

# RPC 프레임워크 제작 삽질기

PyCon APAC 2016

홍민희 ([hongminhee.org](http://hongminhee.org))

스포카 ([spoqa.com](http://spoqa.com))

# 배경

- 사업 규모가 커지고 제품의 양과 복잡도가 늘어나는 상황.
- 자연스레 복잡도를 잡기 위해  
기능 덩어리를 떼서 별도 서비스로 배포하기 시작.
- 배포 단위를 나눌 수 있다.
- 작은 팀 여럿이 각각의 서비스를 독립적으로 만들 수 있다.
-  전통적인 용어로는 서비스 지향 아키텍처(SOA)라고 하기도.

# 서비스 지향 아키텍처 (SOA)

- 저도 잘 모릅니다.
- 전 직장에서도 경험해보긴 했으나, 입사 당시 이미 체계가 닦여 있었음.
- 서비스 지향 아키텍처를 도입해보는 경험은 처음.

# 기존에 익숙한 방법

```
from datetime import date
from coupon import create_birthday_coupons
from messaging import send_message

template = '''{coupon.customer.name} 님 생일 축하합니다.  
선물로 생일 쿠폰을 드립니다.  
{coupon.code}'''  
  
today = date.today()
coupons = create_birthday_coupons(today)
for coupon in coupons:
    message = template.format(coupon=coupon)
    result = send_message(coupon.customer.phone_number,
                          message)
```

# 원격 프로시저 호출 (RPC)

```
from datetime import date
import json
from urllib.request import urlopen
template = '''{coupon[customer][name]} 님 생일 축하합니다.
선물로 생일 쿠폰을 드립니다.
{coupon[code]}'''
today = date.today()
coupons = json.load(
    urlopen(
        'https://coupon-service/coupons/birthday/',
        data=json.dumps({'date': today.isoformat()})
    )
)
for coupon in coupons:
    phone_number = coupon['customer']['phone_number']
    result = urlopen(
        'https://messaging-service/messages/',
        data=json.dumps({
            'phone_number': phone_number,
            'message': template.format(coupon=coupon),
        })
    ).code
```

# 기존에 익숙한 방법

```
from datetime import date
from coupon import create_birthday_coupons
from messaging import send_message

template = '''{coupon.customer.name} 님 생일 축하합니다.  
선물로 생일 쿠폰을 드립니다.  
{coupon.code}'''  
  
today = date.today()
coupons = create_birthday_coupons(today)
for coupon in coupons:
    message = template.format(coupon=coupon)
    result = send_message(coupon.customer.phone_number,
                          message)
```

# 원격 프로시저 호출 (RPC)

```
from datetime import date
import json
from urllib.request import urlopen
template = '''{coupon[customer][name]} 님 생일 축하합니다.
선물로 생일 쿠폰을 드립니다.
{coupon[code]}'''
today = date.today()
coupons = json.load(
    urlopen(
        'https://coupon-service/coupons/birthday/',
        data=json.dumps({'date': today.isoformat()})
    )
)
for coupon in coupons:
    phone_number = coupon['customer']['phone_number']
    result = urlopen(
        'https://messaging-service/messages/',
        data=json.dumps({
            'phone_number': phone_number,
            'message': template.format(coupon=coupon),
        })
    ).code
```

# 파이썬 모듈 쓰듯 서비스를 쓸 수 없을까

- 아주 옛날부터 사람들이 고민해온 문제.
- **RPC** 프레임워크라고 한다.
- SOAP + WSDL
- Thrift
- Protocol Buffers
- Cap'n Proto
- 제가 한번 써보겠습니다.

# Thrift

- 페이스북에서 개발.
- Java, C++, PHP, Erlang, Python 등을 지원.  
(하지만 어쩐지 Java/C++만 잘 되어 있다.)
- 스포카의 엔티티 모델은 서브타입 다형성을 굉장히 많이 사용하는데,  
이에 대응하는 도구가 없다.

# Cap'n Proto

- Protocol Buffers 저자가 두번째로 만든 작품.
- RPC 프레임워크의 C++ 느낌.
- 놀라운 기능이 굉장히 많다. 추천!
  - 메모리 복사 없이 직렬화/직렬화 해제
  - Union
  - Generics
  - Promise pipelining
- C++/Rust/Go/JavaScript/Python 지원.

