터에서 보낸 들어오는 RTN 트리를 채우기와 같은 방법으로 매핑합니다. "일반 어댑터를 사용하여 데이터 채우기 수행"(271페이지)을 참조하십시오.

연합 링크 매핑

연합 링크 매핑은 채우기 링크 매핑의 경우와 같이 자동으로 수행됩니다. "자동 링크 채우기"(275페이지)를 참조하십시오.

일반 어댑터 연합 API

일반 어댑터 연합 API는 일반 어댑터 채우기 API와 매우 유사합니다. 이는 일반 연합 어댑터 Java 인터페이스가 일반 채우기 어댑터 인터페이스와 동일하기 때문입니다.

```
/**
 * <b>Description:</b><br>
 * This Interface is used for implementing
 * an adapter to allow the definition of Integration Points.<br>
 * An adapter that implements this Interface will expose
 * ths ability to run Population flows.<br>
 * The only Population flow exposed by this adapter is the Simple Flow using {@link #getDataResult(com.hp.ucmdb.federationspi.adapter.federation.
 * which will return the entire data set of data each time, and compare it to the last data set (handled by framework).<p>
 * It is highly recommended to allow better link validation by implementing {@link com.hp.ucmdb.federationspi.adapter.ReportsLinks}<br>
 * If possible, it's recommended to implement the {@link PopulationChangesAdapter} instead,
 * allowing the adapter to have better performance and control.<p>
 *
 * @see com.hp.ucmdb.federationspi.data.query.topology.TopologyFactory
 * @see com.hp.ucmdb.federationspi.adapter.ChunkTopologyResultGetter
 * @see PopulationAdapter
 * @since 10.10
 *
 */
public interface PopulationAdapter extends BasicSourceDataAdapter{

    /**
     * Retrieves the calculated result of the given {@code QueryDefinition}.<br>
     * <li>
     * This method is called by the population framework.<br>
     * </li>
     * <p>
     * This method implementation may use the {@code UpdateStatus} to report warnings for specific CIs or Relations.
     * <p/>
     * When implementing this method, it should return the result by being aware to all the conditions
     * that appear on the {@code QueryDefinition} like: topology, cardinality, id conditions and property conditions.
     * </p><br>
     * When implementing this method, one must be aware that different flows may actually be used:<br>
     * 1) A topology only flow (a query definition with conditions or id conditions but with out any layout requested).<br>
     * 2) A layout only flow (a query definition with only ids condition and layout).<br>
     * 3) A full topology flow (a query with property conditions, id conditions and layout)<br>
     *
     * @param input contains the logged in user and queryDefinition that contains the topology, conditions and layout requested by the framework
     * @return {@link com.hp.ucmdb.federationspi.adapter.federation.FederationTopologyAdapterOutput} containing the query's calculated result.
     * @throws com.hp.ucmdb.federationspi.exception.DataAccessException
     */
    public FederationTopologyAdapterOutput getDataResult(FederationTopologyAdapterInput input) throws DataAccessException;

}
```

```
import ...

/**
 * <b>Description:</b><br>
 * This Interface is used for implementing
 * an adapter to allow the definition of Integration Points.<br>
 * An adapter that implements this Interface will expose to run Federation flows.<br>
 * The adapter can report status per CI during the flow with {@link com.hp.ucmdb.federationspi.data.replication.UpdateStatus}.<br>
 * It is highly recommended to allow better link validation by implementing {@link com.hp.ucmdb.federationspi.adapter.ReportsLinks}
 * </p>
 *
 */
public interface FederationTopologyDataAdapter extends FTqlDataAdapter {

    /**
     * Retrieves the calculated result of the given {@code QueryDefinition}.<br>
     * <li>
     * This method is called by the federation framework when the user query includes any
     * federated elements(node or link).<br>
     * </li>
     * <p>
     * This method implementation may use the {@code UpdateStatus} to report warnings for specific CIs or Relations.
     * <p/>
     * When implementing this method, it should return the result by being aware to all the conditions
     * that appear on the {@code QueryDefinition} like: topology, cardinality, id conditions and property conditions.
     * </p><br>
     * When implementing this method, one must be aware that different flows may actually be used:<br>
     * 1) A topology only flow (a query definition with conditions or id conditions but with out any layout requested).<br>
     * 2) A layout only flow (a query definition with only ids condition and layout).<br>
     * 3) A full topology flow (a query with property conditions, id conditions and layout)<br>
     *
     * @param input contains the logged in user and queryDefinition that contains the topology, conditions and layout requested by th
     * @return {@link FederationTopologyAdapterOutput} containing the query's calculated result.
     * @throws com.hp.ucmdb.federationspi.exception.DataAccessException
     */
    public FederationTopologyAdapterOutput getDataResult(FederationTopologyAdapterInput input) throws DataAccessException;

}
```

연합용 일반 어댑터 커넥터 인터페이스

연합 요청에서는 채우기 요청에 사용되는 것과 동일한 메서드를 사용하므로, 동일한 채우기 커넥터 구현을 사용할 수 있습니다. 새 특성은 FlowType이라는 PopulationConnectorInput Java 클래스에 추가되었습니다. FlowType 특성에는 두 개의 값, FEDERATION 또는 POPULATION을 사용할 수 있습니다. 일반 어댑터에서는 이 특성을 기반으로 요청 유형을 파악합니다.

```
/**
 * Holds data needed to process a population request.
 *
 * @author Sergiu Indrie
 * @since 10.20
 */
public interface PopulationConnectorInput {

    QueryDefinition getQueryDefinition();

    /**
     * Indicates the required {@link com.hp.ucmdb.adapters.push.output.ResultTreeNode} structure
     * that the population result must return.
     *
     * For the target mapping <target_mapping name="lMaxMemory" datatype="LONG" value="Root.VMWare_Host_Resource['vm_memory_limit']"/>
     * the resulting tree node should have the following structure: VMWare_Host_Resource will be a child of Root and vm_memory_limit will
     * be an attribute of VMWare_Host_Resource
     *
     * @return a map containing simplified result trees
     * @see PopulationConnectorOutput#getResultTreeNodes()
     */
    Map<String, ResultTreeNodeStructure> getResultTreeNodeStructure();

    /**
     * Returns the flow type of the operation.
     * Can be POPULATION or FEDERATION
     *
     * @return the flow type
     */
    FederationTopologyAdapterInput.FlowType getFlowType();
```

```
import ...

/**
 * Population Connector that will be used by the UCMDB's Generic Population Adapter to provide the external data. The
 * data provided by the connector will be mapped to the UCMDB entities by the adapter configured mapping files.
 * <p/>
 * The Population Connector must be able to process population requests by analyzing the input query and the providing
 * corresponding data from the external system.
 *
 * @author Sergiu Indrie
 * @since 10.20
 */
public interface PopulationAdapterConnector extends GenericConnector, DynamicMappingConnector {

    /**
     * Executes a population request and provides the entities that will be populated in the format of
     * {@link com.hp.ucmdb.adapters.push.output.ResultTreeNode}
     * Used for both population and federation with the flow type flag which is present in the
     * {@link com.hp.ucmdb.adapters.population.connector.PopulationConnectorInput}
     *
     * @param input population request
     * @return a population response
     * @throws DataAccessException
     */
    PopulationConnectorOutput populate(PopulationConnectorInput input) throws DataAccessException;
```

지원되는 연합 쿼리

연합 및 채우기 쿼리는 서로 다른 폴더에 있습니다. PopulationAdapterConnector Java 인터페이스는 지원되는 채우기 및 연합 쿼리를 표시할 수 있는 다음 두 가지 메서드를 제공합니다.

- **getPopulationQueries** –현재 커넥터에서 채우기에 대해 지원하는 쿼리 컬렉션을 반환합니다.
- **getPopulationQueries** –현재 커넥터에서 연합에 대해 지원하는 쿼리 컬렉션을 반환합니다.

```
    /**
     * Returns the collection of queries the current connector supports for population
     * Note: this method is independent to the adapter life cycle (i.e. start/shutdown methods) and must work at all times.
     * <p/>
     * The method also supports return of queries with their folders path: E.g.
     * "Folder/Secondary Folder/Query Name 1"
     * "Folder/Secondary Folder/Query Name 2"
     *
     * @param env the adapter environment
     * @return a collection of the supported queries
     */
    Collection<String> getPopulationQueries(DataAdapterEnvironment env);

    /**
     * Returns the collection of queries the current connector supports for federation
     * Note: this method is independent to the adapter life cycle (i.e. start/shutdown methods) and must work at all times.
     * <p/>
     * The method also supports return of queries with their folders path: E.g.
     * "Folder/Secondary Folder/Query Name 1"
     * "Folder/Secondary Folder/Query Name 2"
     *
     * @param env the adapter environment
     * @return a collection of the supported queries
     */
    Collection<String> getFederationQueries(DataAdapterEnvironment env);
```

연합을 설정하는 방법

이 섹션의 내용은 다음과 같습니다.

어댑터 설정 구성

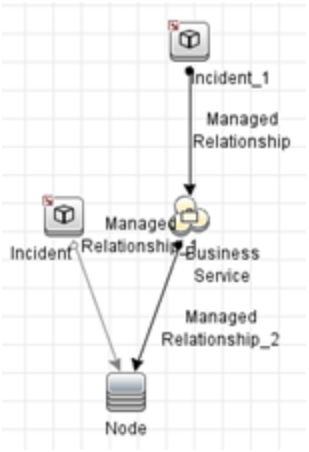
지정된 TQL 쿼리의 경우 일반 어댑터에서 TQL 쿼리의 모든 노드를 <supported-classes> 태그에 선언해야 합니다. 예를 들어, TQL 쿼리 양식이 노드에 링크된 인시던트인 경우 discoveryPatterns 폴더의 어댑터 패키지 ZIP 파일에 있는 어댑터 설정 XML 파일에 노드와 인시던트를 모두 지원되는 클래스로 선언해야 합니다.


```
<supported-classes>
    <supported-class is-derived="true" all-attributes-supported="true" name="node" is-reconciliation-supported="true"/>
    <supported-class is-derived="true" all-attributes-supported="true" name="incident" is-reconciliation-supported="true"/>
</supported-classes>
```

로그 파일에서 연합 쿼리 설정

연합 프레임워크를 사용하기 전에 몇 가지 선행 조건을 충족해야 합니다. 연합 프레임워크는 필요한 데이터를 검색하기 위해 다양한 TQL 쿼리를 사용하여 어댑터에 몇 가지 요청을 생성합니다. 연합 프레임워크에서 보내는 각 TQL 쿼리에 대해 채우기 흐름과 유사한 방식으로 어댑터에 동적 TQL 쿼리를 만들어야 합니다.

연합과 채우기의 차이점은 연합 TQL 쿼리는 프레임워크에서 동적으로 전송하므로 미리 파악할 수 없다는 점입니다. 다음 TQL 쿼리를 예로 들어 보겠습니다.



프레임워크에서 다음 쿼리를 어댑터로 보냅니다.

- 인시던트만 있는 쿼리
- 연결 유형의 관계로 노드에 링크된 인시던트가 있는 쿼리
- 멤버십 유형의 관계로 노드에 링크된 인시던트가 있는 쿼리
- 연결 유형의 관계로 비즈니스 서비스에 링크된 인시던트가 있는 쿼리
- 멤버십 유형의 관계로 비즈니스 서비스에 링크된 인시던트가 있는 쿼리

참고: 연합 엔진에서 어댑터로 보내며, 저장이 필요한 모든 쿼리의 이름은 **사용자 매핑 결합 FTQL** 입니다.

사용자 매핑 결합 FTQL 쿼리의 결과를 처리한 후 개체 특성을 검색하도록 기타 호출이 수행됩니다. 이러한 호출에는 **개체 레이아웃**이라는 쿼리가 포함됩니다. 연합 엔진은 CI의 모든 특성을 가져오려고 시도하지만 커넥터에서 해당 특성을 모두 제공할 필요는 없습니다. 매핑 파일에 필요한 하나만 반환해도 됩니다.

동일한 쿼리를 서로 다른 관계로 보내는 이유는 TQL 쿼리에서 노드와 인시던트/비즈니스 서비스 및 인시던트 사이에는 **managed_relationship** 유형 링크가 있지만, 이러한 CI 유형을 함께 링크할 때 올바른 링크는 연결과 멤버십 링크뿐이기 때문입니다.

```
<tql:link from="incident_12" to="node_10050" class="connection" name="connection_1" id="1"/>
<tql:link from="incident_12" to="node_1000050" class="connection" name="connection_2" id="2"/>
<tql:link from="datacenter_20050" to="node_10050" class="composition" name="composition_30050" id="30050"/>
```

```
<tql:link from="incident_12" to="node_10050" class="membership" name="membership_1" id="1"/>
<tql:link from="incident_12" to="node_1000050" class="membership" name="membership_2" id="2"/>
<tql:link from="datacenter_20050" to="node_10050" class="composition" name="composition_30050" id="30050"/>
```

이러한 접근법을 사용하면 링크 유형은 다르지만 거의 동일한 두 개의 TQL을 정의하는 대신 일반 **managed_relationship** 유형 링크가 있는 정적 TQL을 하나만 정의하면 됩니다.

```
2014-08-07 16:43:55,231 [AdHoc:AD_HOC_TASK_PATTERN_ID-179-1407419035224] TRACE -  >> Received federation call with the following query:
2014-08-07 16:43:55,231 [AdHoc:AD_HOC_TASK_PATTERN_ID-179-1407419035224] TRACE -  >>
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tql:query name="User mapping union FTQL" is-live="true" priority="low" xmlns:ns4="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns3="http://www.hp.com/ucmdb/1-0-0/PolicyRuleD
    <tql:node class="incident" name="incident_16" id="16">
        <tql:where>
            <tql:links>
                <tql:or>
                    <tql:link-ref name="membership_1"/>
                    <tql:link-ref name="membership_2"/>
                </tql:or>
            </tql:links>
        </tql:where>
    </tql:node>
    <tql:node class="business_service" name="business_service_10050" id="10050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="name"/>
                    <tql:list type="string">
                        <tql:string>MyBusServ</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="name"/>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:node class="business_service" name="business_service_1000050" id="1000050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="global_id"/>
                    <tql:list type="string">
                        <tql:string>183ad8038405644e67aba201334714ea</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:link from="incident_16" to="business_service_10050" class="membership" name="membership_1" id="1"/>
    <tql:link from="incident_16" to="business_service_1000050" class="membership" name="membership_2" id="2"/>
</tql:query>
```

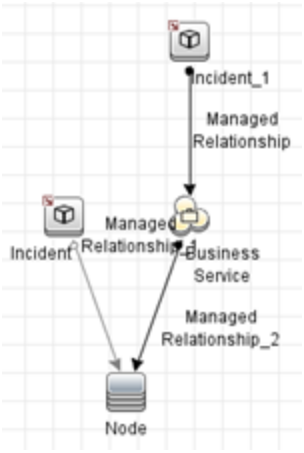
정적 TQL 쿼리는 어댑터에서 제공해야 합니다. 어댑터 개발을 지원하기 위해 연합 프레임워크에서 보낸 TQL 쿼리를 **fcmdb.push.mapping.log** 파일에 작성합니다(TRACE 로그 수준을 사용하도록 설정). 이러한 개발 작업을 용이하게 하려면 설정 <adapter-setting name="dev.mode">true</adapter-setting>을 사용합니다. 이 설정을 True로 설정하면 연합 쿼리가 실행된 후 프레임워크에서 자동으로 현재 일치하지 않는 TQL 쿼리에 대해 비어 있는 연합 결과를 만듭니다.

다음은 fcmdb.push.mapping.log의 연합 쿼리 예입니다.

```
2014-08-07 16:43:55,231 [AdHoc:AD_HOC_TASK_PATTERN_ID-179-1407419035224] TRACE -  >> Received federation call with the following query:
2014-08-07 16:43:55,231 [AdHoc:AD_HOC_TASK_PATTERN_ID-179-1407419035224] TRACE -  >>
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tql:query name="User mapping union FTQL" is-live="true" priority="low" xmlns:ns4="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns3="http://www.hp.com/ucmdb/1-0-0/PolicyRuleD
    <tql:node class="incident" name="incident_16" id="16">
        <tql:where>
            <tql:links>
                <tql:or>
                    <tql:link-ref name="membership_1"/>
                    <tql:link-ref name="membership_2"/>
                </tql:or>
            </tql:links>
        </tql:where>
    </tql:node>
    <tql:node class="business_service" name="business_service_10050" id="10050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="name"/>
                    <tql:list type="string">
                        <tql:string>MyBusServ</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="name"/>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:node class="business_service" name="business_service_1000050" id="1000050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="global_id"/>
                    <tql:list type="string">
                        <tql:string>183ad8038405644e67aba201334714ea</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:link from="incident_16" to="business_service_10050" class="membership" name="membership_1" id="1"/>
    <tql:link from="incident_16" to="business_service_1000050" class="membership" name="membership_2" id="2"/>
</tql:query>
```

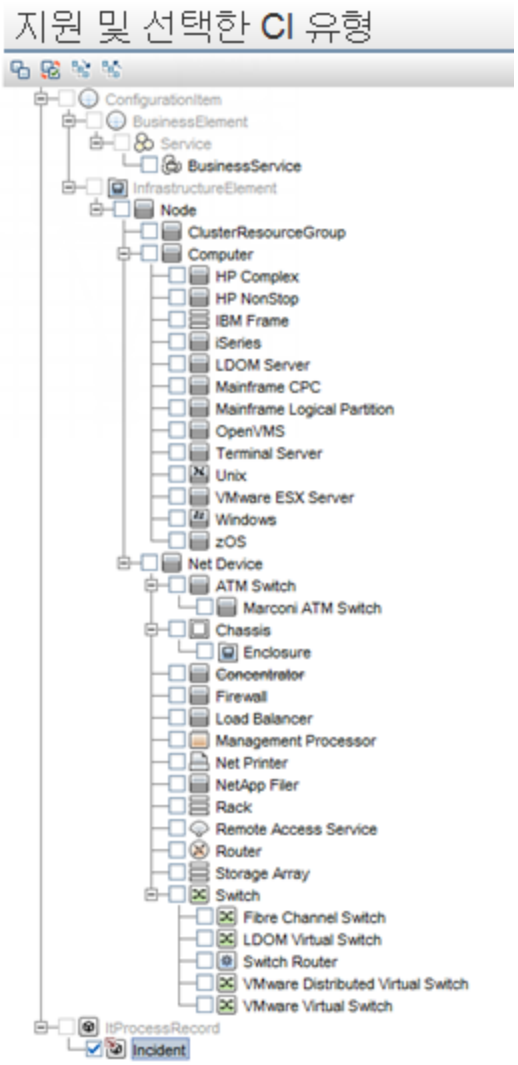
연합 설정 예

예에서는 다음의 연합 TQL을 사용합니다.



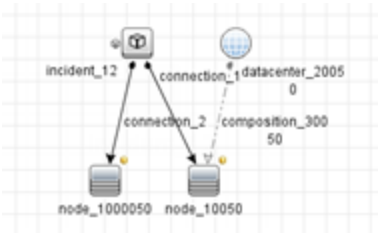
이 TQL 쿼리의 경우 어댑터에서 discoveryPatterns 폴더의 어댑터 패키지 ZIP 파일에 있는 어댑터 설정 XML 파일에 지원되는 클래스를 선언해야 합니다. 지원되는 클래스는 node, incident 및 business_service입니다.

아래 표시된 대로 통합 스튜디오의 **연합** 탭에서 인시던트를 선택해야 합니다.

