**ARadon**

Radon(라돈)은 그리스 신화에 나오는 커다란 뱀 형상의 드래곤으로 헤스페리데스의 황금사과를 지키는 괴물로, 불사의 존재였습니다. 신화에 따르면 모든 신들의 여왕 헤라(Hera)는 제우스(Zeus)와 결혼할 때 제우스에게 황금사과나무를 결혼선물로 받습니다. 헤라는 이 나무를 아틀라스(Atlas)의 세딸들인 헤스페리데스(Hesperides)에게 지키게 합니다. 그러나 그녀들이 제대로 이 나무를 지키지 못하여 헤라는 라돈(Radon)이라는 용에게 이 나무를 지키게 합니다.

후에 헤라클레스(Hercules)는 12가지 노역중 하나로 이 황금사과를 가지러 아틀라스산으로 가는데 헤라클레스의 황금사과 임무는 아틀라스가 대신 해 주었기 때문에, 헤라클레스는 라돈과 싸우지 않았습니다.

과일 나무를 지키고 있는 커다란 뱀의 이야기는 고대 근동지역에서 흔히 볼 수 있는 전설로

에리스토파네스의 희곡 개구리들에서도 라돈이 언급되고 있는데 여기서 라돈은 100개의 머리를 가지고 있고 서로 다른 말을 할 수 있다고 합니다.

1. **ARadon의 목표 :**

* HTTP Protocol을 이용해 추상화된 형태의 공용 서비스 제공 :
* 분산(Process, Repository) Let
* 서비스 생태계(EcoSystem Service) 구축
  1. **Metaphor(고차원 언어와 저차원언어)**

ARadon의 메타포에 대해서 이야기 하기전에 전산한 개론에 배우는 일반적인 얘기를 먼저 하겠습니다.

일반적으로 문제에 합당한 최고 고차원 언어를 사용하면 생산성과 품질이 높아집니다. 고차원이란 명품 등의 Quality가 높다라는 의미가 아니라 추상성이 높은 언어를 고차원 혹은 고급언어라고 부릅니다. 전통적으로 어셈블러는 거의 기계어와 동일하므로 추상성이 낮기 때문에 저차원 언어라 부르고 기계어를 1GL, 어셈블리어를 2GL로 부릅니다. 많이 알려진 PL/1, C, Java등은 고급언어 혹은 3GL 언어라 불리며 4GL은 좀더 자연어에 가까우면서도 데이터베이스 접근을 위한 언어 PL/SQL 같은 언어를 4GL이라고 표현합니다.

4GL 언어의 프로그램 문장은 다음과 같은 형태를 가집니다.  
EXTRACT ALL EMPLOYEES WHERE "SALARY" TOTAL MORE THAN \5,000,000   
  
5GL 또는 5세대 언어는 시각적인 그래픽 인터페이스를 통해, 통상 3GL 또는 4GL 언어 컴파일러로 컴파일 할 수 있는 원시코드를 만들어 줍니다. 5세대 언어는 아직 없지만 5세대 프로그래밍 도구로 MS의 #시리즈, 볼랜드나 IBM등의 회사가 자바 같은 언어를 쓴 응용프로그램을 개발하기 위한 도구를 만들었습니다.

4GL이나 5GL로 문제를 풀지 못했다면 기존의 훌륭한 전통적 프로그래밍 언어(C, Java, 코볼\_를 선택해야 합니다.. 이 시도가 실패한다면, 기계어나 어셈블리 수준으로 내려가야 합니다. (사실 최근에는 4GL이나 5GL 언어를 많이 사용하지 않습니다. )

고차원 언어에서 저차원 언어로 내려가는 이유는 무엇일까요?

기능이 첫째 이유입니다. 흔히 고차원 언어는 특정 문제 영역에만 초점을 맞추므로, 잠정적으로 새로운 문제에는 유용성이 떨어질 가능성이 있습니다.

효율성이 그 다음 이유입니다. 일반적으로 언어 수준이 높아질수록 컴퓨터 자원 효율은 떨어지며, 최악의 경우에는 감내하기 어려운 상태에 이르기도 합니다.

제안된 해결책은(이식성, 안정성, 인간 공학, 테스트 편의성, 이해 용이성, 수정 용이성 등) 반드시 품질 속성으로 평가되어야 합니다. 즉 품질 낮은 해결책으로 낭비하는 시간 비용이 값싼 프로그래밍 해결책으로 절약하는 비용을 넘어서선 안됩니다.

그럼 반대로 저차원 언어에서 고차원 언어로 올라가는 이유는 무엇일까요?

고차원 언어가 제공하는 장점은 많습니다.

1. 생산성이 높아진다. 가끔 놀랄 만큼 높아지기도 한다.
2. 고차원 언어는 오류가 발생할 기회를 제거하므로 전반적인 구현 수준을 높인다. 예를 들어, 어셈블러 프로그래머는 하드웨어 레지스터를 조작해야 하므로 레지스터 할당 오류를 저지를 위험이 있다. 고차원 언어 프로그래머는 레지스터 오류를 저지를 가능성이 없다. 고차원 언어에서는 레지스터를 참조하지도 못하고 참조할 필요도 없기 때문이다.
3. 고차원 해법은 코드 행수가 적다, 따라서 오류가 발생할 기회도 더 적다.
4. 고차원 언어로는 구조적 코드를 작성하기 쉽다. 흥미롭게도, 최고 고차원 해법은 코드가 구조적일 필요도 없다.
5. 고차원 언어로 작성한 코드는 이식성이 더 놓다.
6. 고차원 언어로 작성한 코드는 유지 보수성이 더 높다.
7. 고차원 언어로 작성한 코드는 테스트가 더 용이하다.