# Cap'n Proto

- (여기는 파이콘이지만) Java 바인딩이 RPC 지원 없이 직렬화만 지원.
- 모든 바인딩이 C++로 구현된 Cap'n Proto 런타임 라이브러리를 C FFI로 붙여서 구현함.
- 메모리 복사 없는 직렬화는 Cap'n Proto의 최고 세일즈 포인트.
- ...
- ...
- ...
- ...  
...
- ...  
...

# Cap'n Proto

- (여기는 파이콘이지만) Java 바인딩이 RPC 지원 없이 직렬화만 지원.
- 모든 바인딩이 C++로 구현된 Cap'n Proto 런타임 라이브러리를 C FFI로 붙여서 구현함.
- 메모리 복사 없는 직렬화는 Cap'n Proto의 최고 세일즈 포인트.
- 그런데 문제는 C/C++/Rust 정도가 아니면 아무래도 상관 없다는 것.
- 메모리 복사 비용이 문제인데 파이썬을 쓸까요?
- ...
- ...  
...
- ...  
...

# Cap'n Proto

- (여기는 파이콘이지만) Java 바인딩이 RPC 지원 없이 직렬화만 지원.
- 모든 바인딩이 C++로 구현된 Cap'n Proto 런타임 라이브러리를 C FFI로 붙여서 구현함.
- **메모리 복사 없는 직렬화는 Cap'n Proto의 최고 세일즈 포인트.**
- 그런데 문제는 C/C++/Rust 정도가 아니면 아무래도 상관 없다는 것.
- 메모리 복사 비용이 문제인데 파이썬을 쓸까요?
- 반면 **메모리 복사 없는 직렬화를 위한 구현 전략이** 바인딩의 인터페이스 설계에 중요한 영향을 주었다.
- 파이썬에서는 필요 없지만 Cap'n Proto의 메모리 복사 없는 직렬화를 구현하는데 필요했던 인터페이스/개념들과 씨름해야 한다.

# Cap'n Proto

가장 중요한 결함: 바인딩이 파이썬 객체 모델을 무시한다.

- Cap'n Proto 자료형이 파이썬의 `type` 객체가 아님.
- 필드에 대해 `getattr()` / `setattr()` / `hasattr()` 작동 안함.
- 필드가 비어있거나 없을 경우 `AttributeError` 가 아닌 C FFI 아래쪽서 올라온 Cap'n Proto의 오류가 튀어나옴.
- 객체 없이 딕셔너리로만 프로그래밍하는 기분.
- 사실상 다른 언어를 쓰는 경험.

# Cap'n Proto 비추인가요?

아닙니다. 그래도 기능도 많고 장점도 많습니다.

하지만 스포카는 C++로 작성해야 하는 고성능의 게임 서버를 만드는 회사가 아니었기에, 비즈니스와 잘 맞지 않았습니다.

...

# Cap'n Proto 비추인가요?

아닙니다. 그래도 기능도 많고 장점도 많습니다.

하지만 스포카는 C++로 작성해야 하는 고성능의 게임 서버를 만드는 회사가 아니었기에, 비즈니스와 잘 맞지 않았습니다.

다만, 파이썬에서 쓴다면 비추입니다.

# 작년 초의 결정

- 무식하지만 RFC 프레임워크 없이 RESTful API로 통신하자!
- 하지만 구현 시간의 대부분을 다음과 같은 껍데기 작업에 쓰게 됐다.
  - HTTP 요청 해석.
  - JSON 해석.
  - 유효성 검사. (시각인데 RFC 3339 형식이 아니라거나...)
  - 유효하지 않을 경우, **적절한 오류로 응답**.
  - 유효할 경우, 애플리케이션 내부의 적절한 자료형으로 번역.  
시각이면 `datetime.datetime` 으로, 사용자면 `User` 로...

# 작년 가을의 마음

- 나는 내가 근면하다고 착각했다.
- 그래도 반복되는 일반적인 문제들을 해결해볼 수 있지 않을까?
- 혼자서 만들어보자.
- 스포카에서 RPC 프레임워크에 필요한 것이 무엇일까?

