VSCode gRPC 프로젝트 – 처음부터 끝까지 (업데이트 가이드)

이 문서는 이전 가이드를 보완하여, 실제 진행 중 겪었던 오류와 해결 방법을 포함해 **“딱 맞는 버전”**으로 정리한 단계별 안내서입니다. 각 단계는 복사 가능한 코드 블록과 함께, 명령어 의미/옵션 설명을 바로 아래에 적었습니다. Word 메뉴의 **참조 > 목차**에서 자동 목차를 생성할 수 있도록 모든 장/절은 제목 스타일(제목 1, 제목 2)을 사용했습니다.

# 1. 개요

gRPC를 처음 접하는 사용자가 VSCode에서 폴더 구조를 만들고, 가상환경을 구성하고, proto를 컴파일하여 파이썬 서버/클라이언트를 실행한 뒤, BloomRPC·Postman으로 검증하는 전 과정을 다룹니다. 또한 실제 진행 중 발생한 오류들을 “현상/원인/해결” 표로 정리해 재발을 방지합니다.

# 2. 폴더 구조 만들기

권장 폴더 구조(최상위 폴더 이름은 예시: grpc\_sample):

grpc\_sample/  
 ├─ proto/ ← proto 파일 보관  
 │ └─ sample.proto  
 ├─ server/ ← gRPC 서버 코드  
 │ └─ server.py  
 ├─ client/ ← gRPC 클라이언트 코드  
 │ └─ client.py  
 ├─ generated/ ← proto 컴파일 산출물(sample\_pb2\*.py)  
 ├─ venv/ ← 가상환경  
 └─ requirements.txt ← 패키지 목록

방법 A) VSCode 탐색기에서 새 폴더/파일로 수작업 생성 (가장 이해 쉬움).

방법 B) 터미널에서 한 번에 생성:

```powershell  
mkdir grpc\_sample; cd grpc\_sample  
mkdir proto, server, client, generated  
ni proto\sample.proto -it file | Out-Null  
ni server\server.py -it file | Out-Null  
ni client\client.py -it file | Out-Null  
ni requirements.txt -it file | Out-Null  
```  
명령어 설명:  
- `mkdir`: 폴더 생성  
- `ni`(New-Item): 파일 생성. `-it file`은 종류를 파일로 지정  
- `;`는 파워셸에서 여러 명령을 한 줄로 실행  
- `Out-Null`은 출력 숨김

# 3. 가상환경(venv)과 VSCode 인터프리터 설정

가상환경은 프로젝트별 독립 패키지 공간입니다(버전 충돌 방지).

```powershell  
python -m venv venv  
venv\Scripts\Activate.ps1  
```  
옵션/의미:  
- `python -m venv venv`: 현재 파이썬으로 venv 모듈을 실행해 ./venv 폴더에 가상환경 생성  
- `Activate.ps1`: 가상환경 활성화 (터미널 프롬프트에 (venv) 표기)

VSCode 인터프리터 지정:  
1) `Ctrl+Shift+P` → `Python: Select Interpreter`  
2) `.\venv\Scripts\python.exe` 선택  
3) 상태바에 `Python 3.x ('venv')` 표시 확인

# 4. requirements.txt 작성 및 설치

requirements.txt (정확한 패키지 명칭 사용 ― `grpc`는 아님):

```text  
grpcio==1.64.1  
grpcio-tools==1.64.1  
protobuf>=5.26.1  
```  
설치:  
```powershell  
pip install -r requirements.txt  
```  
설명:  
- `grpcio`: Python gRPC 런타임 (클라이언트/서버 import grpc)  
- `grpcio-tools`: `protoc` 플러그인(파이썬용 코드 생성)  
- `protobuf`: 생성된 코드가 기대하는 런타임 버전(>=5.26.1 권장)  
- `-r requirements.txt`: 파일에 적힌 버전대로 설치