결론적으로 언어차원이 높아질수록 대다수 품질 속성이 좋아진다는 뜻 입이다. 앞서 언급했지만 효율성은 이와 반대입니다. 즉 일반적으로 저차원 언어가 가지는 장점은 고차원 언어가 가지는 단점이 되고 그 역도 마찬가지입니다. 극히 효율적인 해법이 필요한 문제라면, 품질과 효율성 사이에 적절한 타협점을 찾을 때까지 언어수준을 낮추어야 합니다. 생산성과 품질은 좀처럼 함께 가지 못하지만 고차원 언어를 사용한다면 두 토끼를 모두 잡을 가능성이 커집니다.

사실 ICS에서 사용되었던 액션스크립트는 고차원 언어입니다. ICS를 사용하면 생산성이 높아진다는 얘기는 결국 좀더 추상성이 높은 언어를 사용하면 생산성이 높아진다는 말과 같은 말입니다. 이전의 ICafe의 Thoth 스크립트도 마찬가지죠. 결국 본질을 들여다보면 고차원 언어를 사용하는 장점을 활용하겠다는 얘기입니다.

그러나 고차원 언어를 새로 정의하는 것이 만병 통치약이 되지는 못합니다. 첫째로 고차원 언어 자체가 가지는 오류나 코드 구조등에 문제가 있다면 오히려 생산성이 저하될수도 있습니다. 두번째로 새로운 고차원 언어를 배워야 하는 Learning Curve 시간입니다.

어쨌거나 이와 같은 관점에서 보면 ARadon은 사실 거의 최고 수준의 고차원 레벨의 언어를 사용합니다. 게다가 그 언어는 자체적으로 정한게 아니라 이미 정해져 있고 대부분의 사용자는 그 언어에 이미 익숙합니다. 언어의 메소드는 PUT, GET, POST, DELETE 이게 답니다. 그리고 대부분의 사용자가 GET,POST를 브라우저를 통해 매번 사용하고 있습니다. ARadon은 특정 URI(Address Key)를 PUT, GET, POST, DELETE 메소드로 호출함으로써 기능(서비스)을 사용하겠다는 뜻입니다.

ARadon은 고차원 언어가 가지는 저효율성의 문제를 분산이라는 다른 방식으로 해결합니다. ARadon을 사용할 때 PUT, GET, POST, DELETE 메소드와 URI가 전부라면 URI에 매핑된 서비스(혹은 기능)은 상세 구현에 대해 호출자에게 알려줄 사항이 거의 없기 때문에 분산으로 구현이 되어 저효율성의 문제를 상쇄시킬 수 있습니다.

**1.2 비용, 품질, 기간의 의자 뺏기 게임**

혹자는 소프트웨어 개발에 있어서 비용, 품질, 일정을 의자 뺐기 게임에 비유하곤 합니다. 보통의 경우에는 의자가 한 개, 운이 좋다면 의자 2개를 두고서 빙글 빙글 돌다가 먼저 앉는 게임 말이죠.

서비스 개발에 있어서 비용과 품질, 일정은 일반적으로 프로젝트의 트리플 제약 조건이 되며

불가능해 보이는 세가지 목표를 모두 충족시키기 위해 CMMI 등의 조직의 프로세스 개선을 시도하려는 시도는 꾸준히 있어 왔지만 모든 시도가 과거를 다시 한번 반복했을 뿐이었습니다.

저는 휠러의 "전산학의 모든 문제는 또 다른 수준의 간접 층으로 해결할 수 있다." 라는 유명한 말을 토대로 좀 다른 시도를 해보기로 했습니다. (휠러는 추가적으로 "그러나 그러면 또 다른 문제가 생기는 것이 일반적이다" 라는 말도 덧붙였습니다. )

다른 시도라고는 하지만 코드의 중복을 해결하기 위해 라이브러리가 나왔고 구조의 중복을 해결하기 위해 프레임 워크가 나왔다면 서비스의 중복을 해결하기 위해 서비스 플랫폼이 필요하지 않을까? 라는 상식적인 생각에 기인합니다.

서비스 플랫폼이란 결국 각각의 서비스를 돌릴 수 있는 플랫폼 서비스(PasS)의 역할을 하는 서비스를 말하며 사실 그 자체로 몇 가지의 embed 서비스를 제공하기 때문에 요즘 Cloud 환경에서 말하는 PasS와 SaaS 중간 정도가 맞겠군요.

그럼 서비스 프로토콜을 어떻게 만들어야 할까? 부터 고민을 했습니다.

**1.3 Framework vs Service**

서비스 프로토콜에 대해서 언급하지 전에 먼저 프레임워크와 서비의 차이점에 대해서 간단하 언급하고 넘어가겠습니다.

프로그래밍에 있어서 성배가 있다면 그것은 재사용입니다. 어쩌면 재사용은 앞서 말한 의자 뺐기 게임의 거의 유일한 해결책입니다.

프로그램이라는 분야가 처음 생겼을 때부터 재사용은 아주 중요한 문제였고 이를 함수 라이브러리 혹은 모듈 그리고 더 나아가 API 형태로 호출하는 초보적인 형태의 재사용의 이슈가 생겨납니다.

이후 단순히 코드 블럭의 재사용을 뛰어넘어 과정의 재사용 – 즉 일련의 과정을 추상적으로 정의하여 과정을 재사용 할 수 있게 하는 프레임워크로의 형태로 진화하게 됩니다. 그전에 컴포넌트 형태로서의 재사용은 이러한 프레임워크의 과정의 재사용에 영향을 미쳤죠.

이러한 범용 프레임워크는 웹 서비스의 발달과 더불어 그 영향력을 널리 떨쳐 잘 알려진 Spring, Strut, Hibernate, Seam 등의 탄생과 더불어 지금도 수많은 프레임워크들이 나타나고 사라지고 있습니다. 프레임워크는 좋은 점도 많지만 난립한 프레임워크로 과도한 복잡성, Learn Curve의 증가 등이 단점이 꼽힙니다.

