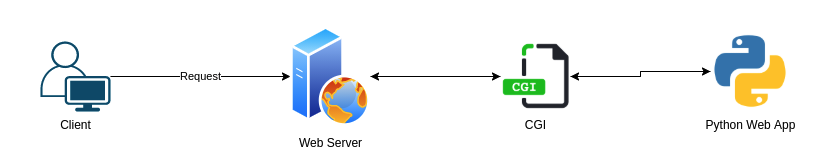
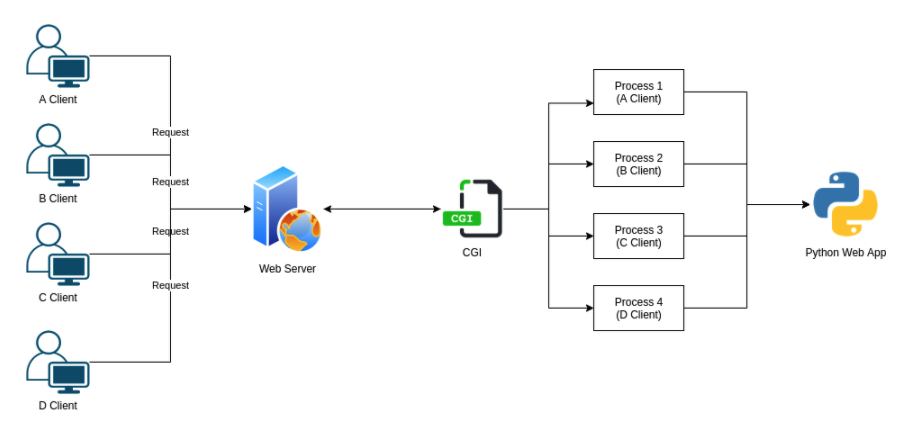
**WSGI & ASGI (참조:** [**https://velog.io/@hwaya2828/WSGI-ASGI**](https://velog.io/@hwaya2828/WSGI-ASGI) **)**

**CGI와 FastCGI**

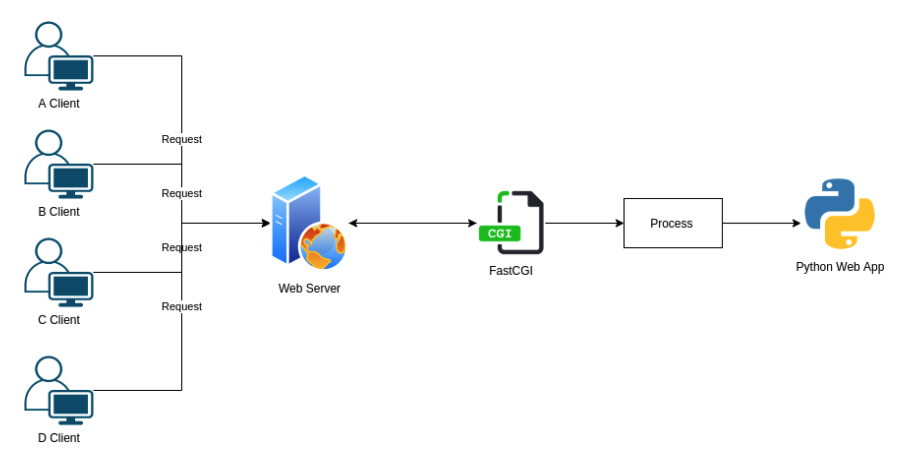
* 외부 애플리케이션과 웹 서버 (Nginx, Apache 등) 와 연결할 때 사용하는 **인터페이스**로 보통 CGI 를 사용
  + 서버로 들어온 요청을 웹 어플리케이션으로 넘겨줄 수 있어야 하는데, 이때 만약 서버마다 애플리케이션 언어마다 다른 형태를 가진다면 매우 번거로운 일이 될 테니 가능한 **공통의 표준 인터페이스를 제공**해주어야 함
    - 이 인터페이스가 CGI, Common Gateway Interface 임
  + **CGI** 는 Common Gateway Interface 의 약자로 **웹 서버와 외부 프로그램을 연결해주는 표준화된 프로토콜**