# 5. proto 작성(sample.proto)

`proto/sample.proto`에 아래를 입력 후 저장:

```proto  
syntax = "proto3";  
  
package sample;  
  
service CostService {  
 rpc Stat (StatisticsQuery) returns (StatisticsReply);  
}  
  
message Filter {  
 string k = 1;  
 string v = 2;  
 string o = 3;  
}  
  
message StatisticsQuery {  
 repeated Filter filter = 1;  
}  
  
message StatisticsReply {  
 map<string, double> results = 1;  
}  
```  
개념:  
- Filter: key/value/operator 형태 조건  
- StatisticsQuery: filter 목록(AND)  
- CostService.Stat: 요청을 받아 집계 결과를 반환

# 6. proto 컴파일(코드 생성)

```powershell  
python -m grpc\_tools.protoc -I./proto --python\_out=./generated --grpc\_python\_out=./generated ./proto/sample.proto  
```  
옵션 설명:  
- `python -m grpc\_tools.protoc`: 파이썬에서 protoc 플러그인 실행  
- `-I./proto`: import 기준 경로(Include path)  
- `--python\_out=./generated`: 메시지 클래스(`\*\_pb2.py`) 출력 폴더  
- `--grpc\_python\_out=./generated`: 서비스 스텁/서버 코드(`\*\_pb2\_grpc.py`) 출력 폴더  
- 마지막 인자 `./proto/sample.proto`: 입력 proto 파일 경로  
생성물:  
- `generated/sample\_pb2.py`: 메시지 정의(파이썬 클래스)  
- `generated/sample\_pb2\_grpc.py`: gRPC Stub/Servicer 코드

# 7. 생성 코드의 역할

- sample\_pb2.py → Filter/StatisticsQuery/StatisticsReply 등 메시지 클래스로 변환

- sample\_pb2\_grpc.py → CostServiceStub(클라이언트)/CostServiceServicer(서버) 제공

※ 상대 임포트 미사용으로 `import sample\_pb2`가 생성될 수 있음 → 필요 시 아래처럼 수정

```python  
# in generated/sample\_pb2\_grpc.py  
from . import sample\_pb2 as sample\_\_pb2  
```

# 8. 서버 코드 작성

`server/server.py` 내용:

```python  
import grpc  
from concurrent import futures  
import generated.sample\_pb2 as sample\_pb2  
import generated.sample\_pb2\_grpc as sample\_pb2\_grpc  
  
class CostService(sample\_pb2\_grpc.CostServiceServicer):  
 def Stat(self, request, context):  
 print("받은 요청:", request)  
 results = {"azure": 200.0, "aws": 150.0}  
 return sample\_pb2.StatisticsReply(results=results)  
  
def serve():  
 server = grpc.server(futures.ThreadPoolExecutor(max\_workers=10))  
 sample\_pb2\_grpc.add\_CostServiceServicer\_to\_server(CostService(), server)  
 server.add\_insecure\_port("[::]:50051")  
 server.start()  
 print("🚀 gRPC 서버 실행 중: localhost:50051")  
 server.wait\_for\_termination()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 serve()  
```  
핵심:  
- Servicer 구현(Stat)  
- 서버 포트 바인딩: 50051  
- add\_\*\_to\_server로 서비스 등록 후 start/대기

# 9. 클라이언트 코드 작성

`client/client.py` 내용:

```python  
import grpc  
import generated.sample\_pb2 as sample\_pb2  
import generated.sample\_pb2\_grpc as sample\_pb2\_grpc  
  
def run():  
 with grpc.insecure\_channel("localhost:50051") as channel:  
 stub = sample\_pb2\_grpc.CostServiceStub(channel)  
 query = sample\_pb2.StatisticsQuery(  
 filter=[sample\_pb2.Filter(k="provider", v="azure", o="eq")]  
 )  
 response = stub.Stat(query)  
 print("서버 응답:", response.results)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 run()  
```  
핵심:  
- insecure\_channel로 서버에 연결(로컬 테스트)  
- Stub 생성 후 원격 메서드 Stat 호출