## 스포카의 경우

- 도도 포인트 등 주로 오프라인 상점을 위한 서비스들.
- 페이스북이나 라인, 카카오 같이 소비자 제품이 아니고, 따라서 규모도 상대적으로 훨씬 작다.
- 사업의 성장에 비례해서 제품이 빠르게 복잡해지긴 하지만, 처리하는 통신량이나 계산량이 빠르게 폭발하지는 않는다.
- 따라서 스포카에서는 Cap'n Proto 같은 성능을 위한 설계보다는 애플리케이션 구현을 간편하게 할 수 있는 설계가 더 유용하다.

# 기본 원칙

- 애플리케이션 개발을 빠르고 쉽게 해주는 것이 목표이다.
- 필요한 기능이 다 완성되지 않아도 써볼 수 있게 하자.
- 성능이 중요하다면 당분간 RPC 프레임워크를 안 쓰고 해결하자.

# 기본 원칙

- 애플리케이션 개발을 빠르고 쉽게 해주는 것이 목표이다.
- 필요한 기능이 다 완성되지 않아도 써볼 수 있게 하자.  중요
- 성능이 중요하다면 당분간 RPC 프레임워크를 안 쓰고 해결하자.

# 모든 것이 완성되지 않아도 써볼 수 있으려면

- 타겟 언어 지원이 불지 않아도 쓸 수 있게 하자.
- 대부분 언어에서 쉽게 쓸 수 있는 JSON을 기반으로 직렬화를 하자.
- 대부분 언어에서 쉽게 쓸 수 있는 HTTP를 기반으로 통신을 하자.

# 스포카에서 많이 쓰는 자료형들

- 길이 제한 없는 정수 (bigint)
- 금액을 표현하기 위한 수 타입 (decimal 등)
- 유니코드 문자열 (반면 바이트열은 많이 안 쓰임)
- UUID
- 날짜/시각
- URI

Thrift나 Cap'n Proto에서 지원 안해서 불편한 적이 많았다.

(그래도 둘 다 유니코드 문자열은 지원합니다.)

# 기본 자료형

다 넣자!

- bigint , decimal , int32 , int64 , float32 , float64
- text , binary
- date , datetime
- bool , uuid , uri
- options ( t? ), lists ( [t] ), sets ( {t} ), maps ( {k: v} )

# 맨날 하는 작업

```
class Money:  
    def __init__(self, amount: Decimal,  
                 currency: Currency):  
        self.amount = amount  
        self.currency = currency  
    def serialize(self) -> Mapping[str, str]:  
        return {  
            'amount': str(self.amount),  
            'currency': str(self.currency),  
        }
```

## 레코드 타입 ( record )

```
record money (
    decimal amount,
    currency currency,
);
```

## 레코드 타입 ( record )

```
{  
  "_type": "money",  
  "amount": "50000",  
  "currency": "krw"  
}
```

1 P

적립됩니다.

전화번호를 입력하시면  
이벤트에 응모됩니다.

QR 카드로 적립



뒤로가기

## 휴대전화 번호 입력

이용약관과 개인정보 취급방침에 동의하시면  
휴대전화 번호 입력 후 아래 적립 버튼을 터치하세요.

010

1

2

3

4

5

6

7

8

9



0

적립

KakaoTalk

입력하신 휴대전화 번호로  
매장의 카카오톡 친구추가  
메시지가 전송됩니다.



dodo

826771

서명/Signature

- 이 카드 한 장으로 모든 도도 제휴매장에서  
멤버십 혜택을 받을 수 있습니다.
- 폰번호나 도도 앱으로도 사용 가능합니다.



주식회사 스포카  
[dodopoint.com](http://dodopoint.com)