* 이러한 모형은 웹 서비스의 가장 일반적인 모습이지만 **과거에는 정적 HTML 파일 하나를 가지고 웹 서비스**를 하였기 때문에 **이러한 요소(CGI)가 필요하지 않았음**
  + 웹 서버가 정적 파일 등을 사용자에게 다운로드 방식으로 제공하는 것이 전부인데, 웹에 대한 수요가 증가함에 따라서 웹 서버가 처리할 수 없는 정보가 요청되었을 경우 그 처리를 외부 애플리케이션이 할 수 있도록 호출함으로써 중계 역할을 해줌



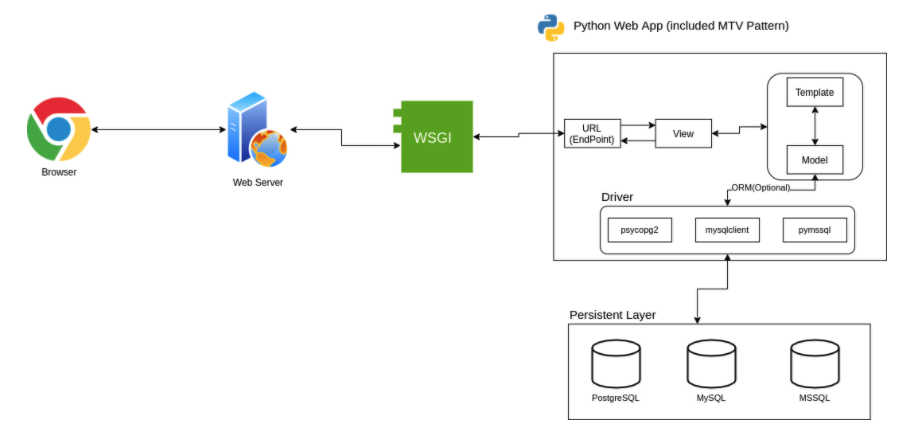
* 그러나 **CGI는 클라이언트의 요청이 발생할 때마다 프로세스를 추가로 생성하고 삭제**하게 됨
  + 예를 들어 위 그림과 같이 다수의 사용자가 동시에 요청할 경우 새로운 프로세스를 생성하고, 삭제하는 것이 빈번해지는데 이는 커널 리소스를 계속 생성•삭제하기 때문에 오버헤드가 심해지고, 성능 저하의 원인이 되기도 함
* 현대의 웹 서비스처럼 빈번하게 REST API가 호출되고, 요청과 응답을 수시로 반복하여 처리하는 곳에서 이러한 아키텍쳐는 맞지 않음
  + 그래서 생긴 것이 바로 **FastCGI**



* **FastCGI**는 **몇 번의 요청이 들어와도 하나의 프로세스만을 가지고 처리**하게 됨
  + 즉 메모리에 단 하나의 프로그램만을 적재하여 계속 재활용하기 때문에 **CGI에 비하여 오버헤드가 월등하게 감소**
* Java의 Tomcat 또한 Web Server + FastCGI를 채택한 형태로 다수의 사용자 접속에 유리
  + JVM 계열의 언어로 서버 프로그램을 작성하면 JBoss, Tomcat 등의 웹 애플리케이션 서버를 사용하는게 보편적이지만 이외에도 CGI를 수동으로 붙여서 사용할 수 있는 방법도 있음
* 그러나 Python에서는 이러한 WAS가 별도로 존재하지 않으며 결국 CGI, FastCGI 등을 이용해서 원하는 WAS 형태를 만들어 사용해야 하는데, 실제로 Python에서는 Python만의 게이트웨이 인터페이스가 존재