# 10. 실행(모듈 방식 권장)

프로젝트 루트(grpc\_sample)에서 각각 실행:  
```powershell  
python -m server.server # 서버 실행(터미널 1)  
python -m client.client # 클라이언트 실행(터미널 2)  
```  
설명:  
- `-m` : 패키지/모듈 경로로 실행(임포트 경로 문제 방지)  
- 루트에서 실행하면 `generated` 패키지를 안정적으로 찾음

# 11. BloomRPC / Postman으로 테스트

BloomRPC

1) proto 추가: `proto/sample.proto`  
2) Target: `localhost:50051`  
3) 메서드: `sample.CostService/Stat`  
4) Body(JSON):  
```  
{  
 "filter": [  
 {"k": "provider", "v": "azure", "o": "eq"}  
 ]  
}  
```

Postman(gRPC 탭)도 동일한 절차로 proto 임포트 후 동일 JSON을 전송합니다.

# 12. 전체 흐름도(텍스트)

```  
[폴더 구성]  
 ↓  
[venv 생성/Interpreter 선택]  
 ↓  
[requirements 설치]  
 ↓  
[proto 작성 → 코드 생성(sample\_pb2\*.py)]  
 ↓  
[서버 구현(server.py) · 클라이언트 구현(client.py)]  
 ↓  
[루트에서 -m 방식으로 실행]  
 ↓  
[BloomRPC/Postman으로 검증]  
```

# 13. 트러블슈팅(현상/원인/해결)

## 13.1 파일을 못 찾음 (No such file or directory)

현상: `python server.py` 실행 시 파일을 찾지 못함.

원인: 경로가 서버 폴더가 아님 또는 오타(예: phtyon).

해결: 루트에서 `python -m server.server` 또는 `cd server; python server.py`.

## 13.2 ModuleNotFoundError: No module named 'generated'

원인: 패키지 인식 실패(실행 위치가 server/이거나 generated에 \_\_init\_\_.py 미존재).

해결:  
- `generated/\_\_init\_\_.py` 생성(빈 파일)  
- 항상 루트에서 `python -m server.server` 실행  
- (대안) server.py 상단에 sys.path 추가:  
```  
import sys, os  
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), "..")))  
```

## 13.3 sample\_pb2 임포트 에러

현상: `generated/sample\_pb2\_grpc.py`에서 `import sample\_pb2`를 못 찾음.

원인: 생성 코드가 상대 임포트를 사용하지 않음.

해결: `generated/sample\_pb2\_grpc.py`를 아래처럼 수정:  
```  
from . import sample\_pb2 as sample\_\_pb2  
```  
또는 실행 전 `PYTHONPATH`에 generated 추가.

## 13.4 UNAVAILABLE: failed to connect to all addresses

원인: 서버가 꺼져있거나 포트가 다름, 방화벽.

해결: 서버 먼저 실행(50051 확인), 클라이언트 채널 주소 일치, 방화벽/보안툴 확인.

## 13.5 pip install 에러 (grpc / protobuf 'runtime\_version')

현상 1: requirements에 `grpc` 넣으면 설치 중 에러.  
원인: 공식 패키지는 `grpcio`. `grpc`는 사용하지 않음.  
해결: requirements에 `grpcio`, `grpcio-tools`만 사용.  
  
현상 2: `ImportError: cannot import name runtime\_version from google.protobuf`.  
원인: protobuf 런타임이 생성 코드가 기대하는 버전보다 낮음.  
해결:  
```powershell  
pip uninstall -y grpc google protobuf  
pip install --upgrade pip  
pip install grpcio==1.64.1 grpcio-tools==1.64.1 protobuf>=5.26.1  
python -m grpc\_tools.protoc -I./proto --python\_out=./generated --grpc\_python\_out=./generated ./proto/sample.proto  
```  
의미:  
- `uninstall -y`: 사용자 확인 없이 제거  
- 버전 맞춤 설치 후 proto 재컴파일

## 13.6 VSCode 경고: 전역 환경에 패키지 설치됨

의미: 전역 환경 패키지와 충돌할 수 있으니 가상환경 사용 권장.