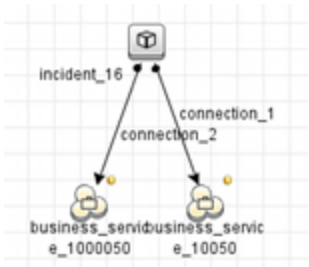


이 TQL 쿼리의 경우 어댑터에 다음 세 가지 TQL 쿼리가 있어야 합니다.

인시던트



노드에 링크된 인시던트



business_service에 링크된 인시던트

정적 TQL을 가져오는 방법에 대한 자세한 내용은 "로그 파일에서 연합 쿼리 설정"(288페이지)을 참조하십시오. 이러한 TQL 쿼리에 UCMDB에 존재하는 데이터에 종속적인 조건이 있는 경우에도 TQL 쿼리의 구조 또는 매핑 수행 방법에는 영향을 미치지 않습니다.

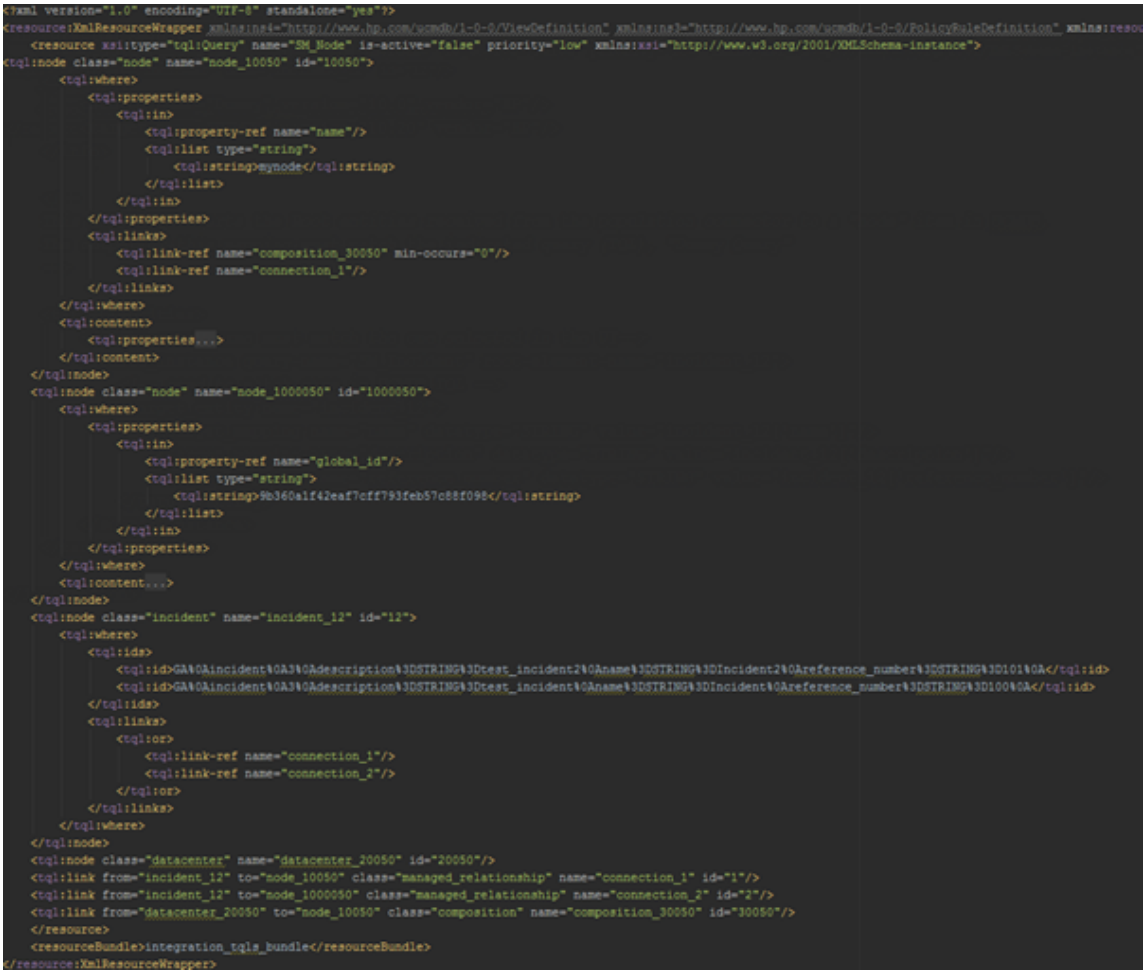
이러한 TQL 쿼리 각각에 대해 다음과 같이 어댑터에 매핑 파일이 필요합니다.

- 인시던트

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<resource:XmlResourceWrapper xmlns:ns1="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns2="http://www.hp.com/ucmdb/1-0-0/PolicyRuleDefiniti
    <resource xsi:type="tql:Query" name="SM_Incident" is-active="false" priority="low" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<tql:node class="incident" name="incident_12" id="12"/>
    </resource>
    <resourceBundle>integration_tqls_bundle</resourceBundle>
</resource:XmlResourceWrapper>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<integration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../generic-adapter.xsd">
    <!-- add scheme reference -->
    <info>
        <source name="Dummy" version="10.0" vendor="HP"/>
        <target name="UCMDB" version="10.20" vendor="HP"/>
    </info>

    <!--
    This mapping converts the Root entities received from the population connector to a "node" item in UCMDB,
    The output of this mapping must match the indicated query (TQL), "Dummy Query"
    -->

    <target_entities>
        <!--The query name must match the one selected in the UI-->
        <source_instance query-name="SM_Incident" root-element-name="incident_12">
            <!-- need to match case in UCMDB TQL -->
            <target_entity name="incident_12">
                <target_mapping name="name" datatype="STRING" value="incident_12['name']"/>
                <target_mapping name="description" datatype="STRING" value="incident_12['description']"/>
                <target_mapping name="reference_number" datatype="STRING" value="incident_12['reference_number']"/>
            </target_entity>
        </source_instance>
    </target_entities>

</integration>
```

- **노드에 링크된 인시던트**



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<resource:XmlResourceWrapper xmlns:ns="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns1="http://www.hp.com/ucmdb/1-0-0/PolicyRuleDefinition" xmlns:resou
    <resource xsi:type="tql:Query" name="SM_Node" is-active="false" priority="low" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<tql:node class="node" name="node_10050" id="10050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="name"/>
                    <tql:list type="string">
                        <tql:string>mynode</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
            <tql:links>
                <tql:link-ref name="composition_30050" min-occurs="0"/>
                <tql:link-ref name="connection_1"/>
            </tql:links>
        </tql:where>
        <tql:content>
            <tql:properties...>
        </tql:content>
    </tql:node>
    <tql:node class="node" name="node_1000050" id="1000050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="global_id"/>
                    <tql:list type="string">
                        <tql:string>9b360a1f42eaf7cff793feb57c88f098</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content...>
    </tql:node>
    <tql:node class="incident" name="incident_12" id="12">
        <tql:where>
            <tql:ids>
                <tql:id>GA%0Aincident%0A3%0Adescription%3DSTRING%3Dtest_incident2%0Aname%3DSTRING%3DIncident2%0Areference_number%3DSTRING%3D101%0A</tql:id>
                <tql:id>GA%0Aincident%0A3%0Adescription%3DSTRING%3Dtest_incident%0Aname%3DSTRING%3DIncident%0Areference_number%3DSTRING%3D100%0A</tql:id>
            </tql:ids>
            <tql:links>
                <tql:or>
                    <tql:link-ref name="connection_1"/>
                    <tql:link-ref name="connection_2"/>
                </tql:or>
            </tql:links>
        </tql:where>
    </tql:node>
    <tql:node class="datacenter" name="datacenter_20050" id="20050"/>
    <tql:link from="incident_12" to="node_10050" class="managed_relationship" name="connection_1" id="1"/>
    <tql:link from="incident_12" to="node_1000050" class="managed_relationship" name="connection_2" id="2"/>
    <tql:link from="datacenter_20050" to="node_10050" class="composition" name="composition_30050" id="30050"/>
    </resource>
    <resourceBundle>integration_tqls_bundle</resourceBundle>
</resource:XmlResourceWrapper>
```

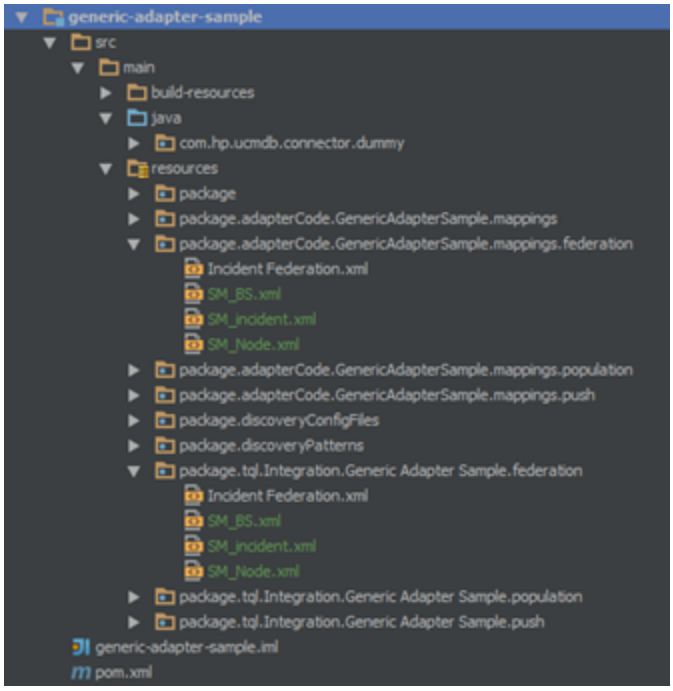
```
<?xml version="1.0" encoding="UTF-8"?>
<integration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../generic-adapter.xsd">
    <!-- add scheme reference -->
    <info>
        <source name="Dummy" version="10.0" vendor="HP"/>
        <target name="UCMDB" version="10.20" vendor="HP"/>
    </info>
    <target_entities>
        <!--The query name must match the one selected in the UI-->
        <source_instance query-name="SM_Node" root-element-name="Computer">
            <!-- need to match case in UCMDB TQL -->
            <target_entity name="node_1000050">
                <target_mapping name="name" datatype="STRING" value="Computer['name']"/>
            </target_entity>


            <for-each-source-entity count-index="i" source-entities="Computer.incident_12" var-name="currIP">
                <target_entity name="incident_12">
                    <target_mapping name="name" datatype="STRING" value="currIP['name']"/>
                    <target_mapping name="description" datatype="STRING" value="currIP['description']"/>
                    <target_mapping name="reference_number" datatype="STRING" value="currIP['reference_number']"/>
                </target_entity>
            </for-each-source-entity>


        </source_instance>
    </target_entities>
</integration>
```

- business_service에 대한 인시던트

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<resource:XmlResourceWrapper xmlns:ns4="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns3="http://www.hp.com/ucmdb/1-0-0/PolicyBu
    <resource xsi:type="tql:Query" name="SM_BS" is-active="false" priority="low" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <tql:node class="incident" name="incident_16" id="16">
        <tql:where>
            <tql:links>
                <tql:or>
                    <tql:link-ref name="connection_1"/>
                    <tql:link-ref name="connection_2"/>
                </tql:or>
            </tql:links>
        </tql:where>
    </tql:node>
    <tql:node class="business_service" name="business_service_10050" id="10050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="name"/>
                    <tql:list type="string">
                        <tql:string>MyBusServ</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="name"/>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:node class="business_service" name="business_service_1000050" id="1000050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="global_id"/>
                    <tql:list type="string">
                        <tql:string>183ad8038405644e67aba201334714ea</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
        <tql:link from="incident_16" to="business_service_10050" class="managed_relationship" name="connection_1" id="1"/>
        <tql:link from="incident_16" to="business_service_1000050" class="managed_relationship" name="connection_2" id="2"/>
    </resource>
    <resourceBundle>integration_tqls_bundle</resourceBundle>
</resource:XmlResourceWrapper>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<integration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../generic-adapter.xsd">
    <!-- add scheme reference -->
    <info>
        <source name="Dummy" version="10.0" vendor="HP"/>
        <target name="UCMDB" version="10.20" vendor="HP"/>
    </info>
    <target_entities>
        <!--The query name must match the one selected in the UI-->
        <source_instance query-name="SM_BS" root-element-name="BS">
            <!-- need to match case in UCMDB TQL -->
            <target_entity name="business_service_10050">
                <target_mapping name="name" datatype="STRING" value="BS['name']"/>
            </target_entity>
            <for-each-source-entity count-index="i" source-entities="BS.incident_16" var-name="currIP">
                <target_entity name="incident_16">
                    <target_mapping name="name" datatype="STRING" value="currIP['name']"/>
                    <target_mapping name="description" datatype="STRING" value="currIP['description']"/>
                    <target_mapping name="reference_number" datatype="STRING" value="currIP['reference_number']"/>
                </target_entity>
            </for-each-source-entity>
        </source_instance>
    </target_entities>
</integration>
```

이러한 TQL 쿼리 및 매핑 파일이 아래 표시된 대로 어댑터에 있어야 합니다.



조정

일반 어댑터 프레임워크를 사용하여 데이터를 채우거나 연합할 때 UCMDB에 허용될 수 있도록 CI에 항상 필수 조정 데이터가 있어야 합니다. 컨테이너 CI 유형이 필요한 CI 유형(예: 실행 중인 소프트웨어)을 채울 때는 항상 필요한 컨테이너 필드(예: **root_container_name** 및 **product_name**)와 컨테이너 CI(예: **Node**)를 채우는지 확인합니다. 루트 컨테이너에 종속적인 CI를 채우려면 같은 단계에서 명시적 링크 채우기 또는 두 CI 간 자동 완료 링크 채우기를 사용하여 CI, 해당 루트 컨테이너 및 이들 간의 링크를 만들어야 합니다.

또한 채워지거나 연합 CI를 매핑할 때에는 **global_id** 특성 매핑을 고려하는 것이 좋습니다. 이는 UCMDB 조정 엔진에 큰 도움이 되며, 정확한 CI 조정을 보장합니다.

일반 어댑터 API

일반 어댑터 프레임워크로 노출되는 API는 다음과 같습니다.

<UCMDB_Server>\lib\push-interfaces.jar
<UCMDB_Server>\lib\integrationFramework\GenericAdapter\generic-adapter-api-factory.jar

일반 어댑터 인스턴스 개발에는 다음과 같이 연합 API도 필요할 수 있습니다.

<UCMDB_Server>\lib\federation-api.jar

리소스 로케이터 API

리소스 로케이터 API는 일반 어댑터 작업을 편집할 때 사용됩니다. 선택한 작업의 TQL 쿼리와 관련된 어댑터 리소스를 찾을 수 있도록 일반 및 채우기 리소스 로케이터 API를 구현합니다.

다음 이미지는 GenericConnector Java 인터페이스의 일반 리소스 로케이터 API를 보여 줍니다.

```
/**
 * This methods reports general resources used by the adapter.<br>
 * This allows editing these resources directly from the Integration Studio.
 *
 *
 * @param env the Adapter's environment
 * @param input  the requested information
 * @param output the returned resources
 * @see com.hp.ucmdb.federationspi.adapter.resource.GeneralResourcesLocator
 */
void locateGeneralResources(DataAdapterEnvironment env, LocateGeneralResourcesInput input, LocateGeneralResourcesOutput output);
```

다음 이미지는 PopulationAdapterConnector Java 인터페이스의 채우기 쿼리 리소스 로케이터 API를 보여 줍니다.

```
/**
 * This methods reports population queries resources used by the adapter.<br>
 * This allows editing these resources directly from the Integration Studio.
 *
 * @param input  the requested information
 * @param output the returned resources
 * @see com.hp.ucmdb.federationspi.adapter.resource.PushQueriesResourceLocator
 */
void locatePopulationQueriesResources(DataAdapterEnvironment env, LocatePopulationQueriesResourcesInput input,
                                      LocatePopulationQueriesResourcesOutput output);
```

작업의 TQL 쿼리와 관련된 리소스를 보려면 다음을 수행합니다.

1. 통합 스튜디오에서 통합 포인트를 선택합니다.
2. 통합 작업 창에서 작업을 선택하고 **쿼리 리소스 편집**  을 클릭합니다.

일반 어댑터 패키지 만들기

일반 어댑터 패키지는 강화된 일반 밀어넣기 어댑터 패키지와 유사합니다. ZIP 아카이브의 초기 구조를 만들려면 기존의 일반 어댑터 패키지를 복사한 후 필요에 따라 사용자 지정하는 것이 좋습니다. 어댑터 패키지에 대한 자세한 내용은 "일반 어댑터를 사용하여 데이터 밀어넣기 수행"(262페이지)을 참조하십시오.

기존의 강화된 일반 밀어넣기 어댑터 패키지와 일반 어댑터 패키지의 차이점은 다음과 같습니다.

- 어댑터 XML 차이점
  - 어댑터 클래스가 다음과 같이 **PushAdapter**에서 **GenericAdapter**로 변경되었습니다.

```
<className>com.hp.ucmdb.adapters.push.PushAdapter</className>
```

```
<className>com.hp.ucmdb.adapters.GenericAdapter</className>
```

- 어댑터 기능에 채우기가 포함됩니다.

```
<support-replicatioin-data>
    <source>
        <push-back-ids/>
        <instance-based-data/>
        <population-queries-resources-locator/>
    </source>
```

- 다음과 같이 채우기 커넥터 정의와 함께 어댑터 설정에 의해 수행됩니다.

```
<adapter-setting name="PopulationConnector.class.name">com.hp.ucmdb.connector.dummy.DummyPopulationConnector</adapter-setting>
```

일반 어댑터(채우기 기능 사용)에도 다음과 같이 밀어넣기 커넥터 클래스에 대한 정의가 있어야 합니다.

```
<adapter-setting name="PushConnector.class.name">com.hp.ucmdb.connector.dummy.DummyPushConnector</adapter-setting>
```

- 매핑 파일 폴더

  강화된 일반 밀어넣기 어댑터는 해당 매핑 파일이 **<adapter_package_zip>/adapterCode/<*adapter_name*>/mappings** 폴더에 있어야 하는 것과 대조적으로, 일반 어댑터는 해당 매핑이 별도의 세 개 폴더(밀어넣기, 채우기, 연합에 대해 하나씩)에 있어야 합니다. 필요한 폴더는 다음과 같습니다.

  <adapter_package_zip>/adapterCode/<*어댑터 이름*>/mappings/push
  <adapter_package_zip>/adapterCode/<*어댑터 이름*>/mappings/population
  <adapter_package_zip>/adapterCode/<*어댑터 이름*>/mappings/federation

  여기서 **<adapter_package_zip>**은 일반 어댑터 패키지에 대해 생성할 ZIP 아카이브를 나타냅니다.

참고: 일반 어댑터는 세 가지 유형의 데이터 동기화(밀어넣기, 채우기, 연합)를 모두 지원하지만, 특정 일반 어댑터에서 해당 유형의 하위 집합만 지원하도록 선택할 수 있습니다.

기존 어댑터에서 새 어댑터를 만들 때 유의할 점

- **TestAdapter\discoveryPatterns\TestAdapter.xml**
- 다음과 같이 **TestAdapter.xml** 파일을 수정합니다.
  - <pattern xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="TestAdapter" xsi:noNamespaceSchemaLocation="../../Patterns.xsd" description="..." schemaVersion="9.0" displayName="...">
  - <adapter-id>TestAdapter</adapter-id>
- 새 어댑터가 포함된 ZIP 파일은 이름이 어댑터 자체(TestAdapter)와 같아야 합니다.

어댑터 패키지 빌드

어댑터 패키지에 다음 폴더가 포함되어 있는지 확인합니다.

- **adapterCode.** 이 폴더 아래에 PushExampleAdapter.java에서 만들어진 .jar 파일을 포함하는 **PushExampleAdapter**라는 폴더를 만듭니다. 여기에는 이전에 만든 매핑 파일인 **computerIPMapping.xml**을 넣을 수 있는 **mappings**라는 폴더도 포함됩니다. **PushFunctions.groovy** 파일을 포함하는 **scripts**라는 폴더도 포함해야 합니다.
- **discoveryConfigFiles.** UpdateResult 사용에 관한 오류를 보고하는 데 사용되는 오류 코드 등의 구성 파일을 포함합니다. 이 예제에서는 폴더가 비어 있습니다.
- **discoveryPatterns. push_example_adapter.xml**을 포함합니다.
- **tql.** 예에서 만든 TQL 쿼리를 포함합니다. 이 폴더는 선택 사항이지만 패키지를 배포할 때 TQL 쿼리가 자동으로 만들어집니다.

어댑터 수준의 특성 및 링크 유효성 검사 사용/사용 안 함

다음 설정을 추가하여 일반 어댑터에 대해 어댑터 수준의 특성 및 링크 유효성 검사를 사용 또는 사용하지 않도록 설정할 수 있습니다.