서비스는 프레임워크와 달리 한 개 이상의 컴포넌트의 집합으로 특정 영역에 대한 완전한 솔류션을 제공합니다. 예를 들어 잡 스케쥴러, Repository Service, 프로토콜 핸들러 등이 있다. 특정 영역에 대한 검증된 완전한 솔류션을 제공함으로서

* 시기 적절한 애플리케이션 개발(Timely Application Development)
* 소프트웨어 품질 향상(Software Quality Improvement)
* 상호 운용(Interoperability)
* 짧은 학습기간(Shorter Learning Curve)

등의 상대적인 장점을 가질 수 있습니다.

**1.3 SOAP vs Restful Service**

이러한 Service의 프로토콜인 SOAP와 Restful 방식을 비교해 보죠

**SOAP**

서비스의 중복 문제에 대하여 이전에 Soap 웹서비스가 그러한 것을 해줄 것이라고 기대한적이 있었고 또 일부의 사람들은 아직 충분히 성숙하지 못했을 뿐이라며 그러한 기대를 아직 가지고 있습니다.

그러나 저는 웹서비스가 일반화 되지 못한 것은 기술의 성숙의 문제와 관계가 없다고 생각합니다.

SOAP 웹 서비스의 가장 큰 단점은 복잡성입니다. SOAP 자체가 복잡성이 높다고 보기는 어려울 수도 있지만 SOAP 헤더, 바디, Fault등의 envelope 방법자체도 그렇고 보안, 트랜잭션 등 부수적인 문제해결을 위한 다양한 표준은 사실 이해하기 쉬운 부분도 아닙니다. (여기서 이야기하는 복잡도는 물론 다른 방법에 비하여 상대적인 관점입니다.)

한가지 또 단점은 상대적으로 무겁다는 겁니다. 이것은 통신기술에 의존하지 않는 표준이라는 특성에 기인하는데 시시콜콜한 것이 너무 많아서 보통의 경우 도구가 없이는 개발하기가 꽤나 험난합니다.

물론 Soap가 가지는 표준의 힘은 마치 국회에서 새로운 법을 지정하면 그 법에 의해 시스템이 돌아가는 것처럼 표준이 있고 그 표준을 지키면 그 서비스는 표준에서 이야기하는 것들이(상호운영, 보안, 라우팅,...) 시스템에서 돌아갈 거라는 확신을 가질 수 있다는데 장점이 있습니다.

즉 내가 SOAP 표준을 지켜 서비스를 구현하면 기술에 대한 세세한 동작 메커니즘을 설명하지 않더라도 전혀 다른 플랫폼이나 언어로 그 서비스를 사용할 수 있을 것이라는 기대를 가질 수 있습니다. (물론 현실적으로 이게 100% 맞는 이야기인지는 SOAP으로 프로젝트를 해 본 사람이라면 의문을 던질 것이지만...)

**RESTful**

그래서 저는 서비스 프로토콜로 SOAP 웹 서비스보다는 RESTful을 생각하게 되었습니다.

RESTful 웹 서비스의 강점은 단순함과 간결함입니다. 잘 구현된 RESTful 웹 서비스는 간결하고 일관된 인터페이스(Uniform Interface)구조를 지니고 있어 어려움 없이 쉽게 사용이 가능합니다.

물론 RESTful은 보안, 정책 등에 표준이 없기 때문에 이런 것까지 고려해서 추가 구현을 해야 한다든가 HTTP Protocol에 의존적이라는 등의 단점이 있습니다. 그러나 REST의 가장 큰 장점인 간결함의 힘도 무시할 수 없습니다.

단순함 좀더 정확히 말해서 간결함은 직관적이며 쉽게 개발하거나 사용할 수 있어 기술의 전파 속도가 굉장히 빠릅니다. 단순함이나 간결함을 가지 기술은 굳이 따로 배우지 않더라도 바로 사용할 수 있는 힘이 있고, 따라서 아키텍처나 설계에 있어 단순함, 간결함을 매우 강조하는 이유가 여기에 있습니다.

물론 프로그래머의 입장에서 본다면 두 가지의 기술 중에 한 가지만 고집하고 사용할 필요는 없습니다. 여력이 되는 데로 모두 배우고 상황에 맞춰 선택하여 사용해도 상관이 없겠지만 여러 웹 사이트에서 자료를 수집해 보니 역시 단순한 기술의 전파력이 무섭다는 것을 알 수 있었습니다.

\* 아마존은 SOAP과 REST 서비스 모두 제공하는데 85% 정도가 REST 서비스를 사용하는 것으로 알려졌다. (2003년)

\* Google은 SOAP 기술에 대해 그리 좋게 보지 않고 있으며 더 이상 SOAP API에 대한 지원을 하지 않는다. (http://radar.oreilly.com/archives/2006/12/google-depreciates-SOAP-API.html)

\* 야후는 현재까지 SOAP API를 제공할 계획이 없다. (http://developer.yahoo.com/faq/#soap)

결국 저는 ARadon이 REST방식의 Service Protocol을 사용하기로 하였습니다.

1. **목 적**

그럼 다시 방법의 구현문제를 잠시 떠나서 왜 이러한 것이 필요할까에 대해 예를 들어보겠습니다.

**2.1 개발자의 문제**

예컨대 여러분이 작은 게시판을 만든다고 가정해보면 해야 할 기능은 그냥 게시판이라고 하지만 실제 기능을 나누어 보면 몇 가지의 다른 서비스로 나누어져 있습니다.

일련번호를 따기 위한 sequence 서비스

등록일을 알아내기 위한 Calendar 서비스

제목과 내용 등을 등록하기 위한 repository 서비스

이진 파일을 등록한다면 binary-repository 서비스

추가적으로 인증과 권한 서비스 등등의 작은 서비스 등의 결합으로 되어 있습니다.

그리고 이러한 작은 서비스는 다른 서비스를 만들더라도 비슷하게 사용될 가능성이 무척 높습니다.