해결: 이미 venv 사용 중이면 무시 가능. 새 프로젝트는 Create로 venv 생성.

# VSCode gRPC 프로젝트 – 처음부터 끝까지 (최신 가이드)

이 문서는 VSCode에서 gRPC 프로젝트를 시작할 때부터 서버/클라이언트 개발, 테스트, Git 관리까지 전 과정을 다룬 단계별 안내서입니다. 실제 진행 중 발생한 오류와 해결 방법도 포함했습니다. 각 단계는 코드 블록(bash 등)을 사용해 배경색상으로 구분되며, 복사 후 바로 실행할 수 있습니다.

## 0. 프로젝트 시작 전 준비 (Git & .gitignore)

VSCode에서 새 프로젝트를 만들 때 가장 먼저 .gitignore 파일을 생성하는 것이 중요합니다. 이렇게 해야 불필요한 venv, 캐시 파일이 원격 저장소에 올라가지 않습니다.

# Python venv  
.venv/  
venv/  
ENV/  
env/  
  
# Cache  
\_\_pycache\_\_/  
\*.py[cod]  
  
# Build  
\*.egg-info/  
dist/  
build/  
  
# IDE  
.vscode/  
.idea/  
  
# Logs  
\*.log

생성 후:

git init  
git add .gitignore  
git commit -m "Add .gitignore"

## 1. 개요

gRPC를 처음 접하는 사용자가 VSCode에서 폴더 구조를 만들고, 가상환경을 구성하고, proto를 컴파일하여 파이썬 서버/클라이언트를 실행한 뒤, BloomRPC·Postman으로 검증하는 전 과정을 다룹니다. 또한 실제 진행 중 발생한 오류들을 “현상/원인/해결” 표로 정리해 재발을 방지합니다.

## 2. Git 브랜치와 Pull Request 쉽게 이해하기

main = 원본/안정 버전

feature = 백업/실험 브랜치

merge/PR = 브랜치 간 합치기

### 브랜치 만들기

git checkout -b feature/grpc-first-test  
git push -u origin feature/grpc-first-test

### main에서 수정 후 feature에 반영하기

git switch feature/grpc-first-test  
git merge main

### feature 결과를 main에 반영하고 싶을 때

GitHub에서 Pull Request 생성

또는 로컬에서:

git switch main  
git merge feature/grpc-first-test  
git push

👉 혼자 개발할 때는 feature를 “백업/실험용”으로 쓰고 main은 깨끗하게 유지하세요. PR은 “실험한 걸 main에 합치고 싶을 때만” 사용하면 됩니다.

## 3. 폴더 구조 만들기

권장 폴더 구조:

grpc\_sample/  
 ├─ proto/  
 │ └─ sample.proto  
 ├─ server/  
 │ └─ server.py  
 ├─ client/  
 │ └─ client.py  
 ├─ generated/  
 ├─ venv/  
 └─ requirements.txt

터미널에서 생성:

mkdir grpc\_sample; cd grpc\_sample  
mkdir proto, server, client, generated  
ni proto\sample.proto -it file | Out-Null  
ni server\server.py -it file | Out-Null  
ni client\client.py -it file | Out-Null  
ni requirements.txt -it file | Out-Null

명령어 설명: - mkdir: 폴더 생성 - ni: 파일 생성 (PowerShell 전용) - ;: 여러 명령 한 줄 실행 - Out-Null: 출력 숨김

## 4. 가상환경(venv)과 VSCode 인터프리터 설정

python -m venv venv  
venv\Scripts\Activate.ps1

VSCode에서 인터프리터 지정 → Python: Select Interpreter → ./venv/Scripts/python.exe 선택.