고객서비스/제휴문의  
02-544-6463

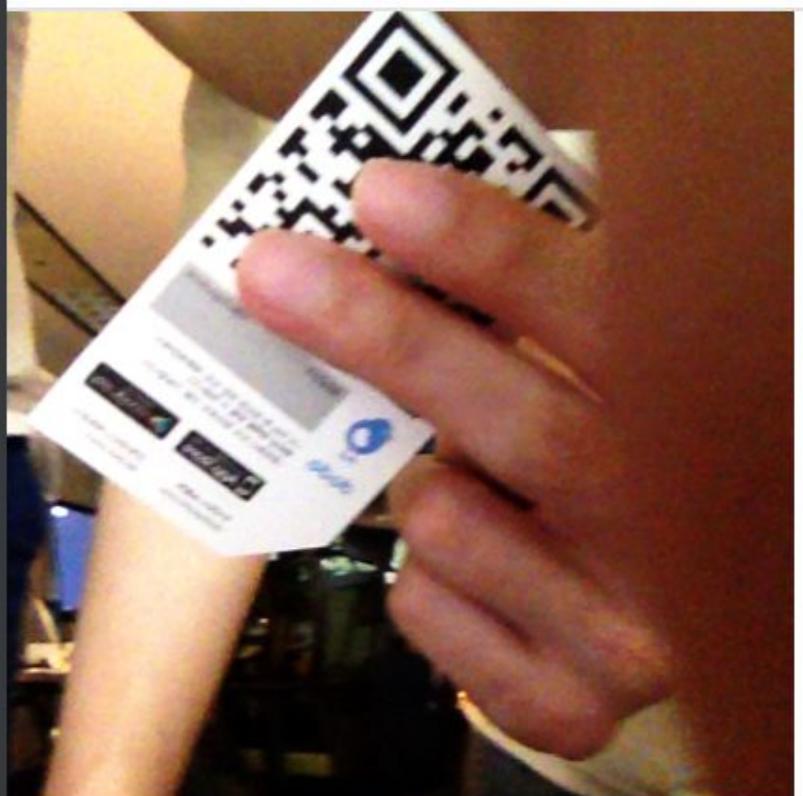
1 P

적립됩니다.

전화번호를 입력하시면  
이벤트에 응모됩니다.

휴대전화 번호로 적립

QR 카드



뒤로가기

KakaoTalk

입력하신 휴대전화 번호로  
매장의 카카오톡 친구추가  
메시지가 전송됩니다.

# 맨날 하는 작업

```
class Identifier:  
    def serialize(self):  
        return {}  
  
class PhoneNumber(Identifier):  
    def __init__(self, phone_number: str):  
        self.phone_number = phone_number  
    def serialize(self):  
        return {'type': 'phone_number',  
                'phone_number': self.phone_number,  
                **super().serialize()}  
  
class QrCard(Identifier):  
    def __init__(self, card_id: uuid.UUID):  
        self.card_id = card_id  
    def serialize(self):  
        return {'type': 'qr_card',  
                'card_id': str(self.card_id),  
                **super().serialize()}
```

## 대수적 자료형 (ADT, union )

```
union identifier = phone-number (text phone-number)
                  | qr-card (uuid card-id)
                  ;
```

# 대수적 자료형 (Algebraic Data Type)

하스켈 ( `data` ), 러스트 ( `enum` ), 스위프트 ( `enum` ) 등 이미 대수적 자료형  
비슷한 것이 제공되는 언어에서는 자연스럽게 번역(는 것을 목표로)합니다.

```
union identifier = phone-number (text phone-number)
                  | qr-card (uuid card-id)
                  ;
```

# 대수적 자료형 (Algebraic Data Type)

하스켈 ( data ), 리스트 ( enum ), 스위프트 ( enum ) 등 이미 대수적 자료형  
비슷한 것이 제공되는 언어에서는 자연스럽게 번역(는 것을 목표로)합니다.

```
data Identifier = PhoneNumber { phoneNumber :: Text }
                | QrCard { uniqueId :: UUID }
deriving (Eq, Ord, Show, Read)
```

# 대수적 자료형 (Algebraic Data Type)

하스켈 ( `data` ), 러스트 ( `enum` ), 스위프트 ( `enum` ) 등 이미 대수적 자료형  
비슷한 것이 제공되는 언어에서는 자연스럽게 번역(는 것을 목표로)합니다.

```
enum Identifier {  
    case PhoneNumber(String)  
    case QrCard(NSUUID)  
}
```

## 대수적 자료형 ( union )

하스켈 ( data ), 러스트 ( enum ), 스위프트 ( enum ) 등 이미 대수적 자료형  
비슷한 것이 제공되는 언어에서는 자연스럽게 번역(는 것을 목표로)합니다.

자바나 파이썬처럼 대수적 자료형이 없는 일반적인 객체 지향 언어에서는 서  
브타입 관계로 번역됩니다.