만약 이러한 작은 서비스를 충분히 일반적으로 만들 수 있다면 그리고 다른 서비스의 하부구조(subsystem)서비스로 그 사용이 쉽다면 좀더 복잡한 서비스를 만들 때 지금처럼 소스 카피나 Jar 카피 이상의 효과를 가지리라고 생각했습니다. 또 커다란 볼륨의 서비스보다는 자신의 할 일을 하는 작은 서비스에 집중함으로써 시스템 디자인 시간을 크게 줄일 수 있으리라 생각합니다.

검증되고 테스트를 거진 작은 서비스를 재사용함으로써 애플리케이션의 품질을 높이고 모듈 유닛을 수월하게 테스트 할 수 있게 되고 이런 하부구조 서비스를 사용하는 상위 서비스간의 상호 운용성도 증가할 수 있습니다.

서비스의 프로토콜이 간결하다면 그리고 해당 영역에서 완전한 솔류션을 제공한다면 개발자는 정해진 서비스 집합에 대해 한번 이해하고 나면 매우 빠르게 상위 서비스를 개발할 수 있습니다. 프로토콜은 기존의 잘 알려진 HTTP 프로토콜을 사용해 만들어졌기 때문에 친숙하고 편하게 사용할 수 있습니다.

최종적으로 이렇게 구축된 서비스는 다른 서비스에 묶여서 각기 다른 서비스가 아우러져 새로운 메타서비스를 쉽게 만들어 낼 수 있는 서비스 생태계 시스템으로의 진화도 생각해 볼 수 있게 됩니다.

여기서 서비스를 일반적으로 만들기란 사실 앞서 말한 것처럼 쉬운 것은 아닙니다. (쉬웠다면 이미 누군가가 개발을 했겠죠.)

예컨대 만약 게시판 서비스 같은걸 만든다고 한다면 사용하는 형태에 따라 파일은 0개 혹은 1개 혹은 그 이상을 등록해야 할 수도 있고 어떤 게시판은 인증을 사용할 수도 혹은 안 할 수도 있기 때문에 사용자 정보를 관리하기도 하고 안 하기도 하고 제목과 내용 그 이상의 추가적인 정보를 관리해야 할 때도 있습니다. HTML규약과 CSS규약을 따르면 좀더 쉽다고는 하지만 화면 템플릿도 매번 바뀌며 등록하고자 하는 데이터가 얽히는 순간 이 작업도 매번 반복이 됩니다.

게시판 서비스를 좀더 낮춰서 이진파일 업로드 서비스 하나만 고려해봐도

첫째 파일 사이즈를 제한시킬 수 있어야 하며 동시에 DDos 공격에 대비할 수 있어야 한다.

둘째 업로드한 파일이름이 겹치는 경우를 생각해야 한다.

셋째 업로드한 파일이름이 아스키가 아닌 경우를 생각해야 한다. 이를테면 서버의 OS는 EUC-KR인데 거기에 일본어 이름의 파일을 올릴 수 있을까?

넷째 업로드한 파일의 총 사이즈가 1Tera가 넘을 경우 용량의 문제로 하나의 컴퓨터에 저장하기 어렵다

등등의 고려해야 할 수십가지를 쉽게 상상 할 수 있으며 이러한 다양한 상황에 “일반적인 솔류션”이 과연 가능한가를 다시 한번 고민하게 됩니다.

아이러니 한 것은 여러분이 하나의 서비스에 대해 더 많은 것을 고려할수록 일정과 비용이 늘어나며 그 복잡성으로 오히려 재활용도가 떨어지는 역전현상이 일어납니다.

사실 비지니스 서비스 개발자는 이런 Core System 서비스에 대해 그렇게 많은 시간을 할애할 수 없고 매번 그러한 것을 신경 써야 한다는 것은 피곤한 일입니다.

이진 파일은 그렇다고 치고 어떤 서비스는 제목과 내용을 저장하고 어떤 서비스는 제목과 내용과 태그를 저장한다고 해보죠. 이 서비스는 재사용이 될 수 있을까요? 매번 비슷한 테이블과 비슷한 쿼리를 짜는 일에 지쳤다면 이제 그런 일을 "일반적으로" 할 수 있는 Repository 서비스를 만드는걸 생각해 봐야겠죠. (물론 코드 복사를 통한 재활용도 재활용이라면 재활용이라고 할 수는 있겠지만 그 경우 다시 테스트부터 많은 작업이 다시 요구됩니다. )

**2.2 고객의 문제**

지금까지 개발자의 문제 즉 개발상의 생산성에 대한 문제에 대해서만 얘기했습니다.

그러나 냉정하게 들릴지 모르지만 고객은 이런 문제에 관심이 없습니다. 고객의 입장에서 중요한 점은 무엇일까요?

개발자를 채찍질해서 개발자가 밤을 매번 꼴딱 세우든 고객입장에서 그건 별로 관심이 없습니다.

다만 고객이 관심 있어야 할 문제는 그래 올라갔다 치자. 해당 컨텐트를 다른 서비스에서 쉽게 이용할 수 있는가? 에 대한 문제입니다.

제가 오랫동안 개발자로서 생활하면서 느낀 점은 고객은 그렇게 생각하지 않지만 결코 소프트웨어는 소프트 하지 않는다는 사실입니다. 여기에 이 필드 하나만 추가해 주세요 라는 요구사항이 고작 몇 분 많아야 몇 시간이면 될 것 같은데 왜 개발자가 난감해 하고 몇 개 이것 저것 고쳐달라고 했더니 심각한 얼굴로 그럼 전면 재개발해야 한다는 황당한 소리를 하는지 이해할 수 없어 하는 경우를 많이 봅니다.

네. 저도 이해합니다. 개발자를 떠나서 상식적으로 본다면 이것 저것 조금 바꾸는 게 그렇게 많은 비용이 들어서는 안됩니다.

이건 프로그램이 얼마나 구체적이었는가에 달린 문제입니다.

예컨대 우리는 보통 내일 오후 2시에 시청 앞 광장에서 보자 라고 약속합니다.