## 5. requirements.txt 작성 및 설치

grpcio==1.64.1  
grpcio-tools==1.64.1  
protobuf>=5.26.1

설치:

pip install -r requirements.txt

## 6. proto 작성(sample.proto)

syntax = "proto3";  
  
package sample;  
  
service CostService {  
 rpc Stat (StatisticsQuery) returns (StatisticsReply);  
}  
  
message Filter {  
 string k = 1;  
 string v = 2;  
 string o = 3;  
}  
  
message StatisticsQuery {  
 repeated Filter filter = 1;  
}  
  
message StatisticsReply {  
 map<string, double> results = 1;  
}

## 7. proto 컴파일(코드 생성)

python -m grpc\_tools.protoc -I./proto --python\_out=./generated --grpc\_python\_out=./generated ./proto/sample.proto

## 8. 서버 코드 작성

import grpc  
from concurrent import futures  
import generated.sample\_pb2 as sample\_pb2  
import generated.sample\_pb2\_grpc as sample\_pb2\_grpc  
  
class CostService(sample\_pb2\_grpc.CostServiceServicer):  
 def Stat(self, request, context):  
 print("받은 요청:", request)  
 results = {"azure": 200.0, "aws": 150.0}  
 return sample\_pb2.StatisticsReply(results=results)  
  
def serve():  
 server = grpc.server(futures.ThreadPoolExecutor(max\_workers=10))  
 sample\_pb2\_grpc.add\_CostServiceServicer\_to\_server(CostService(), server)  
 server.add\_insecure\_port("[::]:50051")  
 server.start()  
 print("🚀 gRPC 서버 실행 중: localhost:50051")  
 server.wait\_for\_termination()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 serve()

## 9. 클라이언트 코드 작성

import grpc  
import generated.sample\_pb2 as sample\_pb2  
import generated.sample\_pb2\_grpc as sample\_pb2\_grpc  
  
def run():  
 with grpc.insecure\_channel("localhost:50051") as channel:  
 stub = sample\_pb2\_grpc.CostServiceStub(channel)  
 query = sample\_pb2.StatisticsQuery(  
 filter=[sample\_pb2.Filter(k="provider", v="azure", o="eq")]  
 )  
 response = stub.Stat(query)  
 print("서버 응답:", response.results)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 run()

## 10. 실행

python -m server.server # 서버 실행  
python -m client.client # 클라이언트 실행

## 11. BloomRPC / Postman 테스트

BloomRPC:

{  
 "filter": [  
 {"k": "provider", "v": "azure", "o": "eq"}  
 ]  
}

## 12. 전체 흐름도(텍스트)

[폴더 구성] → [venv 생성] → [requirements 설치] → [proto 작성/컴파일]  
→ [서버·클라이언트 구현] → [실행] → [BloomRPC/Postman 검증]

## 13. 트러블슈팅 (현상/원인/해결)

문서 마지막에 현상/원인/해결 정리 표는 기존 내용 유지.

✅ 이번 버전에는: - .gitignore 설정을 프로젝트 시작 단계에 추가 - Git 브랜치/PR 개념을 이슈정리 위에 배치 - 코드 블록은 전부 배경색 강조(bash,python, ```proto 등) 적용

으로 업데이트 되었습니다.

# 14. Struct/Value 연습 – proto + 서버 + 클라이언트 + BloomRPC 샘플 (Protobuf 공식 JSON)

14. Struct/Value 연습 – proto + 서버 + 클라이언트 + BloomRPC 샘플 (Protobuf 공식 JSON)

이 장은 한 번에 복사·실행 가능한 “완성 예제”입니다. 아래 순서대로 진행하면 동적 타입(google.protobuf.Value / Struct / ListValue)을 BloomRPC로 확실히 테스트할 수 있습니다. 모든 요청 본문은 Protobuf 공식 JSON 표기로 제공하며, 예상 응답도 함께 제시합니다.