## 대수적 자료형 ( union )

자바나 파이썬처럼 대수적 자료형이 없는 일반적인 객체 지향 언어에서는 서브타입 관계로 번역됩니다.

```
class Identifier:
```

```
    ...
```

```
class PhoneNumber(Identifier):
```

```
    ...
```

```
class QrCard(Identifier):
```

```
    ...
```

## 대수적 자료형 ( union )

```
{  
    "_type": "identifier",  
    "_tag": "phone_number",  
    "phone_number": "+82 10-1234-5678"  
}
```

## 대수적 자료형 ( union )

```
{  
    "_type": "identifier",  
    "_tag": "qr_card",  
    "card_id": "5d6456ad-d5e3-456f-9d0b-63bf0479b6ea"  
}
```

## 대수적 자료형 ( union )

- 대수적 자료형은 Thrift와 Cap'n Proto를 쓰면서 가장 아쉬웠던 기능
- 스포카에서는 기존 서비스 데이터 모델에서 유난히 서브타입 다형성을 많이 쓰는 편이었기 때문
- RPC 프레임워크를 직접 만드려는 생각이 들게 한 가장 첫 요인

# 맨날 하는 작업

```
class Currency(enum.Enum):  
    krw = 'krw'  
    jpy = 'jpy'  
    usd = 'usd'  
    # ...
```

사실 이렇게 직접 할 필요는 없습니다. ~~pip install iso4217~~

## 열거 타입 ( enum )

```
enum currency = krw
    | jpy
    | usd
    // ...
;
```

## 열거 타입 ( enum )

그런데 열거 타입은 좀더 일반적인 대수적 자료형( union )으로도 달성할 수 있습니다.

```
union currency = krw
| jpy
| usd
// ...
;
```

그럼 왜 열거 타입을 따로 두었을까요?

## 열거 타입 ( enum )

직렬화된 결과가 다르기 때문입니다. 보다시피 단순 문자열로 다뤄집니다.

```
"krw"
```

반면 공용체로 만들 경우 객체가 한겹 생기게 됩니다.

```
{"_type": "currency", "_tag": "krw"}
```

물론 이는 효율만을 위한 기능은 아닙니다. 그보다는, 자유 문자열이었던 필드를 몇 종류로만 한정시키거나, 그 반대의 방향으로 스키마를 리팩토링할 때 하위호환성을 유지하기 위해서 씁니다.

## 가두는 타입 ( boxed )

boxed 타입은 필드가 하나인 record 와 비슷합니다.

```
boxed message (text);  
record message (text body);
```

## 가두는 타입 ( boxed )

하지만 직렬화되면 `boxed` 와 `record` 는 다르게 표현됩니다.

`boxed` 일 경우

```
"A message."
```

`record` 일 경우

```
{"_type": "message", "body": "A message."}
```

# 의도적으로 일반화하지 않은 키워드들

- boxed 와 record 의 관계는 enum 과 union 의 관계와 유사
- 전자는 후자로 일반화 가능
- 하지만 직렬화했을 때의 표현이 달라서 별도 키워드로
- 통신하는 프로그램들을 한번에 배포할 수 없는 경우가 많음
- 하위호환성을 유지하며 스키마를 고쳐나갈 수 있게 하기 위한 방편

## 이름의 양면

또 다른 하위호환 방편. IDL의 모든 이름 정의는 양면을 갖는다.

```
record money (
    decimal amount,
    currency currency,
);
```

...

## 이름의 양면

또 다른 하위호환 방편. 니름 IDL의 모든 이름 정의는 양면을 갖는다.

```
record money/money (
    decimal amount/amount,
    currency currency/currency,
);
```

이름 정의는 사람을 위한 앞면과 직렬화를 위한 뒷면으로 이뤄진다.

# 이름의 양면

이름의 양면은 다를 수도 있고

amount/money

같을 수도 있는데

amount/amount

양면이 같다면 뒷면의 이름을 생략하면 된다.

amount

## 이름의 양면

이름의 뒷면은 하위호환성을 위해 기존의 이름을 유지하는 용도.

```
record money/price (decimal amount/money,  
                    currency currency/type);
```

위 레코드의 값을 직렬화하면 아래처럼 표현된다.

```
{"_type": "price", "money": "9.99", "type": "usd"}
```

## 모듈과 `import`

- IDL 소스 코드는 여러 파일로 이뤄짐.
- 하나의 파일이 하나의 모듈.
- 파이썬 패키지-모듈과 비슷하게 디렉토리로 여러 모듈을 묶을 수 있다.