```
<adapter-settings>
    <adapter-setting name="enable.attributes.links.validation">true</adapter-setting>
</adapter-settings>
```

특성 및 링크의 어댑터 수준 유효성 검사를 사용하려면 어댑터 설정 **enable.attributes.links.validation**을 **true**로 설정합니다.

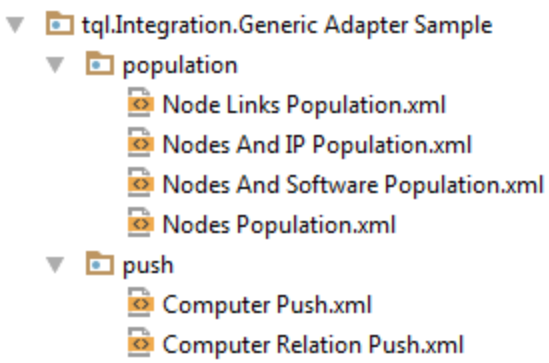
특성 및 링크의 어댑터 수준 유효성 검사를 사용하지 않으려면 어댑터 설정 **enable.attributes.links.validation**을 **false**로 설정합니다.

참고: 설정하지 않는 경우 기본값은 **true**이며, 기본적으로 특성 및 링크 유효성 검사를 사용합니다.

채우기 TQL 쿼리

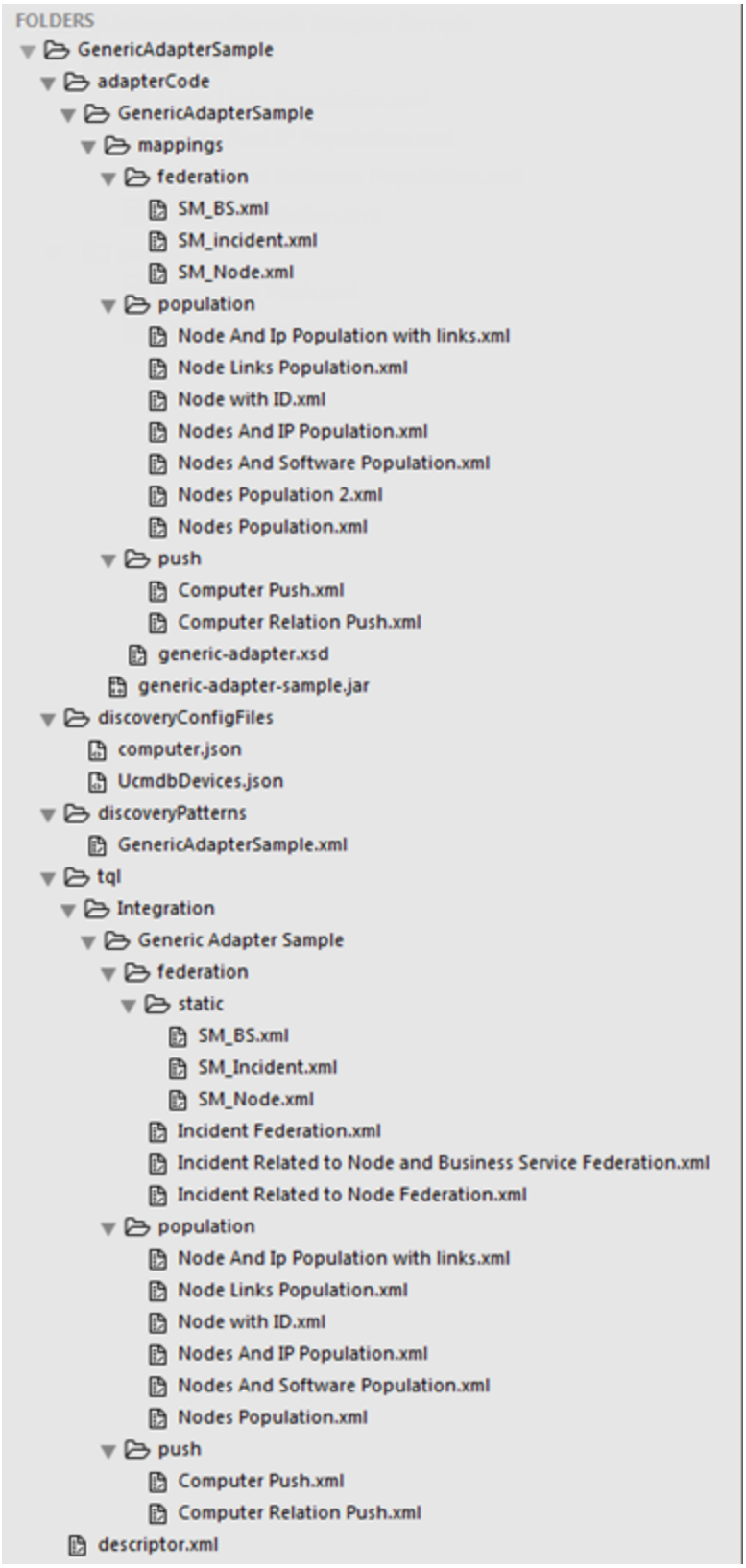
채우기 작업에 사용할 TQL 쿼리는 일반 어댑터의 ZIP 아카이브에 포함하여 어댑터를 통해 UCMDB에 배포해야 합니다. 표시된 TQL 쿼리는 채우기 흐름 중 채우기 요청이 발생할 때 UCMDB에 있어야 합니다.

이러한 TQL 쿼리는 **<zip>/tql/<folder_1>/../<folder_n>**에 포함되어 있어야 합니다. 다음은 폴더 구조의 예입니다.



채우기 TQL 쿼리가 위에 표시된 폴더에 있더라도 채우기 커넥터에서도 지원되는 채우기 TQL 쿼리를 Java 인터페이스의 해당 메서드에서 확인해야 합니다. 자세한 내용은 "채우기 커넥터"(276페이지)를 참조하십시오.

샘플 패키지


FOLDERS
GenericAdapterSample
adapterCode
GenericAdapterSample
mappings
federation
SM_BS.xml
SM_incident.xml
SM_Node.xml
population
Node And Ip Population with links.xml
Node Links Population.xml
Node with ID.xml
Nodes And IP Population.xml
Nodes And Software Population.xml
Nodes Population 2.xml
Nodes Population.xml
push
Computer Push.xml
Computer Relation Push.xml
generic-adapter.xsd
generic-adapter-sample.jar
discoveryConfigFiles
computer.json
UcmdbDevices.json
discoveryPatterns
GenericAdapterSample.xml
tql
Integration
Generic Adapter Sample
federation
static
SM_BS.xml
SM_Incident.xml
SM_Node.xml
Incident Federation.xml
Incident Related to Node and Business Service Federation.xml
Incident Related to Node Federation.xml
population
Node And Ip Population with links.xml
Node Links Population.xml
Node with ID.xml
Nodes And IP Population.xml
Nodes And Software Population.xml
Nodes Population.xml
push
Computer Push.xml
Computer Relation Push.xml
descriptor.xml

밀어넣기와 채우기 매핑의 차이

밀어넣기 및 채우기 매핑 파일의 기본 XML 스키마는 동일하지만 파일은 약간 다르게 해석됩니다. 자세한 내용은 "일반 어댑터 XML 스키마 참조"(305페이지)를 참조하십시오.

다음 밀어넣기 매핑 예는 다음과 같이 해석됩니다. "Computer Push" TQL 쿼리(UCMDB에서 실행)의 결과를 사용하고 루트 트리 구조로 표시되며 나중에 AM으로 전송될 amComputer 엔터티를 만듭니다.

```
<target_entities>
    <!--The query name must match the one selected in the UI-->
    <source_instance query-name="Computer Push" root-element-name="Root">
        <target_entity name="amComputer">
            <target_mapping name="TcpIpHostName" datatype="STRING" value="Root['name']"/>
            <target_mapping name="ComputerDesc" datatype="STRING" value="Root['os_description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

다음 채우기 매핑 예는 다음과 같이 해석됩니다. "Nodes Population" TQL 쿼리(외부 시스템에서 실행)의 결과를 사용하고 PC 트리 구조로 표시되며 나중에 UCMDB에 추가될 UCMDB 루트 엔터티(TQL 쿼리에 표시된 대로 Node 유형)를 만듭니다.

```
<target_entities>
    <!--The query name must match the one selected in the UI-->
    <source_instance query-name="Nodes Population" root-element-name="PC">
        <!-- need to match case in UCMDB class model-->
        <target_entity name="Root">
            <target_mapping name="name" datatype="STRING" value="PC['name']"/>
            <target_mapping name="description" datatype="STRING" value="PC['description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

일반 어댑터 로그 파일

문제 해결 및 디버깅을 위해 다음을 사용합니다.

- 다음 파일의 로깅 수준을 조정합니다(최대 상세 결과의 *loglevel* 변수를 TRACE로 설정).
  - **<UCMDB_DataFlowProbe>\conf\log\fcmdb.push.properties**

    <UCMDB_DataFlowProbe>는 UCMDB Data Flow Probe 설치 디렉터리입니다.
  - **<UCMDB_Server>\conf\log\reconciliation.properties**

    <UCMDB_Server>는 UCMDB 서버 설치 디렉터리입니다.

- 다음 일반 어댑터 로그 파일을 분석합니다.
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.all.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.configuration.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.connector.all.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.connector.configuration.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.mapping.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.all.log**
- 다음 일반 로그 파일을 분석합니다.
  - **<UCMDB_DataFlowProbe>\runtime\log\probe-error.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\WrapperProbeGw.log**
  - **<UCMDB_Server>\runtime\log\error.log**
  - **<UCMDB_Server>\runtime\log\cmdb.reconciliation.log**

일반 어댑터 프레임워크를 사용하는 어댑터

구현 지침으로 UCMDB와 함께 제공된 다음 어댑터를 사용자 지정 일반 어댑터 개발의 참고 자료로 참조하십시오. 어댑터 개발 속도를 높일 수 있습니다.

- Asset Manager 어댑터
- Service Manager 어댑터

일반 어댑터 XML 스키마 참조

일반 어댑터 XML 스키마는 **schema** 디렉터리의 **cmdb.jar** 파일에 있습니다. 일반 어댑터 매핑 파일을 외부 편집기에서 작성하는 동안 스키마 파일을 참조해야 합니다. XSD 파일의 전체 경로는 다음과 같습니다.

<UCMDB_Server_Install_dir>/lib/cmdb.jar/schema/generic-adapter.xsd

II부: API 사용

9장: API 소개

이 장의 내용:

API 개요

HP Universal CMDB에 포함된 API는 다음과 같습니다.

- **UCMDB Java API.** 타사 또는 사용자 지정 도구에서 Java API를 사용하여 데이터 및 계산을 추출하는 방법 및 UCMDB(Universal Configuration Management database)에 데이터를 쓰는 방법을 설명합니다. 자세한 내용은 "HP Universal CMDB API"(308페이지)를 참조하십시오.
- **UCMDB 웹 서비스 API.** 구성 항목 정의 및 UCMDB에 대한 토폴로지 관계를 작성하고 TQL 및 Ad hoc 쿼리로 정보를 쿼리할 수 있습니다. 자세한 내용은 "HP Universal CMDB 웹 서비스 API"(316페이지)를 참조하십시오.
- **데이터 흐름 관리 Java API.** 데이터 흐름 관리의 프로브, 작업, 트리거 및 자격 증명을 관리할 수 있습니다. 자세한 내용은 "데이터 흐름 관리 Java API"(349페이지)를 참조하십시오.
- **데이터 흐름 관리 웹 서비스 API.** 데이터 흐름 관리의 프로브, 작업, 트리거 및 자격 증명을 관리할 수 있습니다. 자세한 내용은 "데이터 흐름 관리 웹 서비스 API"(351페이지)를 참조하십시오.

참고: 전체 API 문서를 보려면 온라인 문서에 액세스하는 것이 좋습니다. PDF 버전에는 html 형식으로 생성되는 API 문서의 링크가 없습니다.

10장: HP Universal CMDB API

이 장의 내용:

규칙

이 장에서는 다음 규칙을 사용합니다.

- UCMDB는 Universal 구성 관리 데이터베이스 자체를 참조합니다. HP Universal CMDB는 해당 응용 프로그램을 참조합니다.
- UCMDB 요소 및 메서드 인수는 인터페이스에 지정된 경우에만 의미가 있습니다.

사용 가능한 API에 관한 전체 문서는 HP UCMDB API Reference를 참조하십시오.

이러한 파일은 다음 폴더에 있습니다.

\\<UCMDB 루트 디렉터리>\hp\UCMDB\UCMDBServer\deploymdb-docs\docs\eng\APIs\UCMDB_JavaAPI\index.html

HP Universal CMDB API 사용

참고: 이 장은 온라인 문서 라이브러리에서 볼 수 있는 API Javadoc와 함께 사용하십시오.

HP Universal CMDB API는 응용 프로그램을 Universal CMDB(CMDB)와 통합하는 데 사용됩니다. API는 다음을 수행할 수 있는 메서드를 제공합니다.

- CMDB에서 CI 및 관계 추가, 제거, 업데이트
- 클래스 모델에 대한 정보 검색
- UCMDB 기록 내역에서 정보 검색
- 가상 시나리오 실행
- 구성 항목 및 관계에 대한 정보 검색

구성 항목 및 관계에 대한 정보를 검색하는 메서드는 일반적으로 TQL(토폴로지 쿼리 언어)을 사용합니다. 자세한 내용은 *HP Universal CMDB 모델링 안내서*에서 토폴로지 쿼리 언어를 참조하십시오.

HP Universal CMDB API의 사용자는 다음 사항을 잘 알고 있어야 합니다.

- Java 프로그래밍 언어
- HP Universal CMDB

이 섹션에는 다음 항목이 포함됩니다.

API의 사용

API는 여러 비즈니스 요구 사항을 만족하는 데 사용됩니다. 예를 들어 사용 가능한 CI(구성 항목)에 대한 정보를 받기 위해 타사 시스템에서 클래스 모델을 쿼리할 수 있습니다. 기타 사용 사례는 "UCMDB API 사용 사례"(313페이지)를 참조하십시오.

사용 권한

관리자는 API에 연결하는 데 필요한 로그인 자격 증명을 제공합니다. API 클라이언트에는 CMDB에서 정의된 통합 사용자의 사용자 이름과 비밀번호가 필요합니다. 이러한 사용자는 CMDB의 실제 사용자가 아니라 CMDB에 연결하는 응용 프로그램을 나타냅니다.

또한 사용자가 로그인하기 위해서는 **SDK에 액세스** 일반 수행 권한이 있어야 합니다.

> **주의:** API 인증 권한이 있는 일반 사용자도 API 클라이언트를 사용할 수 있습니다. 하지만 이 옵션은 권장되지 않습니다.

자세한 내용은 "통합 사용자 만들기"(312페이지)를 참조하십시오.

응용 프로그램의 일반 구조

정적 팩터리는 UcmdbServiceFactory 하나뿐입니다. 이 팩터리는 응용 프로그램의 진입점입니다. UcmdbServiceFactory는 getServiceProvider 메서드를 노출합니다. 이러한 메서드는 **UcmdbServiceProvider** 인터페이스의 인스턴스를 반환합니다.

클라이언트가 인터페이스 메서드를 사용하여 다른 개체를 만듭니다. 예를 들어 새 쿼리 정의를 만들기 위해 클라이언트는 다음 작업을 수행합니다.

1. 주 CMDB 서비스 개체에서 쿼리 서비스를 가져옵니다.
2. 서비스 개체에서 쿼리 팩터리 개체를 가져옵니다.
3. 팩터리에서 새 쿼리 정의를 가져옵니다.