# 1) proto (sample.proto) – 메서드 5종

기존 Stat에 더해, EchoValue / EchoStruct / SumCosts / FilterByProvider / EchoStructAsJson 를 선언합니다.

|  |
| --- |
| syntax = "proto3";  package sample;  import "google/protobuf/struct.proto";  service CostService {  rpc Stat (StatisticsQuery) returns (StatisticsReply);   // 동적 타입 연습용  rpc EchoValue (google.protobuf.Value) returns (google.protobuf.Value);  rpc EchoStruct (google.protobuf.Struct) returns (google.protobuf.Struct);  rpc SumCosts (google.protobuf.Struct) returns (SumReply);  rpc FilterByProvider (google.protobuf.Struct) returns (google.protobuf.Struct);  rpc EchoStructAsJson (google.protobuf.Struct) returns (google.protobuf.Struct); }  message Filter {  string k = 1;  string v = 2;  string o = 3; }  message StatisticsQuery {  repeated Filter filter = 1; }  message StatisticsReply {  map<string, double> results = 1; }  message SumReply {  double total = 1; } |

*proto/sample.proto*

# 2) proto 컴파일

|  |
| --- |
| python -m grpc\_tools.protoc -I./proto --python\_out=./generated --grpc\_python\_out=./generated ./proto/sample.proto |

*옵션: -I include 경로, --python\_out 메시지 코드, --grpc\_python\_out 서비스 스텁/서버 코드 생성*

※ 생성된 generated/sample\_pb2\_grpc.py 안에 `import sample\_pb2 as sample\_\_pb2` 라인이 있다면 아래처럼 수정하세요.

|  |
| --- |
| from . import sample\_pb2 as sample\_\_pb2 # in generated/sample\_pb2\_grpc.py |

# 3) 서버 (server/server.py) – 구현 전체

SumCosts는 관대한 파서로 문자열·다른 키(usdCost)도 처리하고, EchoStructAsJson은 디버깅을 쉽게 해줍니다.

|  |
| --- |
| import grpc from concurrent import futures from google.protobuf import struct\_pb2, json\_format import json  import generated.sample\_pb2 as sample\_pb2 import generated.sample\_pb2\_grpc as sample\_pb2\_grpc   class CostService(sample\_pb2\_grpc.CostServiceServicer):  def Stat(self, request, context):  print("받은 요청(Stat):", request)  results = {"azure": 200.0, "aws": 150.0}  return sample\_pb2.StatisticsReply(results=results)   # 1) Value 그대로 반환  def EchoValue(self, request, context):  print("받은 요청(EchoValue) kind:", request.WhichOneof('kind'))  return request   # 2) Struct 그대로 반환  def EchoStruct(self, request, context):  print("받은 요청(EchoStruct) fields:", len(request.fields))  return request   # 3) { items: [ {usd\_cost:number}, ... ] } 합계  def SumCosts(self, request, context):  req\_json = json\_format.MessageToJson(request)  print("SumCosts 요청(JSON):", req\_json)   try:  req = json.loads(req\_json) # 가장 직관적인 파싱  except Exception:  req = json\_format.MessageToDict(request)   items = req.get("items") or []  total = 0.0  for row in items:  if not isinstance(row, dict):  continue  val = row.get("usd\_cost")  if val is None:  val = row.get("usdCost")  try:  total += float(val)  except (TypeError, ValueError):  pass   return sample\_pb2.SumReply(total=total)   # 4) { data:[{provider,...}], provider:"azure" } → provider 로 필터  def FilterByProvider(self, request, context):  req = json\_format.MessageToDict(request)  want = req.get("provider")  data = req.get("data", [])  filtered = [row for row in data if isinstance(row, dict) and row.get("provider") == want]   out = struct\_pb2.Struct()  json\_format.ParseDict({"data": filtered}, out)  return out   # 5) Struct를 JSON 문자열로 래핑해 돌려줌(디버깅용)  def EchoStructAsJson(self, request, context):  as\_json = json\_format.MessageToJson(request)  out = struct\_pb2.Struct()  json\_format.ParseDict({"as\_json": as\_json}, out)  return out   def serve():  server = grpc.server(futures.ThreadPoolExecutor(max\_workers=10))  sample\_pb2\_grpc.add\_CostServiceServicer\_to\_server(CostService(), server)  server.add\_insecure\_port("[::]:50051")  server.start()  print("🚀 gRPC 서버 실행 중: localhost:50051")  server.wait\_for\_termination()   if \_\_name\_\_ == "\_\_main\_\_":  serve() |