그러나 이걸 프로그램으로 구현하면 2010년 10월 28일 14:01:01.234초에 37도 33'58.87"N 126도 58'40.63"E 에서 본청 첫 번째 출입문을 기준으로 45도 각도로 12.3미터 떨어진 곳에서 보자. 라고 구현됩니다. 어떤 프로그램은 거기서 시선은 N45도 상방 93도 각도로 옷은 파란색 남방에 왼손 두번째 손가락은 아래로 향하게 해라 라는 조건을 주는 경우도 있습니다.

그러다 문득 아 내일 오후에 말고 저녁쯤에 회사 앞에서 보는 게 낫겠다 라고 하는 순간 해당 프로그램은 대응하기 어렵게 됩니다. 농담 삼아 하는 웃기는 소리가 아니라 실제로 그럽니다.

사람은 내일 오후 두시 시청 앞 이라는 추상성을 익숙하게 사용하지만 이것이 프로그램화 되면 추상화된 말은 아주 아주 구체적인 무언가가 됩니다. 그래서 프로그램이 좀더 일반적인 서비스에 기반해서 작성 될 수 있다면 이런 변동에 좀더 쉽게 응대를 할 수 있습니다.

예를 들어 ARadon의 Repository 서비스는 필드의 수나 타입에 크게 구애 받지 않습니다. 즉 어느날 A라는 필드를 새로 저장해야 겠다고 하더라도 새로 Repository를 재 설계해야 할 필요가 없습니다. 혹자는 이렇게 말할 수도 있습니다.

아니 그럼 애초에 처음부터 그렇게 짜면 되지 않냐고? 아쉽지만 그렇게 프로그램을 짜는 것은 무척 어려운 기법에 속합니다. (DB를 만드는 것과 비슷한 노력이 들어갑니다. ) 처음에 말한 바와 같이 이렇게 처음부터 개발하기 위해 고급 개발자와 충분한 일정이 필요하며 이들이 의자를 차지하면 적당한 가격은 의자에 앉을 수가 없습니다.

문제는 초기에 그러한 비용을 감내 할만큼 이 서비스의 해당 기능에 대한 확신이 있냐는 문제가 있습니다. 고객의 입장에서 볼 때 이 프로젝트가 성공할지 어떨지 모르는 상태에서 많은 비용을 추가 하기 곤란한 경우가 많습니다. 그래서 처음에는 비교적 빠른 시간에 저렴한 비용으로 개발하기를 원하죠. 그래서 주먹구구식이지만 적은 비용으로 그럴듯하게 동작하는 서비스를 만들기를 원합니다.

그러나 운이 좋아 그 프로젝트가 아주 성공했다고 하죠. 기뻐할 일이 아닙니다. 더 심각한 문제가 생기니까요. 서비스가 너무 인기를 끌어서 서비스는 여기저기서 빵꾸가 나고 있고 그걸 때우기 위해 사용한 주먹구구식 해결책으로 이미 서비스는 너덜너덜한 상태가 되었습니다. 그나마 그 너덜해진 서비스도 이제는 사용자가 많아짐에 따라 분산을 고려해야 합니다.

개발자는 다시 말합니다. 전면 재개발해야 한다고. 그러나 분산은 매우 어려운 프로그래밍 기법이고 분산이 아닌 기존 서비스를 분산으로 만드는 것은 애초에 분산서비스로 만드는 것보다 더 어렵습니다. 추가적으로 이전 제공되는 서비스와의 호환성도 매우 어려워지는 문제가 됩니다. 당장 사용자가 로그인되어 있다는 상태(즉 세션)를 어떻게 관리 해야 하는가 하는 기초적인 문제부터 고민을 해야 합니다.

ARadon은 이에 대응하여 상위 서비스가 공통적으로 사용하는 하부 구조 서비스를 제공합니다. 따라서 상위 서비스 구현자는 제공되는 하부 구조 서비스를 활용하여 더 빨리 서비스를 런칭할 수 있게 됩니다. 게다가 그 서비스는 추상적으로 완전한 솔류션 형태로 제공되기 때문에 구현자가 이것이 어떻게 작동하는가에 대해 깊게 고민하지 않고 상위 시스템 서비스와 상관없이 하부구조 서비스는 분산화 될 수 있습니다.

**2.3 시장의 문제**

현재 시장의 Fact를 먼저 보죠.

* 구글과 MS 그리고 IBM, 아마존등 소프트웨어 회사에서 클라우드 서비스를 한다.
* 스마트 폰 그리고 패드 등의 새로운 플랫폼이 뜨고 있다.
* 구글은 크롬이라는 브라우저를 내놓았고 이는 앞으로 제공되는 서비스의 클라이언트 플랫폼이 되기를 기대하고 있다.
* 향상된 네트워크 그리고 무선 망이 조밀해지며 확대되고 있다.
* 저가 PC와 UMPC 노트북 업체들이 다시 나타났다.
* 서비스를 패키지 형태가 아닌 앱 서비스형태의 초기 서비스 방식이 나타나고 있다.

네. 위의 Fact를 굳이 말하지 않아도 최근 소프트업계의 화두는 단연 Cloud 입니다. 앞으로는 구글, MS, 아마존, Sun, IBM 등의 몇가지 Cloud 서비스 업체가 모든 소프트웨어를 서비스 할 것이라는 약간 섣부른 기대마저 나오고 있습니다.

Saas는 21세기로 넘어가는 시점부터 몇년간 크게 이슈화 되었지만 실제로는 아시다시피 찻잔 속의 태풍에 그쳤습니다. 인프라가 충분히 성숙하지 못했기 때문입니다. 그러나 이제 상황이 변했습니다. 현재 시장의 Fact는 예전과 비교해서 훨씬 더 명백하고 Cloud를 지향하고 있습니다.

정말 세상에 5개의 컴퓨터만이 남을지는 모르겠지만 소프트웨어가 이제는 서비스 형태로 구현되어야 한다는 사실은 점차 일반화 되어 가고 있습니다.

1. **구현 방법/기술**

ARadon은 Core Subsystem 서비스를 제공하기 위해 다양한 다른 서비스를 연계하는 방법을 사용하고 있습니다.