```
UcmdbServiceProvider provider =
  UcmdbServiceFactory.getServiceProvider(HOST_NAME, PORT);
UcmdbService ucmdbService =
```

```
  provider.connect(provider.createCredentials(USERNAME,
  PASSWORD), provider.createClientContext("Test"));
TopologyQueryService queryService = ucmdbService.getTopologyQueryService();
TopologyQueryFactory factory = queryService.getFactory();
QueryDefinition queryDefinition = factory.createQueryDefinition("Test Query");
queryDefinition.addNode("Node").ofType("host");
Topology topology = queryService.executeQuery(queryDefinition);
System.out.println("There are " + topology.getAllCIs().size() + " hosts in uCMDB");
```

UcmdbService에서 사용 가능한 서비스는 다음과 같습니다.

서비스 메서드	사용
getAuthorizationModelService	인증 작업(사용자 및 사용자 그룹 만들기, 사용자 및 그룹에 역할 할당 등) 수행
getClassModelService	CI 및 관계 유형에 대한 정보
getConfigurationService	인프라 설정 관리, 서버 구성용
getDataStoreMgmtService	연합할 CI 및 특성을 비롯한 데이터 저장소 정보 쿼리
getDDMConfigurationService	데이터 흐름 관리 시스템 구성
getDDMManagementService	데이터 흐름 관리 시스템의 진행, 결과 및 오류를 분석하고 보기
getDDMZoneService	관리 영역 가져오기 및 내보내기(해당 활동 포함)
getHistoryService	모니터링된 CI의 기록 내역에 대한 정보(변경, 제거 등)
getImpactAnalysisService	영향 분석 시나리오(**상관 관계**라고도 함) 실행
getLicensingService	시스템에 설치된 라이센스에 대한 정보 쿼리
getMultipleCMDBService	글로벌 ID 및 UCMDB ID 간 전환
getMultiTenancyService	테넌트 만들기, 읽기, 업데이트 및 삭제
getPersistencyService	이진 데이터를 키-값 쌍으로 지속시킴
getQueryManagementService	쿼리에 대한 액세스 관리 - 기존 쿼리를 저장, 삭제 및 나열. 쿼리 유효성 검사 및 쿼리 종속 관계 디스커버리도 제공
getReconciliationService	식별 및 병합 기능 제공
getResourceBundleManagementService	리소스 태깅("번들링" 서비스). 태그가 지정된 모든 리소스에서 명시적인 새 태그 생성 및 태그 삭제 가능

서비스 메서드	사용
getResourceManagementService	시스템에 TQL 쿼리, 보기, 사용자 등의 리소스 패키지 배포
getSecurityService	자격 증명이 유효한지 여부 확인
getServerService	시스템에 대한 일반 정보 쿼리
getSnapshotService	스냅샷 관리를 위한 서비스 제공(가져오기, 저장, 비교 등)
getSoftwareSignatureService	데이터 흐름 관리 시스템에 의해 디스커버리할 소프트웨어 항목 정의
getStateService	상태 관리를 위한 서비스 제공(목록, 추가, 제거 등)
getSystemHealthService	시스템 상태 서비스 제공(기본 시스템 성능 표시기, 용량 및 가용성 메트릭)
getTopologyQueryService	IT 유니버스에 대한 정보 가져오기
getTopologyUpdateService	IT 유니버스에서 정보 변경
getUcmdbVersion	UCMDB와 컨텐츠 팩 버전 및 빌드 정보 쿼리
getViewArchiveService	보기 결과 아카이브 서비스. 현재 보기 결과를 저장하고 이전에 저장한 결과 검색 가능
getViewService	보기 실행 서비스(정의 실행, 저장된 보기 실행) 및 관리 서비스(기존 보기 저장, 삭제 및 나열). 보기 유효성 검사 및 종속 관계 디스커버리도 제공

클라이언트가 HTTP(S)를 통해 서버와 통신합니다.

클래스 경로에 API Jar 파일 저장

이 API 집합을 사용하려면 **ucmdb-api.jar** 파일이 필요합니다. 웹 브라우저에 http://<localhost>:8080을 입력하고(여기서 localhost는 UCMDB가 설치된 시스템) **API Client Download** 링크를 클릭하여 파일을 다운로드할 수 있습니다.

응용 프로그램을 컴파일하거나 실행하기 전에 **.jar** 파일을 클래스 경로에 저장합니다.

참고: UCMDB Java API Jar을 사용하려면 JRE 버전 6 이상이 설치되어 있어야 합니다.

통합 사용자 만들기

다른 제품과 UCMDB 간의 통합을 위한 전용 사용자를 만들 수 있습니다. 이 사용자를 통해 UCMDB 클라이언트 SDK를 사용하여, 서버 SDK에 인증되고 API를 실행하는 제품을 사용할 수 있습니다. 이 API 집합을 사용하여 작성한 응용 프로그램은 통합 사용자 자격 증명으로 로그온해야 합니다.

주의: 또한 일반 UCMDB 사용자(예: admin)에 연결할 수도 있습니다. 하지만 이 옵션은 권장되지 않습니다. UCMDB 사용자와 연결하려면 API 인증 권한을 부여해야 합니다.

통합 사용자를 만들려면 다음을 수행합니다.

1. 웹 브라우저를 시작하고 서버 주소를 다음과 같이 입력합니다.

   http://localhost:8080/jmx-console

   사용자 이름과 비밀번호를 입력하여 로그인해야 할 수도 있습니다.
2. UCMDB에서 **service=Authorization Services**를 클릭합니다.
3. **createUser** 작업을 찾습니다. 이 메서드는 다음 매개 변수를 수락합니다.
   - **customerID**. 고객 ID입니다.
   - **username**. 통합 사용자의 이름입니다.
   - **userDisplayName**. 통합 사용자의 표시 이름입니다.
   - **userLoginName**. 통합 사용자의 로그인 이름입니다.
   - **password**. 통합 사용자의 비밀번호입니다.

     기본 비밀번호 정책에 따라 UCMDB 비밀번호에 다음 네 가지 문자 유형을 각각 하나 이상 포함해야 합니다.
     - 영문 대문자
     - 영문 소문자
     - 숫자
     - 기호 문자 ,\:/. _?&%=+-[]()|

     또한 비밀번호는 **비밀번호 최소 길이** 설정에 따라 설정된 최소 길이를 유지해야 합니다.
4. **Invoke**를 클릭합니다.
5. 단일 테넌트 환경에서 **setRolesForUser** 메서드를 찾고 다음 매개 변수를 입력합니다.
   - **userName**. 통합 사용자의 이름입니다.
   - **roles.** SuperAdmin.

   Invoke를 클릭합니다.

6. 다중 테넌트 환경에서 **grantRolesToUserForAllTenants** 메서드를 찾고 다음 매개 변수를 입력하여 모든 테넌트와 연결되는 역할을 할당합니다.
   - **userName**. 통합 사용자의 이름입니다.
   - **roles.** SuperAdmin.

   호출을 클릭합니다.

   또는 특정 테넌트와 연결된 역할을 할당하려면 같은 userName 및 roles 매개 변수 값을 사용하여 **grantRolesToUserForTenants** 메서드를 호출합니다. **tenantNames** 매개 변수에 필수 테넌트를 입력합니다.
7. 추가 사용자를 만들거나 JMX 콘솔을 닫습니다.
8. UCMDB에 관리자로 로그온합니다.
9. **관리** 탭에서 **패키지 관리자**를 실행합니다.
10. **사용자 지정 패키지 만들기** 아이콘을 클릭합니다.
11. 새 패키지의 이름을 입력하고 **다음**을 클릭합니다.
12. 리소스 선택 탭에서 **설정** 아래에 있는 **사용자**를 클릭합니다.
13. JMX 콘솔을 사용하여 만든 사용자를 하나 이상 선택합니다.
14. **다음**을 클릭한 다음 **마침**을 클릭합니다. 패키지 관리자의 패키지 이름 목록에 새 패키지가 나타납니다.
15. API 응용 프로그램을 실행할 사용자에게 패키지를 배포합니다.

    자세한 내용은 *HP Universal CMDB 관리 안내서*에서 "패키지를 배포하는 방법" 섹션을 참조하십시오.

참고: 통합 사용자는 고객별입니다. 고객 간에 교차 사용할 수 있는 보다 강력한 통합 사용자를 만들려면 **systemUser**를 사용할 때 **isSuperIntegrationUser** 플래그를 **true**로 설정합니다. **systemUser** 메서드(**removeUser**, **resetPassword**, **UserAuthenticate** 등)를 사용합니다.

기본 시스템 사용자는 두 가지입니다. 설치 후 **resetPassword** 메서드를 사용하여 각 사용자의 비밀번호를 변경하는 것이 좋습니다.

- **sysadmin/sysadmin**
- **UISysadmin/UISysadmin**(이 사용자도 **SuperIntegrationUser**입니다.)

  resetPassword를 사용하여 UISysadmin 비밀번호를 변경하는 경우 다음을 수행해야 합니다.

  i. JMX 콘솔에서 **UCMDB-UI:name=UCMDB Integration** 서비스를 찾습니다.

  ii. 통합 사용자의 사용자 이름과 새 비밀번호로 **setCMDBSuperIntegrationUser**를 실행합니다.

UCMDB API 사용 사례

이 섹션에 나열된 사용 사례에서는 다음 두 가지 시스템을 사용하는 것으로 가정합니다.

- HP Universal CMDB 서버
- 구성 항목 저장소를 포함하는 타사 시스템

이 섹션에는 다음 항목이 포함됩니다.

CMDB 채우기

사용 사례:

- 타사 자산 관리가 자산 관리에서만 사용할 수 있는 정보로 CMDB를 업데이트합니다.
- 여러 타사 시스템에서 CMDB를 채워 변경 내용을 추적하고 영향 분석을 수행할 수 있는 중앙 CMDB를 만듭니다.
- 타사 시스템에서 UCMDB 쿼리 기능을 활용하기 위해 타사 비즈니스 논리에 따라 구성 항목 및 관계를 만듭니다.

CMDB 쿼리

사용 사례:

- 타사 시스템에서 SAP TQL의 결과를 검색하여 SAP 시스템을 나타내는 구성 항목 및 관계를 가져옵니다.
- 타사 시스템에서 지난 5시간 동안 추가되었거나 변경된 Oracle 서버 목록을 가져옵니다.
- 타사 시스템에서 호스트 이름에 **lab**이라는 하위 문자열이 포함된 서버 목록을 가져옵니다.
- 타사 시스템에서 주어진 CI의 인접 항목을 가져와서 이 주어진 CI와 관련된 요소를 찾습니다.

클래스 모델 쿼리

사용 사례:

- 타사 시스템을 통해 사용자가 CMDB에서 검색할 데이터 집합을 지정할 수 있습니다. 클래스 모델을 기반으로 사용자 인터페이스를 작성하면 사용자에게 가능한 속성을 표시하고 필수 데이터를 입력하라는 프롬프트를 표시할 수 있습니다. 그러면 사용자가 검색할 정보를 선택할 수 있습니다.
- 사용자가 UCMDB 사용자 인터페이스에 액세스할 수 없을 때 타사 시스템에서 클래스 모델을 탐색합니다.

변경 영향 분석

사용 사례:

- 타사 시스템에서 지정된 호스트에 대한 변경에 의해 영향을 받을 수 있는 비즈니스 서비스 목록을 출력합니다.

예

다음 코드 샘플을 참조하십시오.

- Create a Connection
- Create and Execute an Ad Hoc Query
- Create and Execute a View
- Add and Delete Data
- Execute an Impact Analysis
- Query the Class Model
- Query a History Sample

이러한 파일은 다음 디렉터리에 있습니다.

\\<UCMDB 루트 디렉터리>\hp\UCMDB\UCMDBServer\deploymdb-docs\docs\eng\APIs\JavaSDK_Samples

11장: HP Universal CMDB 웹 서비스 API

이 장의 내용:

규칙

이 장에서는 다음 규칙을 사용합니다.

- UCMDB는 Universal 구성 관리 데이터베이스 자체를 참조합니다. HP Universal CMDB는 해당 응용 프로그램을 참조합니다.
- UCMDB 요소 및 메서드 인수는 스키마에 지정된 경우에만 의미가 있습니다. 메서드에 대한 인수나 요소는 대문자로 표시되지 않습니다. 예를 들어 relation은 메서드에 전달된 관계 유형의 요소입니다.

요청 및 응답 구조에 관한 전체 문서는 HP UCMDB Web Service API Reference를 참조하십시오. 이러한 파일은 다음 폴더에 있습니다.

<UCMDB 루트 디렉터리>\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\CMDB_Schema\webframe.html

HP Universal CMDB 웹 서비스 API 개요

참고: 이 장은 온라인 문서 라이브러리에서 볼 수 있는 UCMDB 스키마 문서와 함께 사용하십시오.

HP Universal CMDB 웹 서비스 API는 응용 프로그램을 HP Universal CMDB(UCMDB)와 통합하는 데 사용됩니다. API는 다음을 수행할 수 있는 메서드를 제공합니다.

- CMDB에서 CI 및 관계 추가, 제거, 업데이트
- 클래스 모델에 대한 정보 검색
- 영향 분석 검색
- 구성 항목 및 관계에 대한 정보 검색
- 자격 증명 관리: 보기, 추가, 업데이트, 제거
- 작업 관리: 상태 보기, 활성화, 비활성화
- 프로브 범위 관리: 보기, 추가, 업데이트
- 트리거 관리: 트리거 CI 추가 또는 제거, 트리거 TQL 추가, 제거 또는 사용하지 않도록 설정
- 도메인 및 프로브에서 일반 데이터 보기

구성 항목 및 관계에 대한 정보를 검색하는 메서드는 일반적으로 TQL(토폴로지 쿼리 언어)을 사용합니다. 자세한 내용은 *HP Universal CMDB 모델링 안내서*에서 토폴로지 쿼리 언어를 참조하십시오.

HP Universal CMDB 웹 서비스 API의 사용자는 다음 사항을 잘 알고 있어야 합니다.

- SOAP 사양
- 개체 지향 프로그래밍 언어(예: C++, C# 또는 Java)
- HP Universal CMDB
- 데이터 흐름 관리

이 섹션에는 다음 항목이 포함됩니다.

- "API의 사용"(317페이지)
- "사용 권한"(318페이지)

API의 사용

UCMDB 웹 서비스 API는 여러 비즈니스 요구 사항을 만족하는 데 사용됩니다. 예:

- 사용 가능한 CI(구성 항목)에 대한 정보를 받기 위해 타사 시스템에서 클래스 모델을 쿼리할 수 있습니다.
- 타사 자산 관리 도구가 해당 도구에만 사용할 수 있는 정보로 CMDB를 업데이트할 수 있습니다. 이를 통해 해당 데이터를 HP 응용 프로그램에서 수집한 데이터와 통합합니다.
- 여러 타사 시스템에서 CMDB를 채워 변경 내용을 추적하고 영향 분석을 수행할 수 있는 중앙 CMDB를 만들 수 있습니다.
- 타사 시스템에서 해당 비즈니스 논리에 따라 엔터티 및 관계를 만든 다음 CMDB에 이 데이터를 기록하

여 CMDB 쿼리 기능을 활용할 수 있습니다.

- Release Control(CCM) 시스템과 같은 여타 시스템은 변경 내용 분석을 위해 영향 분석 메서드를 사용할 수 있습니다.

사용 권한

웹 서비스의 WSDL 파일에 액세스하려면 **http://localhost:8080/axis2/services/UcmdbService?wsdl**을 방문하십시오. WSDL 파일을 보려면 서버 관리자의 사용자 자격 증명을 제공해야 합니다.

참고: Axis2 관리 콘솔에는 액세스할 수 없습니다.

사용자가 로그인하려면 **레거시 API 실행** 일반 수행 권한이 있어야 합니다.

다음 표에는 각 웹 서비스 API 명령에 대해 필요한 추가 권한이 표시되어 있습니다.

웹 서비스 API 명령	필요한 권한
addCIsAndRelations deleteCIsAndRelations updateCIsAndRelations	일반 수행: **데이터 업데이트**
executeTopologyQueryByName(AdHoc) executeTopologyQueryByNameWithParameters(AdHoc) executeTopologyQueryWithParameters(AdHoc)	일반 수행: **정의별로 쿼리 실행** 각 쿼리에 대해: **보기** 권한
getTopologyQueryExistingResultByName getTopologyQueryResultCountByName releaseChunks pullTopologyMapChunks getCINeighbours getFilteredCIsByType getCIsById getCIsByType getRelationsById	일반 수행: **CI 보기** 각 쿼리에 대해: **보기** 권한
getQueryNameOfView	일반 수행: **CI 보기** 각 보기에 대해: **보기** 권한
getChangedCIs	일반 수행: **기록 내역 보기**, **CI 보기**
calculateImpact getImpactPath getImpactRulesByGroupName getImpactRulesByNamePrefix	일반 수행: **영향 분석 실행**

웹 서비스 API 명령	필요한 권한
getAllClassesHierarchy getClassAncestors getCmdbClassDefinition	없음

참고: UCMDB에서 루트 컨텍스트가 변경된 경우 다음 단계에 따라 웹 서비스 API에 대한 액세스를 사용하도록 설정합니다.

1. **\UCMDB\UCMDBServer\deploy\axis2\WEB-INF\web.xml** 구성 파일을 열고 다음 단락을 찾습니다.

   ```
   <servlet-class>
   org.apache.axis2.transport.http.AxisServlet
   </servlet-class>
   ```

   그 뒤에 다음 줄을 추가합니다.

   ```
   <init-param>
   <param-name>axis2.find.context</param-name>
   <param-value>false</param-value>
   </init-param>
   ```

2. **\UCMDB\UCMDBServer\deploy\axis2\WEB-INF\conf\axis2.xml** 구성 파일을 열고 다음 줄을 찾습니다.

   ```
   <parameter name="enableSwA" locked="false">false</parameter>
   ```

   그 뒤에 다음 줄을 추가합니다.

   ```
   <parameter name="contextRoot" locked="false">test1/setup1/axis2
   </parameter>
   ```

   여기서 **test1/setup1**은 루트 컨텍스트입니다.

   (루트 컨텍스트를 제거하려면 경로에 추가된 텍스트를 제거합니다.)

3. UCMDB 서버를 다시 시작합니다.

HP Universal CMDB 웹 서비스 호출

HP Universal CMDB 웹 서비스 API의 표준 SOAP 프로그래밍 기술을 사용하여 서버 쪽 메서드를 호출할 수 있습니다. 명령문을 구문 분석할 수 없거나 메서드를 호출하는 데 문제가 있는 경우 API 메서드는 SoapFault 예외를 발생시킵니다. SoapFault 예외가 발생하면 UCMDB에서 하나 이상의 오류 메시지, 오류 코드 및 예외 메시지 필드를 채웁니다. 오류가 없으면 호출 결과가 반환됩니다.

SOAP 프로그래머가 다음 위치에서 WSDL에 액세스할 수 있습니다.

http://<서버>[:port]/axis2/services/UcmdbService?wsdl

포트는 표준 설치가 아닐 경우에만 지정해야 합니다. 올바른 포트 번호는 시스템 관리자에게 문의하십시오.

서비스를 호출할 수 있는 URL은 다음과 같습니다.

http://<서버>[:port]/axis2/services/UcmdbService

CMDB에 연결하는 예는 "UCMDB 웹 서비스 API 사용 사례"(347페이지)를 참조하십시오.

CMDB 쿼리

" UCMDB 쿼리 메서드"(329페이지)에 설명된 API를 사용하여 CMDB을 쿼리합니다. 쿼리 및 반환되는 CMDB 요소는 항상 실제 UCMDB ID를 포함합니다. 쿼리 메서드를 사용하는 예는 Query Example를 참조하십시오.

이 섹션에는 다음 항목이 포함됩니다.

적시 응답 계산

모든 쿼리 메서드에 대해 UCMDB 서버는 요청을 받으면 쿼리 메서드가 요청한 값을 계산하고 최신 데이터를 기반으로 한 결과를 반환합니다. TQL 쿼리가 활성 상태이고 이전에 계산된 결과가 있는 경우에도 결과는 항상 요청을 받았을 때 계산됩니다. 그러므로 실행 중인 쿼리가 클라이언트 응용 프로그램에 반환된 결과와 동일한 쿼리가 사용자 인터페이스에 표시되는 결과가 다를 수 있습니다.

팁: 응용 프로그램에서 주어진 쿼리 결과를 두 번 이상 사용하는 경우 결과 데이터를 사용할 때마다 데이터가 크게 변경되지 않을 것 같으면 반복적으로 쿼리를 실행하는 대신 클라이언트 응용 프로그램에 데이터를 저장함으로써 성능을 향상시킬 수 있습니다.

대량 응답 처리

쿼리에 대한 응답에는 항상 쿼리 메서드에서 요청한 데이터의 구조가 포함됩니다. 실제 데이터가 전송되지 않더라도 마찬가지입니다. 데이터가 컬렉션 또는 맵인 여러 메서드의 경우 chunksKey 및 numberOfChunks로 구성된 ChunkInfo 구조도 응답에 포함됩니다. numberOfChunks 필드는 검색해야 할 데이터가 포함된 청크 수를 나타냅니다.

데이터의 최대 전송 크기는 시스템 관리자가 설정합니다. 쿼리로부터 반환되는 데이터가 최대 크기보다 큰 경우 첫 번째 응답의 데이터 구조에 의미 있는 정보가 포함되지 않고 numberOfChunks 필드의 값이 2 이상이 됩니다. 데이터가 최대 크기보다 크지 않은 경우에는 numberOfChunks 필드가 0이고 데이터가 첫 번째 응답에 전송됩니다. 그러므로 응답을 처리할 때는 가장 먼저 numberOfChunks 값을 확인하십시

오. 1보다 크면 전송에서 데이터를 무시하고 데이터의 청크를 요청합니다. 그렇지 않으면 응답에 데이터를 사용합니다.

청크 분할 데이터를 처리하는 방법에 대한 자세한 내용은 HP UCMDB 모듈 쿼리 메서드 항목의 "pullTopologyMapChunks"(339페이지) 및 "releaseChunks"(340페이지)의 스키마 문서를 참조하십시오.

반환할 속성 지정

CI 및 관계는 일반적으로 여러 개의 속성을 갖습니다. 이러한 항목의 컬렉션 또는 그래프를 반환하는 일부 메서드에는 쿼리와 일치하는 각 항목과 함께 반환할 속성 값을 지정하는 입력 매개 변수를 사용할 수 있습니다. CMDB는 빈 속성을 반환하지 않습니다. 그러므로 쿼리에 대한 응답에 포함되는 속성이 쿼리에서 요청한 속성보다 적을 수도 있습니다.

이 섹션에서는 반환할 속성을 지정하는 데 사용되는 설정 유형을 설명합니다.

속성은 다음 두 가지 방식으로 참조할 수 있습니다.

- 이름별
- 미리 정의된 속성 규칙의 이름 사용. CMDB에서 미리 정의된 속성 규칙을 사용하여 실제 속성 이름 목록을 만듭니다.

응용 프로그램이 이름별로 속성을 참조하면 PropertiesList 요소를 전달합니다.

> **팁:** 가능한 경우 규칙 기반 설정을 사용하기보다 PropertiesList를 사용하여 관심 있는 속성 이름을 지정하십시오. 미리 정의된 속성 규칙을 사용하면 대개 필요한 수보다 많은 속성이 반환되어 성능이 저하됩니다.

미리 정의된 속성에는 한정자 속성과 단순 속성이라는 두 가지 유형이 있습니다.

- **한정자 속성.** 이 속성은 클라이언트 응용 프로그램에서 QualifierProperties 요소(속성에 적용할 수 있는 한정자 목록)를 전달해야 하는 경우에 사용합니다. CMDB은 클라이언트 응용 프로그램에서 전달한 한정자 목록을 하나 이상의 한정자를 적용할 속성 목록으로 변환합니다. 이러한 속성 값은 CI 또는 Relation 요소와 함께 반환됩니다.
- **단순 속성.** 단순 규칙 기반 속성을 사용하려는 경우 클라이언트 응용 프로그램에서 SimplePredefinedProperty 또는 SimpleTypedPredefinedProperty 요소를 전달합니다. 이러한 요소에는 CMDB에서 반환할 속성 목록을 생성할 때 적용되는 규칙 이름이 포함되어 있습니다. SimplePredefinedProperty 또는 SimpleTypedPredefinedProperty 요소에 지정할 수 있는 규칙은 CONCRETE, DERIVED 및 NAMING입니다.

구체 속성

구체 속성은 지정된 CIT에 정의된 속성 집합입니다. 파생 클래스의 인스턴스에 대해서는 해당 파생 클래스에 의해 추가된 속성이 반환되지 않습니다.

메서드가 반환하는 인스턴스의 컬렉션은 메서드 호출에 지정된 CIT의 인스턴스와 해당 CIT에서 상속하는 CIT의 인스턴스로 구성될 수 있습니다. 파생 CIT는 지정된 CIT의 속성을 상속합니다. 뿐만 아니라 파생 CIT는 속성을 추가하여 상위 CIT를 확장합니다.

구체 속성의 예:

CIT T1에 P1 및 P2 속성이 있습니다. CIT T11은 T1에서 상속하여 P21 및 P22 속성으로 T1을 확장합니다.

T1 유형의 CI 컬렉션에는 T1 및 T11의 인스턴스가 포함됩니다. 이 컬렉션에 포함된 모든 인스턴스의 구체 속성은 P1 및 P2입니다.

파생 속성

파생 속성은 지정된 CIT에 대해 정의된 속성 집합이고, 각 파생 CIT의 경우 파생 CIT에 의해 추가된 속성입니다.

파생 속성의 예:

구체 속성의 예를 계속 사용하자면, T1 인스턴스의 파생 속성은 P1 및 P2입니다. T11 인스턴스의 파생 속성은 P1, P2, P21 및 P22입니다.

이름 지정 속성

이름 지정 속성은 display_label 및 data_name입니다.

기타 속성 사양 요소

- **PredefinedProperties**

  PredefinedProperties에는 기타 가능한 각 규칙에 대한 QualifierProperties 요소와 SimplePredefinedProperty 요소를 포함할 수 있습니다. PredefinedProperties 집합에는 몇몇 목록 유형을 포함하지 않아도 됩니다.

- **PredefinedTypedProperties**