*server/server.py*

# 4) 클라이언트 (client/client.py) – 전체

|  |
| --- |
| import grpc from google.protobuf import struct\_pb2, json\_format  import generated.sample\_pb2 as sample\_pb2 import generated.sample\_pb2\_grpc as sample\_pb2\_grpc   def call\_echo\_value(stub):  payload = struct\_pb2.Value()  json\_format.Parse('{"listValue":{"values":[{"stringValue":"a"},{"numberValue":1},{"boolValue":true},{"structValue":{"fields":{"x":{"numberValue":10}}}}]}}', payload)  resp = stub.EchoValue(payload)  print("EchoValue 응답(kind):", resp.WhichOneof('kind'))   def call\_echo\_struct(stub):  s = struct\_pb2.Struct()  json\_format.Parse('{"fields":{"msg":{"stringValue":"hi"},"meta":{"structValue":{"fields":{"a":{"numberValue":1},"b":{"listValue":{"values":[{"numberValue":1},{"numberValue":2},{"numberValue":3}]}}}}},"enabled":{"boolValue":true}}}', s)  resp = stub.EchoStruct(s)  print("EchoStruct 응답 fields:", len(resp.fields))   def call\_sum\_costs(stub):  s = struct\_pb2.Struct()  json\_format.Parse('{"fields":{"items":{"listValue":{"values":[{"structValue":{"fields":{"usd\_cost":{"numberValue":120.5}}}},{"structValue":{"fields":{"usd\_cost":{"numberValue":80}}}},{"structValue":{"fields":{"usd\_cost":{"numberValue":0.5}}}}]}}}}', s)  reply = stub.SumCosts(s)  print("SumCosts 응답 total:", reply.total)   def call\_filter\_by\_provider(stub):  s = struct\_pb2.Struct()  json\_format.Parse('{"fields":{"provider":{"stringValue":"azure"},"data":{"listValue":{"values":[{"structValue":{"fields":{"provider":{"stringValue":"aws"},"usd\_cost":{"numberValue":10}}}},{"structValue":{"fields":{"provider":{"stringValue":"azure"},"usd\_cost":{"numberValue":20}}}},{"structValue":{"fields":{"provider":{"stringValue":"azure"},"usd\_cost":{"numberValue":30}}}},{"structValue":{"fields":{"provider":{"stringValue":"gcp"},"usd\_cost":{"numberValue":40}}}}]}}}}', s)  resp = stub.FilterByProvider(s)  print("FilterByProvider 응답 JSON:", json\_format.MessageToJson(resp))   def call\_echo\_struct\_as\_json(stub):  s = struct\_pb2.Struct()  json\_format.Parse('{"fields":{"hello":{"stringValue":"world"}}}', s)  resp = stub.EchoStructAsJson(s)  print("EchoStructAsJson:", json\_format.MessageToJson(resp))   def run():  with grpc.insecure\_channel("localhost:50051") as channel:  stub = sample\_pb2\_grpc.CostServiceStub(channel)   # 기존 Stat  query = sample\_pb2.StatisticsQuery(  filter=[sample\_pb2.Filter(k="provider", v="azure", o="eq")]  )  print("Stat:", stub.Stat(query).results)   # 동적 타입 메서드 테스트  call\_echo\_value(stub)  call\_echo\_struct(stub)  call\_sum\_costs(stub)  call\_filter\_by\_provider(stub)  call\_echo\_struct\_as\_json(stub)   if \_\_name\_\_ == "\_\_main\_\_":  run() |