Content Repository를 위해서는 Document Base DB인 MongoDB(www.**mongodb**.org/)를 사용하고 있고, 파일 Repository는 Local 파일과 Hadoop을 스위칭 할수 있는 모델 형태로 개발할 예정입니다. (상위 서비스는 MongoDB가 C++로 구현되었는지 혹은 파일 서비스의 스위칭를 하는지와 상관없이 서비스 이용이 가능합니다. )

Repository 분산에 있어서는 Mongo의 분산화와 Hadoop의 분산 서비스 기능을 추상화 시켜 제공하는 것을 목표로 하며 Process 분산에 있어서는 현재로서는 Xen을 활용하는 것을 생각하고 있습니다. 역시 Core SubSystem Service가 이러한 것을 추상화시켜 대응시켜 주기 때문에 상위 서비스 즉 비즈니스 서비스는 이것에 대해 신경을 쓰지 않고도 단순히 HTTP 프로토콜을 이용하여 분산의 장점을 사용할 수 있습니다.

사용자는 이러한 Core subsystem service를 활용하여 자신만의 새로운 Service를 Plug in 형태로 정의 할 수도 있습니다. 서비스를 이용하는 쪽과 마찬가지로 새로운 User Service를 구현하는 쪽도 고스란히 분산의 장점과 하부구조의 재사용이라는 장점을 가지게 됩니다. 추가적으로 ARadon이 가지는 프로토콜의 독립성으로 인해 같은 서비스를 현재의 태블릿이나 앱 형태의 무선 서비스에서도 재활용이 가능해집니다.

(ARadon은 3가지 형태의 개발자가 필요합니다. ARadon을 만드는 사람, ARadon을 호출해서 실제 사용자가 사용하는 서비스를 만드는 사람, ARadon의 Core 서비스를 활용하여 상위 서비스를 만드는 사람)

1. **실행방법 :**

1 현재 디렉토리에 다운로드 받은 aradon\_fat.zip의 압축을 푼다.(./aradon\_fat)

2 ./aradon\_fat 디렉토리로 이동하여 samplesite.zip를 현재디렉토리에 압축을 푼다.(./aradon\_fat/samplesite

3 ./aradon\_fat 디렉토리의 start.bat을 실행하여 ARadon을 기동한다. (기본으로 9002번 포트와 27017, 28017 포트를 사용한다. )

4 ./aradon\_fat/samplesite/ 디렉토리의 start.bat을 실행시켜 예제 사용자 서비스를 기동한다. (기본으로 8095번 포트를 사용한다.)

**4.1 demo1 보기**

http://localhost:8095/simple/aradon/board/board.htm 으로 접속하여

List - Add Article : 글 등록 :

View - Pre List : 이전 페이지로, Edit ARticle : 글 수정, Delete : 글 삭제

Edit - Apply Modify : 글수정, Cancel : 글 목록으로, Delete : 글 삭제

**4.2 demo2 보기**

(HTML5 일부 기능을 사용했기 때문에 파이어폭스 3.6.2 이상의 최신버전으로 접속해야 한다. )

http://localhost:8095/simple/aradon/upload/list.htm 로 접속하여

Add Gallery를 클릭하여 등록화면으로 이동한다.

Memo 란에 간단한 내용을 적고 "drop img files here"로 탐색기의 이미지 파일들을 드래그하여 옮긴다. (이미지 파일이 등록된다.)

(만약 브라우저의 주소가 이미지의 파일 주소로 이동했다면 파이어폭스 3.6.2 이상을 사용하고 있는지 다시 확인한다. 현재는 파일이름은 영문으로 된 이미지 파일만을 사용해야 한다. )

Add Content 글자를 클릭하여 Content를 등록한다.

**4.2 demo3 보기**

http://localhost:8095/simple/aradon/employee/employee.htm 로 접속하여

사원을 몇 명 등록하고 검색여부를 확인한다. (사원을 등록하는데 loreLet이 인덱스를 하는데 indexLet 그리고 search를 하는데 searchLet이 사용되었다. )

데모를 보면 실망스러워 할지도 모르겠지만 내부 호출방식은 ARadon의 서비스 제공 방식을 잘 보여주고 있습니다.

Localhost:8095번은 일반적인 사용자 서비스이며

Localhost:9002 번이 ARadon 서비스입니다. (분산을 기본모델로 하였지만 일단 AJAX의 Cross-Site-Script 문제 등도 있으니 Local에서 사용하는 방법을 기본 데모로 보여주고 있습니다. )

ARadon은 그 자체로 하나의 완전한 서비스 솔류션을 제공하기 때문에 특정 다른 RDB나 WAS등의 도움 없이도 혼자 기동합니다.

8095에 html로 구현한 간단한 사용자 서비스 예제는 단지 HTTP Call만을 통해 동적 서비스를 제공하고 있으며 HTML의 HTTP 콜만으로 ARadon의 서비스를 활용할 수 있으며 이는 달리 말해 클라이언트 서비스 플랫폼의 환경에 영향 받지 않는 다는 뜻입니다.

aradon-config.xml에 보면 상세 서비스 항목을 볼수 있는데 demo1은 sample section의 basic\_board 서비스를 사용하고 있으며

demo2는 demo1의 basic\_board를 재활용하고 있으며 추가적으로 system section의 reference와 file 서비스도 사용하고 있습니다.

사실 demo1의 basic\_board 서비스도 system section의 repository와 sequence, utils 서비스를 재사용한 Wrapping에 불과합니다. Basic\_board 서비스를 제공하는 net.ion.radon.sample.BulletinLet는 내부적으로 system service를 inbound Call 하고 있을 뿐입니다.

Basic\_baord 서비스는 후처리 필터가 달려 있는데 net.ion.bulletin.filter.servant.SystemOutLog 필터는 request header와 context attribute를 System.out에 print 하는 단순한 기능을 하고 있습니다. 이러한 전후 처리 필터를 통해 Security와 Logging 혹은 Monitor 서비스 등을 재 사용할 수 있습니다.