  PredefinedTypedProperties는 서로 다른 속성 집합을 각 CIT에 적용하는 데 사용됩니다. PredefinedTypedProperties에는 기타 적용 가능한 각 규칙에 대한 QualifierProperties 요소와 SimpleTypedPredefinedProperty 요소를 포함할 수 있습니다. PredefinedTypedProperties는 각 CIT에 개별적으로 적용되기 때문에 파생 속성은 관련이 없습니다. PredefinedProperties 집합에는 몇몇 적용 가능한 목록 유형을 포함하지 않아도 됩니다.

- **CustomProperties**

  CustomProperties 에는 기본 PropertiesList 및 규칙 기반 속성 목록을 조합하여 포함할 수 있습니다. 속성 필터는 모든 목록에서 반환되는 모든 속성의 합집합입니다.

- **CustomTypedProperties**

  CustomTypedProperties에는 기본 PropertiesList 및 적용 가능한 규칙 기반 속성 목록을 조합하여 포함할 수 있습니다. 속성 필터는 모든 목록에서 반환되는 모든 속성의 합집합입니다.

- **TypedProperties**

TypedProperties는 서로 다른 속성 집합을 각 CIT에 전달하는 데 사용됩니다. TypedProperties는 모든 유형의 유형 이름과 속성 집합으로 구성된 쌍의 컬렉션입니다. 각 속성 집합은 해당 유형에만 적용됩니다.

CMDB 업데이트

업데이트 API를 사용하여 CMDB를 업데이트합니다. API 메서드에 대한 자세한 내용은 " UCMDB 업데이트 메서드"(341페이지)를 참조하십시오.

이 작업에는 다음 단계가 포함됩니다.

- "UCMDB 업데이트 매개 변수"(323페이지)
- "업데이트 메서드를 포함하는 ID 유형 사용"(323페이지)

UCMDB 업데이트 매개 변수

이 항목에서는 서비스의 업데이트 메서드에서만 사용되는 매개 변수를 설명합니다.

- **CIsAndRelationsUpdates**

  CIsAndRelationsUpdates 유형은 CIsForUpdate, relationsForUpdate, referencedRelations, referencedCIs로 구성됩니다. CIsAndRelationsUpdates 인스턴스는 세 요소를 모두 포함하지 않아도 됩니다.

  CIsForUpdate는 CI 컬렉션이고, relationsForUpdate는 Relations 컬렉션입니다. 컬렉션의 CI 및 relation 요소에는 props 요소가 있습니다. CI 또는 관계를 만들 때 CI 유형 정의에 필수 특성이나 키 특성이 있는 속성의 경우 값을 채워야 합니다. 이러한 컬렉션의 항목은 메서드를 통해 업데이트되거나 생성됩니다.

  referencedCIs 및 referencedRelations는 CMDB에 이미 정의된 CI의 컬렉션입니다. 컬렉션의 요소는 모든 키 속성과 연결된 임시 ID로 식별됩니다. 이러한 항목은 업데이트를 위해 CI 및 관계의 ID를 확인하는 데 사용되며, 메서드를 통해 업데이트되거나 생성되지 않습니다.

  이러한 컬렉션의 각 CI 및 relation 요소에는 속성 컬렉션이 있습니다. 새 항목은 이러한 컬렉션에서 속성 값으로 생성됩니다.

업데이트 메서드를 포함하는 ID 유형 사용

다음은 ID CIT 그리고 CI와 관계에 대한 설명입니다. ID가 실제 CMDB ID가 아닌 경우 유형 및 키 특성이 필요합니다.

- **구성 항목 삭제 또는 업데이트**

  항목을 삭제하거나 업데이트할 메서드를 호출할 때 클라이언트에서 임시 ID나 빈 ID를 사용할 수도 있습니다. 이 경우, CI를 식별하는 CI 유형 및 " 키 특성"을 설정해야 합니다.

- **관계 삭제 또는 업데이트**

  관계를 삭제하거나 업데이트할 때 관계 ID는 빈 ID, 임시 ID, 실제 ID 중 하나입니다.

  CI의 ID가 임시 ID인 경우 referencedCIs 컬렉션으로 CI를 전달해야 하며 해당 키 특성을 지정해야 합니다. 자세한 내용은 "CIsAndRelationsUpdates"(323페이지)에서 referencedCIs를 참조하십시오.

- **CMDB 에 새 구성 항목 삽입**

  빈 ID 또는 임시 ID를 사용하여 새 CI를 삽입할 수 있습니다. 그러나 ID가 빈 ID인 경우에는 clientID가 없기 때문에 서버에서 createIDsMap 구조의 실제 CMDB ID를 반환할 수 없습니다. 자세한 내용은 "addCIsAndRelations"(341페이지) 및 " UCMDB 쿼리 메서드"(329페이지)를 참조하십시오.

- **CMDB 에 새 관계 삽입**

  관계 ID는 임시 ID 또는 빈 ID 중 하나입니다. 그러나 관계는 새 관계이지만 관계의 양끝에 있는 구성 항목이 CMDB에 이미 정의된 경우에는 기존의 CI를 실제 CMDB ID로 식별하거나 referencedCIs 컬렉션에서 지정해야 합니다.

UCMDB 클래스 모델 쿼리

클래스 모델 메서드는 CIT 및 관계에 대한 정보를 반환합니다. 클래스 모델은 CI 유형 관리자를 사용하여 구성됩니다. 자세한 내용은 *HP Universal CMDB 모델링 안내서*에서 CI 유형 관리자를 참조하십시오.

이 섹션에서는 CIT 및 관계에 대한 정보를 반환하는 다음 메서드에 대한 정보를 제공합니다.

getClassAncestors

getClassAncestors 메서드는 주어진 CIT와 이 CIT의 루트 간 경로(루트 포함)를 검색합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
className	유형 이름입니다. 자세한 내용은 "유형 이름"(327페이지)을 참조하십시오.

출력

매개 변수	설명
classHierarchy	클래스 이름과 상위 클래스 이름의 쌍으로 이루어진 컬렉션입니다.
comments	내부 전용입니다.

getAllClassesHierarchy

getAllClassesHierarchy 메서드는 전체 클래스 모델 트리를 검색합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.

출력

매개 변수	설명
classesHierarchy	클래스 이름과 상위 클래스 이름의 쌍으로 이루어진 컬렉션입니다.
comments	내부 전용입니다.

getCmdbClassDefinition

getCmdbClassDefinition 메서드는 지정된 클래스에 대한 정보를 검색합니다.

getCmdbClassDefinition을 사용하여 키 특성을 검색하는 경우 상위 클래스도 기본 클래스까지 쿼리해야 합니다. getCmdbClassDefinition은 className으로 지정한 클래스 정의에 ID_ATTRIBUTE가 설정된 특성만 키 특성으로 식별합니다. 상속된 키 특성은 지정된 클래스의 키 특성으로 인식되지 않습니다. 그러므로 지정된 클래스의 모든 키 특성 목록은 해당 클래스 및 모든 상위 클래스(루트 포함)에 있는 모든 키의 합집합입니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
className	유형 이름입니다. 자세한 내용은 "UCMDB 일반 매개 변수 "(326페이지)을 참조하십시오.

출력

매개 변수	설명
cmdbClass	name, classType, displayLabel, description, parentName, 한정자 및 특성으로 구성된 클래스 정의입니다.
comments	내부 전용입니다.

영향 분석에 대한 쿼리

영향 분석 메서드의 Identifier는 서비스의 응답 데이터를 가리킵니다. 현재 응답에 대해 고유하고 10분 동안 사용하지 않으면 서버의 메모리 캐시에서 제거됩니다.

영향 분석 메서드를 사용하는 예는 Impact Analysis Example를 참조하십시오.

UCMDB 일반 매개 변수

이 섹션에서는 서비스의 메서드에 가장 많이 사용되는 매개 변수를 설명합니다.

이 섹션에는 다음 항목이 포함됩니다.

CmdbContext

모든 UCMDB 웹 서비스 API 서비스를 호출하려면 CmdbContext 인수가 필요합니다. CmdbContext는 서비스를 호출하는 응용 프로그램을 식별하는 callerApplication 문자열입니다. CmdbContext는 로깅 및 문제 해결에 사용됩니다.

ID

모든 CI 및 관계에는 ID 필드가 있습니다. 이 필드는 대/소문자를 구분하는 ID 문자열과 ID가 임시 ID인지 여부를 나타내는 temp 플래그(선택 사항)로 구성됩니다.

키 특성

일부 컨텍스트에서 CI 또는 Relation을 식별하기 위해 CMDB ID 대신 키 특성을 사용할 수 있습니다. 키 특성은 클래스 정의에 ID_ATTRIBUTE가 설정된 특성입니다.

사용자 인터페이스의 구성 항목 유형 특성 목록에서 키 특성 옆에는 키 아이콘이 있습니다. 자세한 내용은 *HP Universal CMDB 모델링 안내서*에서 특성 추가/편집 대화 상자를 참조하십시오. API 클라이언트 응용 프로그램에서 키 특성을 식별하는 방법에 대한 자세한 내용은 "getCmdbClassDefinition"(325페이지)을 참조하십시오.

ID 유형

ID 요소에는 실제 ID 또는 임시 ID가 포함될 수 있습니다.

실제 ID는 데이터베이스의 항목을 식별하는 문자열로, CMDB에서 할당합니다. 임시 ID는 현재 요청에서 고유한 임의의 문자열일 수 있습니다.

임시 ID는 클라이언트에서 할당할 수 있으며 클라이언트에서 저장한 대로 CI의 ID를 나타내는 경우가 많습니다. 반드시 CMDB에 이미 만들어져 있는 엔터티를 나타낼 필요는 없습니다. 클라이언트에서 임시 ID를 전달한 경우 CMDB에서 CI 키 속성을 사용하여 기존 데이터 구성 항목을 식별할 수 있으면 해당 CI는 실제 ID를 사용하여 식별된 것처럼 컨텍스트에 적합한 CI로 사용됩니다.

CIProperties

CIProperties 요소는 여러 개의 컬렉션으로 구성되며, 각 컬렉션에는 컬렉션 이름이 나타내는 유형의 속성을 지정하는 이름-값 요소의 시퀀스가 포함됩니다. 필수 컬렉션은 없으므로 CIProperties 요소에는 어떤 조합의 컬렉션이나 포함할 수 있습니다.

CIProperties는 CI 및 관계 요소에 사용됩니다. 자세한 내용은 "CI(구성 항목)"(327페이지) 및 "관계"(328페이지)를 참조하십시오.

속성 컬렉션은 다음과 같습니다.

- dateProps - DateProp 요소의 컬렉션
- doubleProps - DoubleProp 요소의 컬렉션
- floatProps - FloatProp 요소의 컬렉션
- intListProps - intListProp 요소의 컬렉션
- intProps - IntProp 요소의 컬렉션
- strProps - StrProp 요소의 컬렉션
- strListProps - StrListProp 요소의 컬렉션
- longProps - LongProp 요소의 컬렉션
- bytesProps - BytesProp 요소의 컬렉션
- xmlProps - XmlProp 요소의 컬렉션

유형 이름

유형 이름은 구성 항목 유형 또는 관계 유형의 클래스 이름입니다. 유형 이름은 코드에서 클래스를 참조하는 데 사용됩니다. 표시 이름과 혼동하지 마십시오. 표시 이름은 클래스를 설명하는 사용자 인터페이스에 표시되지만 코드에서는 의미가 없습니다.

CI(구성 항목)

CI 요소는 ID, type, 및 props 컬렉션으로 구성됩니다.

" UCMDB 업데이트 메서드"를 사용하여 CI를 업데이트하면 실제 CMDB ID나 클라이언트에서 할당한 임시 ID를 ID 요소에 포함할 수 있습니다. 임시 ID를 사용하는 경우 temp 플래그를 true로 설정합니다. 항목을

삭제하면 빈 ID가 될 수 있습니다. " UCMDB 쿼리 메서드"가 실제 ID를 입력 매개 변수로 사용하여 쿼리 결과에 실제 ID를 반환합니다.

유형은 CI 유형 관리자에 정의된 유형 이름일 수 있습니다. 자세한 내용은 HP Universal CMDB 모델링 안내서에서 CI 유형 관리자를 참조하십시오.

props 요소는 CIProperties 컬렉션입니다. 자세한 내용은 "UCMDB 일반 매개 변수 "(326페이지)를 참조하십시오.

관계

관계는 두 구성 항목을 연결하는 엔터티입니다. 관계 요소는 ID, type, 연결되는 두 항목의 식별자(end1ID 및 end2ID) 그리고 props 컬렉션으로 구성됩니다.

" UCMDB 업데이트 메서드"를 사용하여 Relation을 업데이트하면 Relation의 ID 값이 실제 CMDB ID나 임시 ID가 될 수 있습니다. 항목을 삭제하면 빈 ID가 될 수 있습니다. " UCMDB 쿼리 메서드"가 실제 ID를 입력 매개 변수로 사용하여 쿼리 결과에 실제 ID를 반환합니다.

관계 유형은 UCMDB 클래스의 Type Name입니다. 이 클래스에서 관계가 인스턴스화됩니다. 유형은 CMDB에 정의된 관계 유형 중 하나일 수 있습니다. 클래스 또는 유형에 대한 자세한 내용은 " UCMDB 클래스 모델 쿼리 "(324페이지)를 참조하십시오.

자세한 내용은 *HP Universal CMDB 모델링 안내서*에서 CI 유형 관리자를 참조하십시오.

관계의 양끝 ID는 현재 관계의 ID를 만드는 데 사용되기 때문에 빈 ID가 아니어야 합니다. 그러나 양끝 ID 모두에 클라이언트에서 할당한 임시 ID를 사용할 수는 있습니다.

props 요소는 CIProperties 컬렉션입니다. 자세한 내용은 "CIProperties"(327페이지)를 참조하십시오.

UCMDB 출력 매개 변수

이 섹션에서는 서비스의 메서드에 가장 많이 사용되는 출력 매개 변수를 설명합니다. 자세한 내용은 online schema documentation를 참조하십시오.

이 섹션에는 다음 항목이 포함됩니다.

CI

CI는 CI 요소의 컬렉션입니다.

ShallowRelation

ShallowRelation은 두 개의 구성 항목을 연결하는 엔터티로, ID, 유형, 연결되는 두 항목의 식별자(end1ID 및 end2ID)로 구성됩니다. 관계 유형은 CMDB 클래스의 Type Name입니다. 이 클래스에서 관계가 인스턴스화됩니다. 유형은 CMDB에 정의된 관계 유형 중 하나일 수 있습니다.

Topology

Topology는 CI 요소 및 관계의 그래프입니다. Topology는 CIs 컬렉션 및 하나 이상의 Relation 요소가 포함된 Relations 컬렉션으로 구성됩니다.

CINode

CINode는 CI 컬렉션과 label로 구성됩니다. CINode의 label은 쿼리에 사용되는 TQL의 노드에 정의된 레이블입니다.

RelationNode

RelationNode는 Relation 컬렉션 집합과 label로 구성됩니다. RelationNode의 label은 쿼리에 사용되는 TQL의 노드에 정의된 레이블입니다.

TopologyMap

TopologyMap은 TQL 쿼리와 일치하는 쿼리 계산의 출력입니다. TopologyMap의 labels는 쿼리에 사용되는 TQL에 정의된 노드 레이블입니다.

TopologyMap의 데이터는 다음 형식으로 반환됩니다.

- CINodes. 하나 이상의 CINode입니다("CINode"(329페이지) 참조).
- relationNodes. 하나 이상의 RelationNode입니다("RelationNode"(329페이지) 참조).

이러한 두 구조의 labels에 따라 구성 항목 및 관계 목록의 순서가 지정됩니다.

ChunkInfo

쿼리가 대량의 데이터를 반환하면 서버는 데이터를 청크라는 세그먼트로 분할하여 저장합니다. 클라이언트에서 청크로 분할된 데이터를 검색하는 데 사용하는 정보는 쿼리에서 반환한 ChunkInfo 구조에 있습니다. ChunkInfo는 검색되어야 하는 numberOfChunks와 chunksKey로 구성됩니다. chunksKey는 이 특정 쿼리를 호출하는 데 필요한 서버 데이터의 고유한 식별자입니다.

자세한 내용은 "대량 응답 처리"(320페이지)의 스키마 문서를 참조하십시오.

UCMDB 쿼리 메서드

이 섹션에서는 다음 메서드에 대한 정보를 제공합니다.

- "executeTopologyQueryByNameWithParameters"(330페이지)
- "executeTopologyQueryWithParameters "(331페이지)
- "getChangedCIs"(332페이지)

executeTopologyQueryByNameWithParameters

executeTopologyQueryByNameWithParameters 메서드는 매개 변수화된 지정된 쿼리와 일치하는 topologyMap 요소를 검색합니다.

쿼리 매개 변수의 값은 parameterizedNodes 인수로 전달됩니다. 지정된 TQL에는 각 CINode 및 각 relationNode에 대해 정의된 고유한 레이블이 있어야 합니다. 그렇지 않을 경우 메서드 호출에 실패합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
queryName	CMDB에서 맵을 가져오는 데 사용할 매개 변수화된 TQL의 이름입니다.
parameterizedNodeList	쿼리 결과에 포함되기 위해 각 노드가 충족해야 하는 조건입니다.
queryTypedProperties	특정 구성 항목 유형의 항목에 대해 검색할 속성 집합 컬렉션입니다.

출력

매개 변수	설명
topologyMap	자세한 내용은 "TopologyMap"(329페이지)를 참조하십시오.
chunkInfo	자세한 내용은 "ChunkInfo"(329페이지) 및 "대량 응답 처리"(320페이지)를 참조하십시오.

executeTopologyQueryWithParameters

executeTopologyQueryWithParameters 메서드는 매개 변수화된 쿼리와 일치하는 topologyMap 요소를 검색합니다.

쿼리는 queryXML 인수로 전달됩니다. 쿼리 매개 변수의 값은 parameterizedNodeList 인수로 전달됩니다. TQL에는 각 CINode 및 각 relationNode에 대해 정의된 고유한 레이블이 있어야 합니다.

Ad hoc 쿼리를 전달하기 위해서는 CMDB에 정의된 쿼리에 액세스하지 않고 executeTopologyQueryWithParameters 메서드를 사용합니다. 쿼리를 정의하는 데 필요한 UCMDB 사용자 인터페이스에 대한 액세스 권한이 없는 경우나 데이터베이스에 쿼리를 저장하지 않으려는 경우 이 메서드를 사용할 수 있습니다.

내보낸 TQL을 이 메서드의 입력으로 사용하려면 다음을 수행합니다.

1. 웹 브라우저를 시작하고 다음 주소를 입력합니다.
   http://localhost:8080/jmx-console

   사용자 이름과 비밀번호를 입력하여 로그인해야 할 수도 있습니다.
2. **UCMDB:service=TQL Services**를 클릭합니다.
3. **exportTql** 작업을 찾습니다.
   - **customerId** 매개 변수 상자에 **1**(기본값)을 입력합니다.
   - **patternName** 매개 변수 상자에 유효한 TQL 이름을 입력합니다.
4. **Invoke**를 클릭합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
queryXML	리소스 태그 없이 TQL을 나타내는 XML 문자열입니다.
parameterizedNodeList	쿼리 결과에 포함되기 위해 각 노드가 충족해야 하는 조건입니다.

출력

매개 변수	설명
topologyMap	자세한 내용은 "TopologyMap"(329페이지)를 참조하십시오.
chunkInfo	자세한 내용은 "ChunkInfo"(329페이지) 및 "대량 응답 처리"(320페이지)를 참조하십시오.

getChangedCIs

getChangedCIs 메서드는 지정된 CI와 관련된 모든 CI에 대한 변경 데이터를 반환합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
ids	관련 CI에서 변경 내용을 확인한 루트 CI의 ID 목록입니다. 이 컬렉션에서는 실제 CMDB ID만 유효합니다.
fromDate	CI가 변경되었는지 확인할 기간의 시작 날짜입니다.
toDate	CI가 변경되었는지 확인할 기간의 끝 날짜입니다.

출력

매개 변수	설명
getChangedCIsResponseList	0개 이상의 ChangedDataInfo 요소 컬렉션입니다.

getCINeighbours

getCINeighbours 메서드는 지정된 CI의 바로 인접한 CI를 반환합니다.

예를 들어 쿼리가 CI A에 바로 인접해 있고 CI A에 CI C를 사용하는 CI B가 포함된 경우, CI B는 반환되지만 CI C는 반환되지 않습니다. 즉, 지정된 유형의 인접 CI만 반환됩니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
ID	인접한 CI를 검색할 CI의 ID입니다. 이 ID는 실제 CMDB 또는 글로벌 ID여야 합니다.
neighbourType	검색할 인접 CI의 CIT 이름입니다. 지정된 유형 및 해당 유형에서 파생된 유형의 인접 유형이 반환됩니다. 자세한 내용은 "유형 이름"(327페이지)을 참조하십시오.
CIProperties	각 구성 항목에 대해 반환될 데이터로서, 사용자 인터페이스의 쿼리 레이아웃이라고 합니다. 자세한 내용은 "TypedProperties"(322페이지)를 참조하십시오.

매개 변수	설명
relationProperties	각 관계에 대해 반환될 데이터로서, 사용자 인터페이스의 쿼리 레이아웃이라고 합니다. 자세한 내용은 "TypedProperties"(322페이지)를 참조하십시오.

출력

매개 변수	설명
topology	자세한 내용은 "Topology"(329페이지)를 참조하십시오.
comments	내부 전용입니다.

getClsByID

getClsByID 메서드는 구성 항목의 CMDB 또는 글로벌 ID로 구성 항목을 검색합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
ClsTypedProperties	유형이 지정된 속성 컬렉션입니다. 자세한 내용은 "기타 속성 사양 요소"(322페이지)를 참조하십시오.
IDs	이 컬렉션에서는 실제 CMDB 또는 글로벌 ID만 유효합니다.

출력

매개 변수	설명
CI	CI 요소의 컬렉션입니다.
chunkInfo	자세한 내용은 "ChunkInfo"(329페이지) 및 "대량 응답 처리"(320페이지)를 참조하십시오.

getClsByType

getClsByType 메서드는 지정된 유형 및 지정된 유형에서 상속하는 모든 유형의 구성 항목 컬렉션을 반환합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
type	클래스 이름입니다. 자세한 내용은 "유형 이름"(327페이지)을 참조하십시오.
properties	각 구성 항목에 대해 반환할 데이터입니다. 자세한 내용은 "CustomProperties"(322페이지)를 참조하십시오.

출력

매개 변수	설명
CI	CI 요소의 컬렉션입니다.
chunkInfo	자세한 내용은 "ChunkInfo"(329페이지) 및 "대량 응답 처리"(320페이지)를 참조하십시오.

getFilteredCIsByType

getFilteredCIsByType 메서드는 이 메서드에 사용된 조건을 충족하는 지정된 유형의 CI를 검색합니다. 조건은 다음 항목으로 구성됩니다.

- 속성 이름을 포함하는 이름 필드
- 비교 연산자를 포함하는 연산자 필드
- 값 또는 값 목록을 포함하는 값 필드(선택 사항)

이러한 항목은 함께 부울 식을 만듭니다.
<*item*>.property.value [operator] <*condition*>.value

예를 들어 조건 이름이 root_actualdeletionperiod이면 조건 값은 40이고 연산자는 ==이므로, 부울 문은 다음과 같습니다.

<item>.*root_actualdeletionperiod*.value = = 40

이 쿼리는 다른 조건이 없다고 간주하고 root_actualdeletionperiod가 40인 모든 항목을 반환합니다.

conditionsLogicalOperator 인수가 AND인 경우에는 쿼리가 conditions 컬렉션에서 모든 조건을 충족하는 항목을 반환합니다. conditionsLogicalOperator 인수가 OR인 경우에는 쿼리가 conditions 컬렉션에서 조건 중 하나 이상을 충족하는 항목을 반환합니다.

다음 표에는 비교 연산자가 나열되어 있습니다.

연산자	조건/설명 유형
ChangedDuring	날짜 범위 확인입니다. 조건 값은 시간 단위로 지정됩니다. 날짜 속성 값이 메서드를 호

연산자	조건/설명 유형
	출한 시간에서 조건 값을 더하거나 뺀 범위에 있는 경우 조건이 충족됩니다. 예를 들어 조건 값이 24인 경우 날짜 속성 값이 어제의 현재 시간과 내일의 현재 시간 사이에 있으면 조건이 충족됩니다. **참고:** ChangedDuring 이름은 이전 버전과의 호환을 위해 그대로 유지됩니다. 이전 버전에서는 연산자가 시간 속성 만들기 및 수정에만 사용되었습니다.
==	문자열 및 숫자
EqualIgnoreCase	문자열
Greater	숫자
GreaterEqual	숫자
In	문자열, 숫자 및 목록 조건 값은 목록입니다. 속성 값이 목록의 값 중 하나인 경우 조건이 충족됩니다.
InList	목록 조건 값 및 속성 값이 목록입니다. 조건 목록의 모든 값이 항목의 속성 목록에도 나타나면 조건이 충족됩니다. 속성 값이 조건에 지정된 수보다 많을 수 있으며, 이 경우 조건 충족 여부에는 영향을 주지 않습니다.
IsNull	문자열, 숫자 및 목록 항목의 속성에 값이 없습니다. IsNull 연산자를 사용하면 조건 값이 무시되고 어떤 경우에는 nil이 될 수 있습니다.
Less	숫자
LessEqual	숫자
Like	문자열 조건 값이 속성 값의 하위 문자열 값입니다. 조건 값은 백분율 기호(%)와 함께 괄호 안에 포함되어야 합니다. 예를 들어 %Bi%의 경우 Bismark 및 Bay of Biscay는 일치하는 조건이지만 biscuit은 일치하는 조건이 아닙니다.
LikeIgnoreCase	문자열 LikeIgnoreCase 연산자는 Like 연산자를 사용할 때와 마찬가지로 사용합니다. 그러나 일치하는 문자열을 찾을 때 대/소문자는 구분하지 않습니다. 따라서 %Bi%는 biscuit과 일치합니다.

연산자	조건/설명 유형
NotEqual	문자열 및 숫자
UnchangedDuring	날짜 범위 확인입니다. 조건 값은 시간 단위로 지정됩니다. 날짜 속성 값이 메서드를 호출한 시간에서 조건 값을 더하거나 뺀 범위에 있는 경우 조건이 충족되지 않습니다. 해당 범위 밖에 있는 경우에는 조건이 충족됩니다. 예를 들어 조건 값이 24인 경우 날짜 속성 값이 어제의 현재 시간 이전이거나 내일의 현재 시간 이후이면 조건이 충족됩니다. **참고:** UnchangedDuring 이름은 이전 버전과의 호환을 위해 그대로 유지됩니다. 이전 버전에서는 연산자가 시간 속성 만들기 및 수정에만 사용되었습니다.

조건을 설정하는 작업의 예:

```
FloatCondition fc = new FloatCondition();
FloatProp fp = new FloatProp();
fp.setName("attr_name");
fp.setValue(11f);
fc.setCondition(fp);
fc.setFloatOperator(FloatCondition.FloatOperator.EQUAL);
```

상속한 속성을 쿼리하는 작업의 예:

대상 CI는 name과 size라는 두 개의 특성이 있는 sample입니다. sampleII는 level과 grade라는 두 개의 특성이 있는 CI를 확장합니다. 이 예제에서는 sample에서 상속된 sampleII의 속성을 이름별로 지정하여 쿼리하도록 설정합니다.

```
GetFilteredCIsByType request = new GetFilteredCIsByType()
request.setCmdbContext(cmdbContext)
request.setType("sampleII");
CustomProperties customProperties = new CustomProperties();
PropertiesList propertiesList = new PropertiesList();
propertiesList.setPropertyNames(Arrays.asList("name","size"));
customProperties.setPropertiesList(propertiesList);
request.setProperties(customProperties);
```

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.

매개 변수	설명
type	클래스 이름입니다. 자세한 내용은 "유형 이름"(327페이지)을 참조하십시오. 유형은 CI 유형 관리자를 사용하여 정의한 유형 중 하나일 수 있습니다. 자세한 내용은 *HP Universal CMDB 모델링 안내서*에서 CI 유형 관리자를 참조하십시오.
properties	각 CI에 대해 반환될 데이터로서, 사용자 인터페이스의 쿼리 레이아웃이라고 합니다. 자세한 내용은 "CustomProperties"(322페이지)를 참조하십시오.
conditions	이름-값 쌍의 컬렉션 및 서로를 연결하는 연산자입니다. 예: host_hostname like QA.
conditionsLogicalOperator	• **AND.** 모든 조건을 충족해야 합니다. • **OR**. 조건 중 하나 이상을 충족해야 합니다.

출력

매개 변수	설명
CI	CI 요소 컬렉션입니다.
chunkInfo	자세한 내용은 "ChunkInfo"(329페이지) 및 "대량 응답 처리"(320페이지)를 참조하십시오.

getQueryNameOfView

getQueryNameOfView 메서드는 지정된 보기의 기반이 되는 TQL의 이름을 검색합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
viewName	CMDB에 있는 클래스 모델의 하위 집합인 보기의 이름입니다.

출력

매개 변수	설명
queryName	CMDB에서 보기의 기반이 되는 TQL의 이름입니다.

getTopologyQueryExistingResultByName

getTopologyQueryExistingResultByName 메서드는 지정된 TQL의 최근 실행 결과를 검색합니다. 이 메서드를 호출해도 TQL은 실행되지 않습니다. 이전 실행에서 결과가 없었다면 아무것도 반환되지 않습니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
queryName	TQL의 이름입니다.
queryTypedProperties	특정 구성 항목 유형의 항목에 대해 검색할 속성 집합 컬렉션입니다.

출력

매개 변수	설명
topologyMap	자세한 내용은 "TopologyMap"(329페이지)을 참조하십시오.
chunkInfo	자세한 내용은 "ChunkInfo"(329페이지) 및 "대량 응답 처리"(320페이지)를 참조하십시오.

getTopologyQueryResultCountByName

getTopologyQueryResultCountByName 메서드는 지정된 쿼리와 일치하는 각 노드의 인스턴스 수를 검색합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
queryName	TQL의 이름입니다.
countInvisible	true인 경우 쿼리에서 숨김으로 정의된 CI가 출력에 포함됩니다.

출력

매개 변수	설명
getTopologyQueryResultCountByNameResponse	쿼리와 일치하는 인스턴스 개수입니다.

pullTopologyMapChunks

pullTopologyMapChunks 메서드는 메서드에 대한 응답이 포함된 청크 중 하나를 검색합니다.

각 청크에는 응답의 일부인 topologyMap 요소가 포함됩니다. 첫 번째 청크 번호가 1로 지정되므로 검색 루프 카운터가 1부터 <*응답 개체*>.getChunkInfo().getNumberOfChunks()까지 반복됩니다.

자세한 내용은 "ChunkInfo"(329페이지) 및 "CMDB 쿼리"(320페이지)를 참조하십시오.

클라이언트 응용 프로그램에서 부분 맵을 처리할 수 있어야 합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
ChunkRequest	검색할 청크 수 및 쿼리 메서드에 의해 반환되는 ChunkInfo입니다.
queryTypedProperties	특정 CI 유형의 항목에 대해 검색할 속성 집합 컬렉션입니다.

출력

매개 변수	설명
topologyMap	자세한 내용은 "TopologyMap"(329페이지)를 참조하십시오.
comments	내부 전용입니다.

청크 처리의 예:

```
GetCIsByType request =
    new GetCIsByType(cmdbContext, typeName, customProperties);
GetCIsByTypeResponse response =
    ucmdbService.getCIsByType(request);
ChunkRequest chunkRequest = new ChunkRequest();
chunkRequest.setChunkInfo(response.getChunkInfo());
for(int j=1; j<=response.getChunkInfo().getNumberOfChunks(); j++){
   chunkRequest.setChunkNumber(j);
```

```
   PullTopologyMapChunks req =new PullTopologyMapChunks(cmdbContext,chunkRequest);
   PullTopologyMapChunksResponse res =
      ucmdbService.pullTopologyMapChunks(req);
      for(int m=0 ;
         m < res.getTopologyMap().getCINodes().sizeCINodeList() ;
         m++) {
         CIs cis =
         res.getTopologyMap().getCINodes().getCINode(m).getCIs();
         for(int i=0 ; i < cis.sizeCIList() ; i++) {
            // your code to process the CIs
         }
      }
}
```