*client/client.py*

# 5) BloomRPC – Protobuf 공식 JSON으로 요청/예상 응답

5-1) EchoValue (Value → Value) | 요청

|  |
| --- |
| {  "listValue": {  "values": [  { "stringValue": "a" },  { "numberValue": 1 },  { "boolValue": true },  { "structValue": { "fields": { "x": { "numberValue": 10 } } } }  ]  } } |

예상 응답

|  |
| --- |
| { ...동일 구조의 Value... } |

5-2) EchoStruct (Struct → Struct) | 요청

|  |
| --- |
| {  "fields": {  "msg": { "stringValue": "hi" },  "meta": {  "structValue": {  "fields": {  "a": { "numberValue": 1 },  "b": {  "listValue": {  "values": [  { "numberValue": 1 },  { "numberValue": 2 },  { "numberValue": 3 }  ]  }  }  }  }  },  "enabled": { "boolValue": true }  } } |

예상 응답

|  |
| --- |
| { "fields": { ...요청과 동일... } } |

5-3) SumCosts (Struct → SumReply) | 요청

|  |
| --- |
| {  "fields": {  "items": {  "listValue": {  "values": [  { "structValue": { "fields": { "usd\_cost": { "numberValue": 120.5 } } } },  { "structValue": { "fields": { "usd\_cost": { "numberValue": 80 } } } },  { "structValue": { "fields": { "usd\_cost": { "numberValue": 0.5 } } } }  ]  }  }  } } |

예상 응답

|  |
| --- |
| { "total": 201.0 } |

5-4) FilterByProvider (Struct → Struct) | 요청

|  |
| --- |
| {  "fields": {  "provider": { "stringValue": "azure" },  "data": {  "listValue": {  "values": [  { "structValue": { "fields": { "provider": { "stringValue": "aws" }, "usd\_cost": { "numberValue": 10 } } } },  { "structValue": { "fields": { "provider": { "stringValue": "azure" }, "usd\_cost": { "numberValue": 20 } } } },  { "structValue": { "fields": { "provider": { "stringValue": "azure" }, "usd\_cost": { "numberValue": 30 } } } },  { "structValue": { "fields": { "provider": { "stringValue": "gcp" }, "usd\_cost": { "numberValue": 40 } } } }  ]  }  }  } } |

예상 응답

|  |
| --- |
| {  "data": [  {"provider":"azure","usd\_cost":20},  {"provider":"azure","usd\_cost":30}  ] } |

5-5) EchoStructAsJson (Struct → Struct{as\_json:string}) | 요청

|  |
| --- |
| { "fields": { "hello": { "stringValue": "world" } } } |

예상 응답

|  |
| --- |
| { "as\_json": "{"hello":"world"}" } |

# 6) 실행 순서 (루트=grpc\_sample)

|  |
| --- |
| # 1) 가상환경 활성화 venv\Scripts\Activate.ps1  # 2) 의존성 설치 pip install -r requirements.txt  # 3) proto 컴파일 python -m grpc\_tools.protoc -I./proto --python\_out=./generated --grpc\_python\_out=./generated ./proto/sample.proto  # 4) 서버 실행(터미널 1) python -m server.server  # 5) 클라이언트 실행(터미널 2) python -m client.client  # 6) BloomRPC에서 5개 메서드 호출 확인 # (요청 본문은 본 장의 샘플 JSON 사용) |