ARadon은 향후 기존의 RDB의 Repository도 같은 방식으로 추상화된 방식으로 제공할 수 있게 Context에 DB 연결정보 등을 설정하여 사용할 수도 있습니다.

Dev-config.xml을 보면 향후 Aradon이 지원해야 할 서비스 형태를 보여주고 있습니다. 사용자는 직접 개발한 Let을 플러그인 형태로 등록하여 사용할 수 있어야 하며 Let과 필터를 바로 config를 통해서도 정의 할 수 있어야 합니다.

Aradon의 서비스는 HTTP Protocol로 활용되기 때문에 GET 방식으로 <http://localhost:9002/system/repository/bulletin.bleujin>로 접속하여 사용자가 등록한 글들을 볼 수 있다.(물론 내부적으로 페이징을 사용하지만 그것에 대해 구체적으로 알 필요는 없다는게 장점입니다.:.)

이와 마찬가지로 다른 모든 Service들도 GET, POST, DELETE, PUT 메소드로 억세스가 가능하며, 보안의 문제는 Sercurity Filter등을 도입하여 해결할 생각입니다.

1. **추후 개발 진행**
   1. **추가 제공되어야 할 Common 서비스**

Index / Search Service

Notification Service

Schedular Service

Service Monitoring Service

Content Repository JCR-283 지원

JDBC 연계 서비스

다양한 언어를 통한 Template 서비스

다양한 쿼리 방식 지원

Batch Transaction Request 지원

자체 도움말 등을 위한 Wiki Service

* 1. **추후 제공되어야 할 아키텍쳐**분산 프로세스(Xen)  
     분산 Service Farm

**뱀다리**

앞에서 말한

“일반적으로 소프트웨어는 하나의 레이어를 더함으로써 해결할 수 있지만 그럴 경우 추가적인 문제가 생긴다”

의 휠러가 지적한 또 다른 문제란

레이어를 통한 간접과 계층화는 공간과 시간의 부담을 추가하고 코드의 가독성을 해칠 수 있다는 문제점등을 예로 들 수 있습니다.

보통의 경우에는 공간과 공간상의 추가 부담은 그리 크지 않기 때문에 일반적으로 큰 관심사가 되지 못합니다. 대부분의 경우 추가적인 포인터 참조나 서브루틴 호출에 의한 시간 지연은 전반적인 구조 개선에 비할 때 사소한 수준에 그칩니다. 사실 요즘의 현대적인 프로그래밍 언어들은 추가적인 유연성을 얻기 위한 목적으로 일부 연산들의 경우 항상 하나의 간접 층을 거치도록 하는 경향을 보이고 있습니다.

예를 들어 java나 C#의 경우 객체에 대한 모든 접근이 하나의 포인터 간접을 거치게 하는데, 이는 쓰레기 수거를 위한 것입니다. 또한 Java에서는 인스턴스 Method에 대한 거의 모든 호출이 하나의 조회 테이블을 통해서 분배되는데, 이는 다른 클래스를 상속하는 클래스들이 실행시점에서 Method를 재 정의 할 수 있도록 하기 위한 것입니다. 모든 객체 접근과 Method 호출에 부가되는 이러한 추가부담에도 불구하고 두 플랫폼은 시장에 선전을 펼치고 있다는 것은 이문제가 휠러가 지적한 또다른 문제의 핵심이 되진 않으리라 예상할 수 있습니다.

사실 공간과 시간이 부담이 크게 문제가 되지 않는다면 코드의 가독성은 어떨까요?(즉 유지 보수의 문제와 연결됩니다.) 코드의 가독성에 대한 간접의 영향은 아주 중요한 문제입니다.

지난 50년간 CPU의 속도는 엄청나게 빨라진 반면 코드를 이해하는 사람의 능력은 별로 발전하지 않았다는 점을 감안한다면 충분히 이해할 수 있을 것입니다. 그래서 애자일 프로세스 옹호자들은 오늘이 구체적인 요구가 아니라 미래에 생길 수도 있는 애매하고 명시되지 않는 요구사항들을 처리하기 위해 계층들을 도입할 때에는 불필요한 계층을 넣지 않도록 아주 신중해야 한다고 조언합니다.

스몰더스는 이에 대해 성능 안티 패턴을 논의하면서 계층은 케이크를 위한 것이지 소프트웨어를 위한 것이 아니다 라고 비꼰 바 있습니다.

프레임워크로서의 오픈소스 : 정부지원 프레임워크는 올해 초에 처음 보고 꽤 흥미롭게 공개된 파일들의 사이트 소스들을 보았습니다. 그리고 느낀 점은 의미 있는 시도지만 실제 쓰기에는 다소 무리가 있다는 생각이 들었습니다. 왜냐하면 너무 Detail 하기 때문입니다.

너무 Detail한 소스는 재사용하기에 여러가지 무리가 생깁니다.

이를테면 특정 DB의 SQL로 작성되어서 다른 DB를 사용할때는 다시 작성해야 하거나,

혹은 너무 ANSI 구문으로만 작성되어서 충분히 효율적인 기능을 살리지 못하거나

그 어느 쪽도 문제가 되기 때문입니다.

추가적으로 회사마다 변수나 컬럼명에도 나름의 네이밍 규칙도 있을테고 관리 규칙도 있을 겁니다. Jakarta의 common library처럼 Library형태로도 쓰기 힘들고 그렇다고 spring등의 다른 Framework처럼 충분히 추상적이고 일반적이지 못하다면, 즉 지나치게 디테일 해서 소스를 카피해서 자신의 상황에 맞게 다시 고쳐야 한다면 그건 Framework라기보단 템플릿에 가깝다고 생각합니다.

삼성 SDS나 마이플랫폼 등도 간단히 보았지만 마찬가지 이유로 사용하기 곤란하더군요. 저는 지난 몇 년 동안 회사에서 System Framework를 만들었습니다. DB, Configuration, Message등등 지속적으로 유지 보수하면서 만들면서 느낀 점은 언젠가 들은 “Framework는 정글에 난 길과 같다”라는 말이었습니다.