```
GetCIsByType request =
    new GetCIsByType(cmdbContext, typeName, customProperties);
GetCIsByTypeResponse response =
    ucmdbService.getCIsByType(request);
ChunkRequest chunkRequest = new ChunkRequest();
chunkRequest.setChunkInfo(response.getChunkInfo());
for(int j=1 ; j <= response.getChunkInfo().getNumberOfChunks() ; j++) {
   chunkRequest.setChunkNumber(j);
   PullTopologyMapChunks req = new PullTopologyMapChunks(cmdbContext, chunkRequest);
   PullTopologyMapChunksResponse res =
      ucmdbService.pullTopologyMapChunks(req);
      for(int m=0 ;
         m < res.getTopologyMap().getCINodes().getCINodes().size();
         m++) {
         CIs cis =
         res.getTopologyMap().getCINodes().getCINodes().get(m).getCIs();
         for(int i=0 ; i < cis.getCIs().size(); i++) {
            // your code to process the CIs
         }
      }
}
```

releaseChunks

releaseChunks 메서드는 쿼리에서 검색한 데이터가 포함된 청크의 메모리를 비웁니다.

팁: 10분 후에 서버에서 데이터를 제거합니다. 데이터를 읽는 즉시 제거하기 위해 이 메서드를 호출하면 서버 리소스가 확보됩니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
chunksKey	서버에서 청크로 분할된 데이터의 식별자입니다. 키는 ChunkInfo의 요소입니다.

UCMDB 업데이트 메서드

이 섹션에서는 다음 메서드에 대한 정보를 제공합니다.

- "addCIsAndRelations"(341페이지)
- "addCustomer"(342페이지)
- "deleteCIsAndRelations"(342페이지)
- "removeCustomer"(343페이지)
- "updateCIsAndRelations"(343페이지)

addCIsAndRelations

addCIsAndRelations 메서드는 CI 및 관계를 추가하거나 업데이트합니다.

CMDB에 CI 또는 관계가 없는 경우 CI 또는 관계가 추가되고 CIsAndRelationsUpdates 인수의 컨텐츠에 따라 해당 속성이 설정됩니다.

CMDB에 CI 또는 관계가 있는 경우 updateExisting이 **true**이면 CI 또는 관계가 새 데이터로 업데이트됩니다.

updateExisting이 **false**이면 CIsAndRelationsUpdates가 기존 구성 항목이나 관계를 참조할 수 없습니다. updateExisting이 false인 경우 기존 항목을 참조하려고 하면 예외가 발생합니다.

updateExisting이 **true**이면 ignoreValidation의 값에 관계없이 CI의 유효성이 검사되지 않고 추가 또는 업데이트 작업이 수행됩니다.

updateExisiting이 **false**이고 ignoreValidation이 **true**이면 CI의 유효성이 검사되지 않고 추가 작업이 수행됩니다.

updateExisiting이 **false**이고 ignoreValidation이 **false**이면 추가 작업 전에 CI의 유효성이 검사됩니다.

관계의 유효성은 검사되지 않습니다.

CreatedIDsMap은 클라이언트의 임시 ID를 그에 해당하는 실제 CMDB ID와 연결하는 ClientIDToCmdbID 유형의 맵 또는 사전입니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
updateExisting	CMDB에 이미 있는 항목을 업데이트하려면 **true**로 설정합니다. 항목이 이미 있는 경우 예외를 발생시키려면 **false**로 설정합니다.
ClsAndRelationsUpdates	업데이트하거나 만들 항목입니다. 자세한 내용은 "ClsAndRelationsUpdates"(323페이지)를 참조하십시오.
ignoreValidation	true이면 CMDB 업데이트 전에 확인이 수행되지 않습니다.
dataStore	변경한 사람 정보입니다.

출력

매개 변수	설명
createdIDsMapList	CMDB ID에 매핑된 클라이언트 ID 목록입니다. 자세한 내용은 위의 설명을 참조하십시오.
comments	내부 전용입니다.

addCustomer

addCustomer 메서드는 고객을 추가합니다.

입력

매개 변수	설명
CustomerID	고객의 숫자 ID입니다.

deleteClsAndRelations

deleteClsAndRelations 메서드는 CMDB에서 지정된 구성 항목 및 관계를 제거합니다.

CI가 삭제될 때 CI가 하나 이상의 관계 항목의 한 끝이면 해당 관계 항목도 삭제됩니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
ClsAndRelationsUpdates	삭제할 항목입니다. 자세한 내용은 "ClsAndRelationsUpdates"(323페이지)를 참조하십시오.
dataStore	변경한 사람 정보입니다.

removeCustomer

removeCustomer 메서드는 고객 기록을 삭제합니다.

입력

매개 변수	설명
CustomerID	고객의 숫자 ID입니다.

updateClsAndRelations

updateClsAndRelations 메서드는 지정된 CI 및 관계를 업데이트합니다.

업데이트 시 ClsAndRelationsUpdates 인수의 속성 값을 사용합니다. CMDB에 CI 또는 관계가 없는 경우 예외가 발생합니다.

CreatedIDsMap은 클라이언트의 임시 ID를 그에 해당하는 실제 CMDB ID와 연결하는 ClientIDToCmdbID 유형의 맵 또는 사전입니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
ClsAndRelationsUpdates	업데이트할 항목입니다. 자세한 내용은 "ClsAndRelationsUpdates"(323페이지)를 참조하십시오.
ignoreValidation	true이면 CMDB 업데이트 전에 확인이 수행되지 않습니다.
dataStore	변경한 사람 정보입니다.

출력

매개 변수	설명
createdIDsMapList	CMDB ID에 매핑된 클라이언트 ID 목록입니다. 자세한 내용은 "addCIsAndRelations"(341페이지)를 참조하십시오.

UCMDB 영향 분석 메서드

이 섹션에서는 다음 메서드에 대한 정보를 제공합니다.

- "calculateImpact"(344페이지)
- "getImpactPath"(345페이지)
- "getImpactRulesByNamePrefix"(345페이지)

calculateImpact

calculateImpact 메서드는 CMDB에 정의된 규칙에 따라 주어진 CI의 영향을 받는 CI를 계산합니다.

그리고 규칙의 이벤트 트리거로 인한 결과를 보여 줍니다. calculateImpact의 identifier 출력은 "getImpactPath"(345페이지)에 대한 입력으로 사용됩니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
impactCategory	시뮬레이션 중인 규칙을 트리거하는 이벤트 유형입니다.
IDs	CMDB 또는 글로벌 ID 요소의 컬렉션입니다.
impactRulesNames	ImpactRuleName 요소의 컬렉션입니다.
severity	트리거 이벤트의 심각도입니다.

출력

매개 변수	설명
impactTopology	자세한 내용은 "Topology"(329페이지)를 참조하십시오.
identifier	서버 응답에 대한 키입니다.

getImpactPath

getImpactPath 메서드는 영향 받는 CI와 이 CI에 영향을 주는 CI 사이의 경로를 나타내는 토폴로지 그래프를 검색합니다.

"calculateImpact"(344페이지)의 identifier 출력은 getImpactPath의 identifier 입력 인수로 사용됩니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
identifier	calculateImpact가 반환한 서버 응답에 대한 키입니다.
relation	impactTopology 요소의 calculateImpact가 반환한 "ShallowRelation" 중 하나를 기반으로 한 관계입니다.

출력

매개 변수	설명
impactPathTopology	CI 컬렉션 및 ImpactRelations 컬렉션입니다.
comments	내부 전용입니다.

ImpactRelations 요소는 ID, type, end1ID, end2ID, rule, action으로 구성됩니다.

getImpactRulesByNamePrefix

getImpactRulesByNamePrefix 메서드는 접두사 필터를 사용하여 규칙을 검색합니다.

이 메서드는 접두사를 사용하여 이름을 지정한 영향 규칙에 적용됩니다. 이름의 접두사는 규칙이 적용되는 컨텍스트를 나타냅니다(예: SAP_myrule, ORA_myrule 등). 이 메서드는 모든 영향 규칙 이름에서 ruleNamePrefixFilter 인수로 지정된 접두사로 시작하는 이름을 필터링합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
ruleNamePrefixFilter	일치시킬 규칙 이름의 첫 문자를 포함하는 문자열입니다.

출력

매개 변수	설명
impactRules	impactRules는 0개 이상의 impactRule로 구성됩니다. 변경으로 인한 영향을 지정하는 impactRule은 ruleName, description, queryName 및 isActive로 구성됩니다.

실제 상태 웹 서비스 API

실제 상태 웹 서비스 API는 기본적으로 Service Manager에서 특정 CMDB ID 또는 글로벌 ID와 특정 고객 ID에 대한 실제 상태 정보를 검색하는 데 사용됩니다. API는 **통합/SM 쿼리** 폴더 아래에서 일치하는 폴더를 찾고, CMDB ID 또는 글로벌 ID를 조건으로 사용하여 TQL를 실행하며, 쿼리의 출력을 반환합니다.

웹 서비스 URL: http://[machine_name]:8080/axis2/services/ucmdbSMService

웹 서비스 스키마: http://[machine_name]:8080/axis2/services/ucmdbSMService?xsd=xsd0

흐름

API 메서드가 호출되면 이 메서드는 **통합/SM 쿼리** 폴더에서 적절한 쿼리를 찾으려고 합니다. 이 메서드는 이름이 **Root**인 **QueryElement**를 검색하여 요청한 CMDBID/GlobalID 유형을 이후의 폴더에 있는 쿼리 중 하나와 일치시키며, 일치하는 항목이 없을 경우 요청한 CMDBID/GlobalID와 같은 유형의 **QueryNode**를 사용하려고 합니다. 적절한 Query 및 QueryNode를 찾은 경우 CMDBID/GlobalID를 QueryNode의 조건으로 넣어 Query를 실행합니다. 그러면 API의 호출자에 결과가 반환됩니다.

변환을 사용하여 결과 조작

결과 XML에 추가 변환을 적용해야 하는 경우가 있습니다(예: 모든 디스크의 크기를 요약하고 이 요약을 CI에 추가적인 특성으로 추가하는 경우). TQL 결과에서 변환을 추가하려면 다음과 같이 **[tql_name].xslt**라는 리소스를 어댑터 구성에 배치합니다. **어댑터 관리 > ServiceDeskAdapter7-1 > 구성 파일 > [tql_name].xslt**

실제 상태 웹 서비스 API의 로그

UCMDB의 로그 구성은 **UCMDBServer/Conf/log**의 여러 ***.properties** 파일에 있습니다.

SM 실제 상태 흐름의 로그를 보려면 다음을 수행합니다.

1. **cmdb_soaapi.properties** 파일을 열고 다음과 같이 로그 수준을 DEBUG로 변경합니다. **loglevel=DEBUG**
2. **fcmdb.properties** 파일을 열고 다음과 같이 로그 수준을 DEBUG로 변경합니다. **loglevel=DEBUG**
3. 서버가 변경 사항을 검색하도록 1분 정도 기다립니다.
4. SM에서 실제 상태를 실행합니다.
5. **UCMDBServer/Runtime/log**에서 다음 로그 파일을 봅니다.

- cmdb.soaapi.log
- fcmdb.log

루트 컨텍스트를 변경한 후에 복제된 CI의 실제 상태 활성화

UCMDB 액세스에 사용되는 루트 컨텍스트를 변경한 경우에는 다음과 같이 구성을 변경하여 복제된 CI의 실제 상태를 활성화해야 합니다.

1. **UCMDBServer\deploy\axis2\WEB-INF**에서 **web.xml** 파일을 엽니다.
2. 다음 **servlet init** 매개 변수를 AxisServlet에 추가합니다(줄 28 뒤에 다음 네 줄 붙여넣기).

   <init-param>

   <param-name>axis2.find.context</param-name>

   <param-value>false</param-value>

   </init-param>

   이 설정을 사용하면 Axis2가 컨텍스트 루트를 계산하지 않고 axis2.xml에서 명시적으로 찾도록 합니다.
3. **UCMDBServer\deploy\axis2\WEB-INF\conf**에서 **axis2.xml** 파일을 엽니다.
4. 줄 58에서 **contextRoot** 매개 변수의 주석을 제거하고 다음과 같이 편집합니다.

   <parameter name="contextRoot" locked="false">test/axis2</parameter>

   (test는 cmdb.xml의 새 루트 컨텍스트).

   참고: test/axis2 시작 부분에는 슬래시가 없습니다.

UCMDB 웹 서비스 API 사용 사례

다음의 사용 사례에서는 두 가지 시스템을 가정합니다.

- HP Universal CMDB 서버
- **구성 항목 저장소를 포함하는 타사 시스템**

이 섹션에는 다음 항목이 포함됩니다.

CMDB 채우기

사용 사례:

- 타사 자산 관리가 자산 관리에서만 사용할 수 있는 정보로 CMDB를 업데이트합니다.
- 여러 타사 시스템에서 CMDB를 채워 변경 내용을 추적하고 영향 분석을 수행할 수 있는 중앙 CMDB를 만듭니다.
- 타사 시스템에서 CMDB 쿼리 기능을 활용하기 위해 타사 비즈니스 논리에 따라 구성 항목 및 관계를 만듭니다.

CMDB 쿼리

사용 사례:

- 타사 시스템에서 SAP TQL의 결과를 가져와 SAP 시스템을 나타내는 구성 항목 및 관계를 가져옵니다.
- 타사 시스템에서 지난 5시간 동안 추가되었거나 변경된 Oracle 서버 목록을 가져옵니다.
- 타사 시스템에서 호스트 이름에 *lab*이라는 하위 문자열이 포함된 서버 목록을 가져옵니다.
- 타사 시스템에서 주어진 CI의 인접 항목을 가져와서 이 주어진 CI와 관련된 요소를 찾습니다.

클래스 모델 쿼리

사용 사례:

- 타사 시스템을 통해 사용자가 CMDB에서 검색할 데이터 집합을 지정할 수 있습니다. 클래스 모델을 기반으로 사용자 인터페이스를 작성하면 사용자에게 가능한 속성을 표시하고 필수 데이터를 입력하라는 프롬프트를 표시할 수 있습니다. 그러면 사용자가 검색할 정보를 선택할 수 있습니다.
- 사용자가 UCMDB 사용자 인터페이스에 액세스할 수 없을 때 타사 시스템에서 클래스 모델을 탐색합니다.

변경 영향 분석

사용 사례:

타사 시스템에서 지정된 호스트에 대한 변경에 의해 영향을 받을 수 있는 비즈니스 서비스 목록을 출력합니다.

예

다음 코드 샘플을 참조하십시오.

- The Example Base Class
- Query Example
- Update Example
- Class Model Example
- Impact Analysis Example

이러한 파일은 다음 디렉터리에 있습니다.

\\<UCMDB 루트 디렉터리>\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\WebServiceAPI_Samples

12장: 데이터 흐름 관리 Java API

이 장의 내용:

데이터 흐름 관리 Java API 사용

참고: 이 장은 온라인 문서 라이브러리에서 볼 수 있는 DFM API Javadoc와 함께 사용하십시오.

이 장에서는 타사 또는 사용자 지정 도구에서 HP 데이터 흐름 관리 Java API를 사용하여 데이터 흐름을 관리하는 방법을 설명합니다. 이 API는 다음을 수행할 수 있는 메서드를 제공합니다.

- **자격 증명 관리** 보기, 추가, 업데이트, 제거
- **작업 관리** 상태 보기, 활성화, 비활성화
- **프로브 범위 관리** 보기, 추가, 업데이트
- **트리거 관리** 트리거 CI 추가 또는 제거, 트리거 TQL 추가, 제거 또는 사용하지 않도록 설정
- **일반 데이터 보기** 도메인 및 프로브의 데이터

다음 서비스는 디스커버리 서비스 패키지에서 사용할 수 있습니다.

- **DDMConfigurationService.** Data Flow Probe, 클러스터, IP 범위 및 자격 증명을 구성하기 위한 서비스입니다. Universal Discovery 서버는 XML 파일로 또는 Data Flow Probe를 통해 구성할 수 있습니다.
- **DDMManagementService.** Universal Discovery 실행의 진행률, 결과 및 오류를 분석하고 보기 위한 서비스입니다.
- **DDMSoftwareSignatureService.** Data Flow Probe 구성 요소에 의해 디스커버리될 소프트웨어 항목을 정의하기 위한 서비스입니다. 이러한 정의는 시스템 전체에 적용됩니다. Data Flow Probe 구성 요소가 두 개 이상 정의되는 경우 그러한 정의는 모든 구성 요소에 적용됩니다.
- **DDMZoneService.** 영역 기반 디스커버리를 관리하기 위한 서비스입니다.

이러한 서비스 외에도, Jython 어댑터를 만드는 데 사용되는 데이터 흐름 관리 API가 있습니다. 자세한 내용은 "Jython 어댑터 개발"(36페이지)을 참조하십시오.

사용 권한

관리자는 API에 연결하는 데 필요한 로그인 자격 증명을 제공합니다. API 클라이언트에는 CMDB에서 정의된 통합 사용자의 사용자 이름과 비밀번호가 필요합니다. 이러한 사용자는 CMDB의 실제 사용자가 아니라 CMDB에 연결하는 응용 프로그램을 나타냅니다.

또한 사용자가 로그인하기 위해서는 **SDK에 액세스** 일반 수행 권한이 있어야 합니다.

주의: API 인증 권한이 있는 일반 사용자도 API 클라이언트를 사용할 수 있습니다. 하지만 이 옵션은 권장되지 않습니다.

자세한 내용은 "통합 사용자 만들기"(312페이지)를 참조하십시오.

13장: 데이터 흐름 관리 웹 서비스 API

이 장의 내용:

데이터 흐름 관리 웹 서비스 API 개요

이 장에서는 타사 또는 사용자 지정 도구에서 HP 데이터 흐름 관리 웹 서비스 API를 사용하여 데이터 흐름을 관리하는 방법을 설명합니다.

HP 데이터 흐름 관리 웹 서비스 API는 응용 프로그램을 HP Universal CMDB와 통합하는 데 사용됩니다. API는 다음을 수행할 수 있는 메서드를 제공합니다.

- **자격 증명 관리** 보기, 추가, 업데이트, 제거
- **작업 관리** 상태 보기, 활성화, 비활성화
- **프로브 범위 관리** 보기, 추가, 업데이트
- **트리거 관리** 트리거 CI 추가 또는 제거, 트리거 TQL 추가, 제거 또는 사용하지 않도록 설정
- **일반 데이터 보기** 도메인 및 프로브의 데이터

HP 데이터 흐름 관리 웹 서비스 사용자는 다음 사항을 잘 알고 있어야 합니다.

- SOAP 사양
- 개체 지향 프로그래밍 언어(예: C++, C# 또는 Java)
- HP Universal CMDB
- 데이터 흐름 관리

참고:

- 사용자가 로그인하려면 **레거시 API 실행** 일반 수행 권한이 있어야 합니다.
- 로그인한 사용자가 메서드에 액세스하려면 **디스커버리 및 통합 실행** 일반 수행 권한이 있어야 합니다.

사용 가능한 작업에 관한 전체 문서는 *HP Universal Discovery 스키마 참조*를 참조하십시오. 이러한 파일은 다음 폴더에 있습니다.

<UCMDB 루트 디렉터리>\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\DDM_Schema\webframe.html

규칙

이 장에서는 다음 규칙을 사용합니다.

- 이 스타일 Element는 메서드가 반환했거나 메서드에 전달된 구조를 포함하여 항목이 데이터베이스의 엔터티 또는 스키마에 정의된 요소임을 나타냅니다. 일반 텍스트는 해당 항목이 일반 컨텍스트로 설명됨을 나타냅니다.
- 데이터 흐름 관리 요소 및 메서드 인수는 스키마에 지정된 경우에만 의미가 있습니다. 이는 일반적으로 클래스 이름 또는 클래스 인스턴스의 일반 참조가 대문자로 표시됨을 의미합니다. 메서드에 대한 인수나 요소는 대문자로 표시되지 않습니다. 예를 들어 credential은 메서드에 전달된 Credential 유형의 요소입니다.

HP 데이터 흐름 관리 웹 서비스 호출

HP 데이터 흐름 관리 웹 서비스 API를 통해 표준 SOAP 프로그래밍 기술을 사용하여 서버측 메서드를 호출할 수 있습니다. 명령문을 구문 분석할 수 없거나 메서드를 호출하는 데 문제가 있는 경우 API 메서드는 SoapFault 예외를 발생시킵니다. SoapFault 예외가 발생하면 해당 서비스에서 하나 이상의 오류 메시지, 오류 코드 및 예외 메시지 필드를 채웁니다. 오류가 없으면 호출 결과가 반환됩니다.

서비스를 호출하려면 다음을 사용합니다.

- 프로토콜: 서비스 구성에 따라 http 또는 https
- URL: <UCMDB 서버>:8080/axis2/services/DiscoveryService
- 기본 비밀번호: "admin"
- 기본 사용자 이름: "admin"

SOAP 프로그래머가 다음 위치에서 WSDL에 액세스할 수 있습니다.

- axis2/services/DiscoveryService?wsdl

데이터 흐름 관리 메서드 및 데이터 구조

이 섹션에서는 데이터 흐름 관리 웹 서비스 API 메서드 및 데이터 구조를 나열하고 각 사용 방법에 대해 간단히 요약하여 설명합니다. 각 작업의 요청 및 응답에 관한 전체 문서는 *HP Universal Discovery 스키마 참조*를 참조하십시오.

이 섹션에는 다음 항목이 포함됩니다.

- "데이터 구조"(353페이지)
- "디스커버리 작업 메서드 관리"(353페이지)
- "트리거 메서드 관리"(355페이지)

데이터 구조

데이터 흐름 관리 웹 서비스 API에 사용되는 몇 가지 데이터 구조가 있습니다.

CIProperties

CIProperties는 여러 컬렉션 중 하나의 컬렉션입니다. 각 컬렉션에는 다른 데이터 유형의 속성이 포함되어 있습니다. 예를 들어 dateProps 컬렉션, strListProps 컬렉션, xmlProps 컬렉션 등이 있습니다.

각 유형 컬렉션에는 지정된 유형의 개별 속성이 포함되어 있습니다. 이러한 속성 요소의 이름은 컨테이너와 동일하지만 단수형입니다. 예를 들어, datePropse에 dateProp 요소가 포함됩니다. 각 속성은 이름-값 쌍입니다.

*HP Universal Discovery 스키마 참조*에서 CIProperties를 참조하십시오.

IPList

IP 요소 목록으로서 각 요소에 IPv4 또는 IPv6 주소가 포함됩니다.

*HP Universal Discovery 스키마 참조*에서 IPList를 참조하십시오.

IPRange

IPRange에는 두 개의 요소, 즉 Start와 End가 있습니다. 각 요소에는 IPv4 또는 IPv6 주소인 Address 요소가 포함되어 있습니다.

*HP Universal Discovery 스키마 참조*에서 IPRange를 참조하십시오.

Scope

두 개의 IPRanges입니다. Exclude는 작업에서 제외할 IPRanges의 컬렉션입니다. Include는 작업에 포함할 IPRanges의 컬렉션입니다.

*HP Universal Discovery 스키마 참조*에서 Scope를 참조하십시오.

디스커버리 작업 메서드 관리

activateJob

지정된 작업을 활성화합니다.

"코드 샘플"(363페이지)을 참조하십시오.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
JobName	작업 이름입니다.

deactivateJob

지정된 작업을 비활성화합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
JobName	작업 이름입니다.

dispatchAdHocJob

프로브 Ad hoc에 대한 작업을 발송합니다. 작업은 활성 상태여야 하고 지정된 트리거 CI를 포함해야 합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
JobName	작업 이름입니다.
CIID	트리거 CI의 ID입니다.
ProbeName	프로브의 이름입니다.
Timeout	밀리초 단위

getDiscoveryJobsNames

작업 이름의 목록을 반환합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.

출력

매개 변수	설명
strList	작업 이름의 목록입니다.

isJobActive

작업이 활성 상태인지 여부를 확인합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
JobName	확인할 작업의 이름입니다.

출력

매개 변수	설명
JobState	작업이 활성 상태일 경우 true입니다.

트리거 메서드 관리

addTriggerCI

지정된 작업에 새 트리거 CI를 추가합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
JobName	작업 이름입니다.
CIID	트리거 CI의 ID입니다.

addTriggerTQL

지정된 작업에 새 트리거 TQL을 추가합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
JobName	작업 이름입니다.

매개 변수	설명
TqlName	추가할 TQL의 이름입니다.

disableTriggerTQL

TQL이 작업을 트리거하지 않도록 하지만 작업을 트리거하는 쿼리 목록에서 영구적으로 제거하지는 않습니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
JobName	작업 이름입니다.

removeTriggerCI

작업을 트리거하는 CI 목록에서 지정된 CI를 제거합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
JobName	작업 이름입니다.
CIID	트리거 CI의 ID입니다.

removeTriggerTQL

작업을 트리거하는 쿼리 목록에서 지정된 TQL을 제거합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
JobName	확인할 작업 이름의 컬렉션입니다.
CIID	제거할 TQL의 ID입니다.

setTriggerTQLProbesLimit

작업에서 TQL이 활성 상태인 프로브를 지정된 목록으로 제한합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
JobName	작업 이름입니다.
tqlName	TQL 이름입니다.
probesLimit	TQL이 활성 상태인 프로브의 목록입니다.

도메인 및 프로브 데이터 메서드

getDomainType

도메인 유형을 반환합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
domainName	도메인의 이름입니다.

출력

매개 변수	설명
domainType	도메인 유형입니다.

getDomainsNames

현재 도메인의 이름을 반환합니다.

"코드 샘플"(363페이지)을 참조하십시오.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.

출력

매개 변수	설명
domainNames	도메인 이름의 목록입니다.

getProbeIPs

지정된 프로브의 IP 주소를 반환합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
domainName	확인할 도메인입니다.
probeName	해당 도메인에 사용된 프로브의 이름입니다.

출력

매개 변수	설명
probeIPs	프로브에 있는 주소의 "IPList"입니다.

getProbesNames

지정된 도메인에서 프로브의 이름을 반환합니다.

"코드 샘플"(363페이지)을 참조하십시오.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
domainName	확인할 도메인입니다.

출력

매개 변수	설명
probesName	도메인에 있는 프로브의 목록입니다.

getProbeScope

지정된 프로브의 범위 정의를 반환합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
domainName	확인할 도메인입니다.

매개 변수	설명
probeName	프로브의 이름입니다.

출력

매개 변수	설명
probeScope	프로브의 "Scope"입니다.

isProbeConnected

지정된 프로브가 연결되었지 여부를 확인합니다.

"코드 샘플"(363페이지)을 참조하십시오.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
domainName	확인할 도메인입니다.
probeName	확인할 프로브입니다.

출력

매개 변수	설명
isConnected	프로브가 연결된 경우 true입니다.

updateProbeScope

기존 범위를 다시 정의하는, 지정된 프로브의 범위를 설정합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
domainName	도메인입니다.
probeName	업데이트할 프로브입니다.
newScope	프로브에 설정할 "Scope"입니다.

자격 증명 데이터 메서드

addCredentialsEntry

지정된 도메인의 지정된 프로토콜에 자격 증명 항목을 추가합니다.

"코드 샘플"(363페이지)을 참조하십시오.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
domainName	업데이트할 도메인입니다.
protocolName	프로토콜의 이름입니다.
credentialsEntryParameters	새 자격 증명의 "CIProperties" 컬렉션입니다.

출력

매개 변수	설명
credentialsEntryID	새 자격 증명 항목의 CI ID입니다.

getCredentialsEntriesIDs

지정된 프로토콜에 대해 정의된 자격 증명의 ID를 반환합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
domainName	자격 증명을 가져올 도메인입니다.
protocolName	해당 도메인에 사용된 프로토콜의 이름입니다.

출력

매개 변수	설명
credentialsEntryIDs	도메인에 있는 프로토콜의 자격 증명 ID 목록입니다.

getCredentialsEntry

지정된 프로토콜에 대해 정의된 자격 증명을 반환합니다. 암호화된 특성이 빈 상태로 반환됩니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
domainName	자격 증명을 가져올 도메인입니다.
protocolName	해당 도메인에 사용된 프로토콜의 이름입니다.
credentialsEntryID	가져올 자격 증명 ID입니다.

출력

매개 변수	설명
credentialsEntryParameters	자격 증명의 "CIProperties" 컬렉션입니다.

removeCredentialsEntry

지정된 자격 증명을 프로토콜에서 제거합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
domainName	도메인입니다.
protocolName	해당 도메인에 사용된 프로토콜의 이름입니다.
credentialsEntryID	제거할 자격 증명의 ID입니다.

updateCredentialsEntry

지정된 자격 증명 항목의 속성에 대한 새 값을 설정합니다.

기존 속성이 제거되며 이러한 속성이 설정됩니다. 이 호출에 값이 설정되지 않은 모든 속성이 정의되지 않은 상태로 남아 있습니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
domainName	자격 증명을 업데이트할 도메인입니다.
protocolName	해당 도메인에 사용된 프로토콜의 이름입니다.

매개 변수	설명
credentialsEntryID	업데이트할 자격 증명의 ID입니다.
credentialsEntryParameters	자격 증명의 속성으로 설정할 "CIProperties" 컬렉션입니다.

데이터 새로 고침 메서드

rediscoverCIs

지정된 CI 개체를 디스커버리한 트리거를 찾아 해당 트리거를 다시 실행합니다. **rediscoverCIs**는 비동기적으로 실행됩니다. **checkDiscoveryProgress**를 호출하여 다시 디스커버리가 완료된 시기를 확인합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
CmdbIDs	다시 디스커버리할 개체의 ID 컬렉션입니다.

출력

매개 변수	설명
isSucceed	CI의 재디스커버리에 성공하는 경우 true입니다.

checkDiscoveryProgress

지정된 ID에 대한 최근 **rediscoverCIs** 호출의 진행률을 반환합니다. 응답은 0에서 1 사이의 값입니다. 응답이 1이면 **rediscoverCIs** 호출이 완료된 것입니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
CmdbIDs	재디스커버리 호출에서 추적할 개체의 ID 컬렉션입니다.

출력

매개 변수	설명
progress	완료된 작업의 진행률은 1이고 완료되지 않은 작업의 진행률은 1 미만의 소수입니다.

rediscoverViewCIs

지정된 보기를 채울 데이터를 만든 트리거를 찾아 해당 트리거를 다시 실행합니다. rediscoverViewCIs는 비동기적으로 실행됩니다. checkViewDiscoveryProgress를 호출하여 다시 디스커버리가 완료된 시기를 확인합니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
viewName	확인할 보기입니다.

출력

매개 변수	설명
isSucceed	CI의 재디스커버리에 성공하는 경우 true입니다.

checkViewDiscoveryProgress

지정된 보기에 대한 최근 rediscoverViewCIs 호출의 진행률을 반환합니다. 응답 값은 0-1입니다. 응답이 1이면 rediscoverCIs 호출이 완료된 것입니다.

입력

매개 변수	설명
cmdbContext	자세한 내용은 "CmdbContext"(326페이지)를 참조하십시오.
viewName	확인할 보기의 컬렉션입니다.

출력

매개 변수	설명
progress	완료된 작업의 진행률은 1이고 완료되지 않은 작업의 진행률은 1 미만의 소수입니다.

코드 샘플