**WSGI**

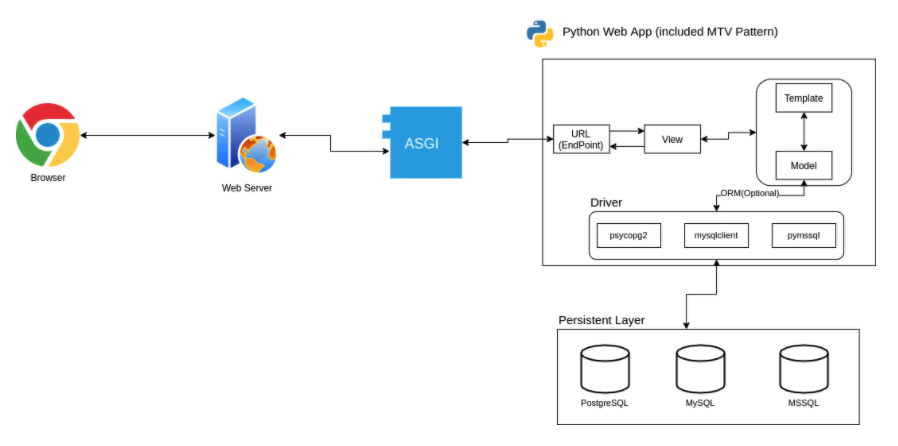
* **Python 애플리케이션, 스크립트가 웹 서버와 통신하기 위한 인터페이스**로써 **CGI 디자인 패턴을 모태**로 하여 만들어진 것이지만 실제 CGI와는 다소 차이가 있음
  + CGI의 가장 큰 단점은 요청이 들어올 때마다 새로운 프로세스를 생성한다는 점인데, 이런 단점을 보완하기 위해서 고안된 개념



* Python Web App을 MTV 패턴과 같이 뷰가 동시에 적용되는 프레임워크를 사용한다고 가정했을 때 모습
  + **WSGI**는 **CGI와 동일하게 웹 서버와 애플리케이션 중간에 위치**하는데, CGI와 다른 점은 **CGI는 매 요청마다 프로세스를 생성**한다는 점이고, **WSGI는 한 프로세스에서 모든 요청을 받는다**는 것
  + 좀 더 디테일하게 설명하자면 CGI는 매 요청마다 Fork 등의 함수를 통해 커널 리소스를 추가•반납하고, WSGI는 많은 요청을 콜백 (Callback) 으로 받아 처리
* WSGI로 대표적인 두 가지 방법 있음
  + ① Nginx, Apache에서 내장 모듈로 제공하는 server-often high profile 방식
    - 예시: mod-wsgi, mod-python 등
  + ② Python 코드로 작성된 Web App Server
    - 예시: gunicorn, uvicorn, cherrypy 등
* 결국 **WSGI**는 **웹 서버와 애플리케이션 사이에 미들웨어 역할**을 하며 기술적으로는 **웹 서버도 WSGI에 대한 작동 코드가 필요**하고, **애플리케이션 또한 WSGI에 대한 작동 코드가 필요한 Client-Server Model을 응용**한 것
  + 좀 더 간단히 설명해보면, 웹 서버가 애플리케이션의 코드를 직접적으로 읽을 수 없으므로 **중간의 미들웨어가 해당 코드를 읽어서 결과를 대신 반환**해주는 역할