좋은 프레임워크가 가져야 할 특징은 그 길을 어떻게 만드느냐 보다 그 길의 유연성과 확장성이 더 중요하다는 걸 알게 되었습니다. 정글에 나 있는 길만 따라가서 목적지에 도착할 수 있다면 아주 행복하겠지요. 하지만 그런 Framework라는 건 있을 수 없거나 아주 제한적인 상황에서 일뿐입니다. 보통의 경우 Framework가 상상한 그 길을 벗어난 순간 그 순간부터 Framework를 사용하는 것은 정글 숲을 헤쳐가는 것처럼 악전 고투가 되어 버리니까요.

즉 프레임워크는 특정 영역에 있어서 좀더 쉬운 확장이 가능해야 할 뿐만 아니라 그 특정 영역이 아닌 곳에 있어서는 방해물이 되지 않도록 하는 것이 전자만큼이나 중요하게 추구해야 할 가치라고 생각합니다. 그러다 보니 business framework 보다는 비교적 중립적인 system framework만을 만들게 되더군요.

그러다가 소프트웨어 개발에 있어서의 성배가 결국 DRY(Don’t Repeat Yourself)라면 business Framework 개발에 있어서도 무언가 다른 시도를 해야 하지 않을까 생각하게 되었습니다.

제가 생각한 바는 더 추상적이 되자 라는 거였고 함수레벨의 중복을 막는 라이브러리, 중간형태의 컴포넌트, 그리고 구조적 중복을 막는 Framework로 충분하지 못하다면

서비스 레벨의 웹서비스를 생각하게 되었고 UNIX의 명령어처럼 하나를 잘하는 서비스들을 만들어서 그것들을 Chain 형태로 연결시키자 라는 생각을 하게 되었습니다.

언어적 관점에서 본다면 저급언어인 기계어나 어셈블리로부터 3GL이라고 불리는 C나 Java가 있지만 이보다 더 추상적인 언어가 필요했습니다.

일반적으로 고차원 언어가 제공하는 장점은 많습니다.

1.     생산성이 높아진다. 가끔 놀랄 만큼 높아지기도 한다.

2.     고차원 언어는 오류가 발생할 기회를 제거하므로 전반적인 구현 수준을 높인다. 예를 들어, 어셈블러 프로그래머는 하드웨어 레지스터를 조작해야 하므로 레지스터 할당 오류를 저지를 위험이 있다. 고차원 언어 프로그래머는 레지스터 오류를 저지를 가능성이 없다. 고차원 언어에서는 레지스터를 참조하지도 못하고 참조할 필요도 없기 때문이다.

3.     고차원 해법은 코드 행수가 적다, 따라서 오류가 발생할 기회도 더 적다.

4.     고차원 언어로는 구조적 코드를 작성하기 쉽다. 흥미롭게도, 최고 고차원 해법은 코드가 구조적일 필요도 없다.

5.     고차원 언어로 작성한 코드는 이식성이 더 놓다.

6.     고차원 언어로 작성한 코드는 유지 보수성이 더 높다.

7.     고차원 언어로 작성한 코드는 테스트가 더 용이하다.

그래서 몇 년전에 스크립트성 언어를 직업 만들어 보기도 했지만 언어 구조 자체가 튼튼하지 못했기 때문에 추상적인 언어가 가지는 장점을 충분히 활용하지 못했고 Learning Curve등의 다른 문제가 생겼습니다.

그래서 언어라는 것이 단순히 일련의 절차를 기술하는 것이라고 정의할 수 있다면 HTTP의 URI와 HTTP의 Method도 충분히 언어의 역할을 할 수 있지 않을까 생각했습니다.

그래서 구조적으로 웹 서비스(REST) 형태를 언어적으로는 HTTP 프로토콜 그 자체를 사용하여 서비스를 정의할 수 있게 해보자 라는게 ARadon의 시초입니다.

만약 휠러가 지적한 “문제를 해결기 위해 도입한 하나의 레이어를 더했을 때 생기는 추가적인 문제”만 해결할 수 있다면 이것이야 말로 재사용이라는 성배에 조금 더 가까이 갈수 있지 않을까 해서요. 성능에 있어서의 분산과 별도로 최근에 저는 ARadon에 멀티 HTTP Request를 하나의 Request로 처리해주는 로직을 만들고 있습니다. 즉 여러 개의 서비스를 묶어서 한번의 Request로 호출하는 기능입니다. 이 기능이 구현되면 REST로 구현된 서비스를 마치 RPC를 하듯이 호출해서 사용할 수 있을거라 기대합니다. (더불어 Transaction의 문제도 상당부분 해결이 가능합니다. )

만약 정부지원 프레임워크와의 관련성에 대해서 묻는다면 ARadon은 너무 Detail 하지 않으면서 DRY라는 성배를 추구하는 방법에 있어서 하나의 대안입니다. ARadon(Abstract Radon)에 올라가는 ServiceLet은 충분히 추상적이기 때문에 각각의 Let을 매번 수정하지 않는 상태에서 충분히 구체적인 서비스에 사용될 수 있습니다.

ARadon에서 모든 Content(Node라고 불리는)는 전세계에서 유니크한 UID를 부여 받고 ARadon이 충분히 추상적이기 때문에 다른 서비스를 하는 ARadon과의 정보교환도 쉽게 가능합니다. 만약 ARadon이 제 기대대로 충분한 성능과 확장성을 보장하는 형태대로 Platform 서비스를 할 수 있다면 ARadon 플랫폼 위에서 돌아가는 ServiceLet을 누가 만들든 누구나 Uniform한 형태로 이용할 수 있기 때문에 한가지를 잘하는 서비스 들을 묶어서 좀더 쉽게 구체적인 서비스와 메쉬업 서비스를 만들 수 있을 거라 생각합니다.