```
import java.net.URL;
import org.apache.axis2.transport.http.HTTPConstants;
import org.apache.axis2.transport.http.HttpTransportProperties;
import com.hp.ucmdb.generated.params.discovery.*;
import com.hp.ucmdb.generated.services.*;
import com.hp.ucmdb.generated.types.*;
```

```
public class test {
   static final String HOST_NAME = "<my_hostname>";
   static final int PORT = 8080;
   private static final String PROTOCOL = "http";
   private static final String FILE = "/axis2/services/DiscoveryService";

   private static final String PASSWORD = "<my_password>";
   private static final String USERNAME = "<my_username>";

   private static CmdbContext cmdbContext = new CmdbContext("ws tests");

   public static void main(String[] args) throws Exception {
      // Get the stub object
      DiscoveryService discoveryService = getDiscoveryService();

      // Activate Job
      discoveryService.activateJob(new ActivateJobRequest(
         "Range IPs by ICMP", cmdbContext));

      // Get domain & probes info
      getProbesInfo(discoveryService);
      // Add credentilas entry for ntcmd protcol
      addNTCMDCredentialsEntry();
   }

   public static void addNTCMDCredentialsEntry() throws Exception {
      DiscoveryService discoveryService = getDiscoveryService();

      // Get domain name
      StrList domains =
         discoveryService.getDomainsNames(
            new GetDomainsNamesRequest(cmdbContext)).
            getDomainNames();
      if (domains.sizeStrValueList() == 0) {
         System.out.println("No domains were found, can't create credentials");
         return;
      }
      String domainName = domains.getStrValue(0);
      // Create propeties with one byte param
      CIProperties newCredsProperties = new CIProperties();

      // Add password property - this is of type bytes
      newCredsProperties.setBytesProps(new BytesProps());
      setPasswordProperty(newCredsProperties);

      // Add user & domain properties - these are of type string
      newCredsProperties.setStrProps(new StrProps());
      setStringProperties("protocol_username", "test user", newCredsProperties);
      setStringProperties("ntadminprotocol_ntdomain",
         "test doamin", newCredsProperties);
```

```
        // Add new credentials entry
        discoveryService.addCredentialsEntry(
            new AddCredentialsEntryRequest(domainName,
            "ntadminprotocol",  newCredsProperties, cmdbContext));
        System.out.println("new credentials craeted for domain: " + domainName + " in ntcmd protocol");
    }

    private static void setPasswordProperty(CIProperties newCredsProperties) {
        BytesProp bProp = new BytesProp();
        bProp.setName("protocol_password");
        bProp.setValue(new byte[] {101,103,102,104});
        newCredsProperties.getBytesProps().addBytesProp(bProp);
    }

    private static void setStringProperties(String propertyName, String value, CIProperties
newCredsProperties) {
        StrProp strProp = new StrProp();
        strProp.setName(propertyName);
        strProp.setValue(value);
        newCredsProperties.getStrProps().addStrProp(strProp);
    }


    private static void getProbesInfo(DiscoveryService discoveryService) throws Exception {
        GetDomainsNamesResponse result = discoveryService.getDomainsNames(new
GetDomainsNamesRequest(cmdbContext ));
        // Go over all the domains
        if (result.getDomainNames().sizeStrValueList() > 0) {
            String domainName =
            result.getDomainNames().getStrValue(0);
            GetProbesNamesResponse probesResult =
                discoveryService.getProbesNames(
                    new GetProbesNamesRequest(domainName, cmdbContext));
            // Go over all the probes
            for (int i=0; i<probesResult.getProbesNames().sizeStrValueList(); i++) {
                String probeName = probesResult.getProbesNames().getStrValue(i);
                // Check if connected
                IsProbeConnectedResponce connectedRequest =
                    discoveryService.isProbeConnected(
                        new IsProbeConnectedRequest(
                        domainName, probeName, cmdbContext));
                Boolean isConnected = connectedRequest.getIsConnected();
                // Do something ...
                System.out.println("probe " + probeName + " isconnect=" + isConnected);
            }
        }
    }

    private static DiscoveryService getDiscoveryService() throws Exception {
```

```
        DiscoveryService discoveryService = null;
        try {
            // Create service
            URL url = new URL(PROTOCOL,HOST_NAME,PORT, FILE);
            DiscoveryServiceStub serviceStub =
                new DiscoveryServiceStub(url.toString());

            // Authenticate info
            HttpTransportProperties.Authenticator auth =
                new HttpTransportProperties.Authenticator();
            auth.setUsername(USERNAME);
            auth.setPassword(PASSWORD);
            serviceStub._getServiceClient().getOptions().setProperty(
                HTTPConstants.AUTHENTICATE,auth);

            discoveryService = serviceStub;
        } catch (Exception e) {
            throw new Exception("cannot create a connection to service ", e);
        }
        return discoveryService;
    }
}
```

자격 증명 추가 작업의 예

```
import java.net.URL;
import org.apache.axis2.transport.http.HTTPConstants;
import org.apache.axis2.transport.http.HttpTransportProperties;
import com.hp.ucmdb.generated.params.discovery.*;
import com.hp.ucmdb.generated.services.DiscoveryService;
import com.hp.ucmdb.generated.services.DiscoveryServiceStub;
import com.hp.ucmdb.generated.types.BytesProp;
import com.hp.ucmdb.generated.types.BytesProps;
import com.hp.ucmdb.generated.types.CIProperties;
import com.hp.ucmdb.generated.types.CmdbContext;
import com.hp.ucmdb.generated.types.StrList;
import com.hp.ucmdb.generated.types.StrProp;
import com.hp.ucmdb.generated.types.StrProps;

public class test {
    static final String HOST_NAME = "hostname";
    static final int PORT = 8080;
    private static final String PROTOCOL = "http";
    private static final String FILE = "/axis2/services/DiscoveryService";

    private static final String PASSWORD = "admin";
    private static final String USERNAME = "admin";
```

```
    private static CmdbContext cmdbContext = new CmdbContext("ws tests");

    public static void main(String[] args) throws Exception {
        // Get the stub object
        DiscoveryService discoveryService = getDiscoveryService();

        // Activate Job
        discoveryService.activateJob(new ActivateJobRequest("Range IPs by ICMP", cmdbContext));

        // Get domain & probes info
        getProbesInfo(discoveryService);
        // Add credentilas entry for ntcmd protcol
        addNTCMDCredentialsEntry();
    }

    public static void addNTCMDCredentialsEntry() throws Exception {
        DiscoveryService discoveryService = getDiscoveryService();

        // Get domain name
        StrList domains =
            discoveryService.getDomainsNames(new GetDomainsNamesRequest
(cmdbContext)).getDomainNames();
        if (domains.sizeStrValueList() == 0) {
            System.out.println("No domains were found, can't create credentials");
            return;
        }
        String domainName = domains.getStrValue(0);
        // Create propeties with one byte param
        CIProperties newCredsProperties = new CIProperties();

        // Add password property - this is of type bytes
        newCredsProperties.setBytesProps(new BytesProps());
        setPasswordProperty(newCredsProperties);

        // Add user & domain properties - these are of type string
        newCredsProperties.setStrProps(new StrProps());
        setStringProperties("protocol_username", "test user", newCredsProperties);
        setStringProperties("ntadminprotocol_ntdomain", "test doamin", newCredsProperties);

        // Add new credentials entry
        discoveryService.addCredentialsEntry(new AddCredentialsEntryRequest(domainName,
"ntadminprotocol",  newCredsProperties, cmdbContext));
        System.out.println("new credentials craeted for domain: " + domainName + " in ntcmd protocol");
    }

    private static void setPasswordProperty(CIProperties newCredsProperties) {
        BytesProp bProp = new BytesProp();
        bProp.setName("protocol_password");
        bProp.setValue(new byte[] {101,103,102,104});
```

```
        newCredsProperties.getBytesProps().addBytesProp(bProp);
    }

    private static void setStringProperties(String propertyName, String value, CIProperties
newCredsProperties) {
        StrProp strProp = new StrProp();
        strProp.setName(propertyName);
        strProp.setValue(value);
        newCredsProperties.getStrProps().addStrProp(strProp);
    }

    private static void getProbesInfo(DiscoveryService discoveryService) throws Exception {
        GetDomainsNamesResponse result = discoveryService.getDomainsNames(new
GetDomainsNamesRequest(cmdbContext ));
        // Go over all the domains
        if (result.getDomainNames().sizeStrValueList() > 0) {
            String domainName = result.getDomainNames().getStrValue(0);
            GetProbesNamesResponse probesResult =
                discoveryService.getProbesNames(new GetProbesNamesRequest(domainName,
cmdbContext));
            // Go over all the probes
            for (int i=0; i<probesResult.getProbesNames().sizeStrValueList(); i++) {
                String probeName = probesResult.getProbesNames().getStrValue(i);
                // Check if connected
                IsProbeConnectedResponce connectedRequest =
                    discoveryService.isProbeConnected(new IsProbeConnectedRequest(domainName,
probeName, cmdbContext));
                Boolean isConnected = connectedRequest.getIsConnected();
                // Do something ...
                System.out.println("probe " + probeName + " isconnect=" + isConnected);
            }
        }
    }

    private static DiscoveryService getDiscoveryService() throws Exception {
        DiscoveryService discoveryService = null;
        try {
            // Create service
            URL url = new URL(PROTOCOL,HOST_NAME,PORT, FILE);
            DiscoveryServiceStub serviceStub = new DiscoveryServiceStub(url.toString());

            // Authenticate info
            HttpTransportProperties.Authenticator auth = new HttpTransportProperties.Authenticator();
            auth.setUsername(USERNAME);
            auth.setPassword(PASSWORD);
            serviceStub._getServiceClient().getOptions().setProperty(HTTPConstants.AUTHENTICATE,auth);

            discoveryService = serviceStub;
        } catch (Exception e) {
            throw new Exception("cannot create a connection to service ", e);
```

```
            }
            return discoveryService;
        }
    } // End class
```

문서 피드백 보내기

이 문서에 대한 의견이 있는 경우 전자 메일로 문서 팀에 의견을 보내 주십시오. 이 시스템에 전자 메일 클라이언트가 구성되어 있을 경우 위의 링크를 클릭하면 제목 줄에 다음 정보가 포함된 전자 메일 창이 열립니다.

개발자 참조 안내서에 대한 피드백(Universal CMDB 10.20)

전자 메일에 피드백을 추가하고 보내기를 클릭하기만 하면 됩니다.

전자 메일 클라이언트를 사용할 수 없으면 위의 정보를 웹 메일 클라이언트에서 새 메시지에 복사하고 피드백을 cms-doc@hp.com에 보내십시오.

피드백을 보내 주셔서 감사합니다!