# 모듈과 import

i18n/currencies.nrm 파일에 정의된 currency 타입 임포트:

```
import i18n.currencies (currency);
```

- 같은 이름의 타입을 중복해서 임포트하면 오류.  
조용히 덮어쓰는 것이 아니라 오류로 중복된 두 항목을 알려줌.
- 임포트 사이에 순환(cycle)이 있을 경우도 오류.  
역시 오류 메세지로 순환 경로를 보여준다:

```
i18n.locales -> i18n.currencies -> i18n.countries  
-> i18n.locales
```

# 서비스 인터페이스 ( service )

한 (마이크로)서비스가 제공하는 기능들을 메서드의 목록으로 명세.

```
service pdf-service (
    blob render-uri (uri uri),
    blob render-html (text html),
);
```

# 서비스 인터페이스 ( service )

서비스는 타겟 언어의 추상 인터페이스로 컴파일된다.

```
class PdfService(Service):
    def render_uri(self, uri: str) -> bytes:
        raise NotImplementedError()
    def render_html(self, html: str) -> bytes:
        raise NotImplementedError()
```

# 서비스 구현

서비스 애플리케이션 구현자는 컴파일된 인터페이스를 구현하고:

```
class PdfServiceImpl(PdfService):
    def render_uri(self, uri: str) -> bytes:
        ...
        return ...
    def render_html(self, html: str) -> bytes:
        ...
        return ...
```

# 서비스 구현

타겟 언어 바인딩이 제공하는 런타임 기능을 써서 서버로 띄운다:

```
from wsgiref.simple_server import make_server
from nirum.rpc import WsgiApp

app = WsgiApp(PdfServiceImpl())
make_server('', 8080, app).serve_forever()
```

파이썬의 경우 `nirum.rpc.WsgiApp`을 통해 서비스를 평범한 WSGI 앱으로 만들 수 있으므로, 평소에 쓰던 WSGI 서버(예: Gunicorn, uWSGI 등)로 배포할 수 있다.

# 서비스 클라이언트

서비스 인터페이스를 통해 클라이언트 구현도 함께 생성된다.

```
from pdf_service import PdfService_Client

pdf_service = PdfService_Client('http://localhost:8080/')
pdf_bytes = pdf_service.render_html('''
<h1>PyCon APAC 2016에 오신 것을 환영합니다!</h1>
''')
with open('pycon.pdf', 'wb') as f:
    f.write(pdf_bytes)
```

## 빌드 결과

- 빌드 결과는 해당 타겟 언어의 패키지 단위.
- 파이썬이라면 `setup.py` 파일을 포함.
- JS라면 `package.json` 파일을 포함.
- **니름 IDL을 고치는 사람이 아니라면,**  
**니름 컴파일러를 설치하지 않아도 되게 하는 것이 목표였기 때문.**
- 최근의 프론트엔드 도구들이 프론트엔드를 전혀 고치지 않는 동료에게도 `node.js`, `npm`, `webpack` 등을 설치하게 하는 것이 불편하다 느꼈고,  
니름은 그런 불편을 피하게 하고 싶었다.

# 스포카 사내 RPC 프레임워크: 니름

- 처음에는 텔레파시(telepathy)를 떠올렸으나  
이미 다른 프로젝트에서 많이 쓰이는 이름이고 검색도 잘 안됨.
- 오픈 소스로 공개하고 싶었기 때문에 검색도 잘 되길 바랐다.
- 그러다가 이영도의 소설 **눈물을 마시는 새**에 나오는 **니름**을 떠올림.
- 텔레파시와 꽤 비슷한 개념이고,  
한국어 소설에 나왔기 때문에 다른 프로젝트에서 쓰인 적도 없음.
- 로마자 표기로는 **nirum** 사용.  
(로마자 표기법으로는 “nireum”이지만 글자수를 줄이고 싶었음.)

✨ 니름 ✨



✨ github.com/spoqa/nirum ✨



# 앞으로의 계획

- 어노테이션
- RESTful HTTP API
- 문서화 도구 (이미 문서화 문법은 있음!)
- 사용자 정의 타입 매팅
- 원격 엔티티 (동기화)
- 타겟 언어 추가 (JavaScript, Swift, Kotlin, Rust)

# 감사합니다.

슬라이드 공유:

[j.mp/pycon-apac-2016-hong](https://j.mp/pycon-apac-2016-hong)

니름 프로젝트:

[github.com/spoqa/nirum](https://github.com/spoqa/nirum)

연사 홈페이지:

[hongminhee.org](https://hongminhee.org)