**ASGI**

* WSGI만으로도 우리는 충분히 클라이언트의 요청을 웹 서버가 받아 내 애플리케이션으로 올 수 있도록 구성할 수 있었음
  + 하지만 공교롭게도 이것만으로는 현대 웹 서비스의 대용량 트래픽 처리를 유연하게 하기 위한 만족된 조건을 충족하기 어려운 점이 있음
* **Python**에서는 **asyncio, coroutine과 같은 비동기 처리를 지원**
  + 그러나 **WSGI**는 **동기 함수 처리만을 지원하여 여러 작업을 동시에 처리하는 것에 한계**가 있음
  + 가령 현대 웹 서비스에서는 웹 소켓 등을 사용한 실시간 채팅 서비스 등을 할 수도 있는데, WSGI로는 이러한 서비스를 구현하는 데 어려움
* 물론 안되는 것은 아님
  + **비동기 큐와 같은 Celery**를 잘만 활용한다면 **대용량 트래픽 처리를 요구하는 서비스 구현이 불가능한 것은 아니지만 트레이싱 등과 같은 유지 보수, 기본적인 구현이 쉽지 않다는 점**이 있음
  + 따라서 최근에는 **Django 3.0** 뿐만 아니라 **FastAPI** 등의 프레임워크에서도 **ASGI 인터페이스를 적용**하였으며 뒤따라 Falcon 프레임워크도 3.0부터 ASGI 개발에 들어갔음



* 운영 아키텍쳐로 봤을 때는 크게 다르지 않음
  + 그러나 WSGI와 다르게 **ASGI**는 **기본적으로 요청을 비동기로 처리**한다는 점이 큼
  + 따라서 WSGI에서는 지원되지 않는 **웹 소켓 프로토콜과 HTTP 2.0을 지원**
* 이러한 대표적인 ASGI Web App에는 **uvicorn**이 있음
  + uvicorn은 ASGI 기반의 웹 애플리케이션 서버로써 그 내장 모듈로 uvloop을 사용
  + uvloop에서 uv는 libuv 즉, Javascript V8에서 사용되는 비동기 모듈을 사용한 것
  + ASGI는 Cython 기반으로 C++ 언어로 작성되어 매우 빠른 속도를 제공한다는 것이 특징인데다다가 libuv를 사용하여 비동기 처리를 하니 Node.js와 같은 비동기 처리 속도를 어느 정도 누릴 수도 있다는 장점이 있음

**Tip!** **추가 내용**

**역사**

**CGI**

* 외부 애플리케이션과 웹서버 (Nginx, Apache) 와 연결할 때 사용하는 인터페이스
* 정적 HTML 하나로 웹 서비스 (웹 서버가 정적 파일 등을 사용자에게 다운로드 방식으로 제공) 를 하던 과거와는 달리, 웹에 대한 수요가 증가함에 따라 웹 서버로 처리할 수 없을 때 그 처리를 외부 애플리케이션이 할 수 있도록 CGI가 중계 역할을 해줌
  + 하지만 CGI는 클라이언트의 요청이 발생할 때 마다 프로세스를 추가•생성•삭제를 해야 함
  + 그래서 많은 사용자가 동시 요청을 하게 되면 성능 저하가 발생

**FastCGI**

* 현대의 웹 서비스는 REST API가 빈번히 발생하며, 요청과 응답을 수 없이 반복 처리 해야 함
  + 그런 이유로 FastCGI가 등장
  + 하나의 프로세스로 여러 요청을 처리할 수 있음

**WSGI**

* 파이썬에는 **파이썬 만의 게이트웨이 인터페이스**가 존재하는데 그것이 **WSGI**
* 스크립트가 웹 서버와 통신하기 위한 인터페이스이며, CGI 디자인 패턴을 참고한 것이지만 실제 CGI와는 차이가 있음
* 웹 서버가 애플리케이션의 코드를 직접적으로 읽을 수 없기 때문에, 중간에서 **미들웨어가 해당 코드를 읽어 결과를 대신 반환**

**WSGI & ASGI**

* django-admin startproject 를 통해 생기는 메인 프로젝트의 디렉토리에는 asgi.py와 wsgi.py가 생성됨
* **django**는 웹프레임워크일 뿐이므로 작동하기 위해 **웹 서버가 필요**
  + 하지만 대부분의 웹 서버는 기본적으로 Python을 사용하지 않음
  + 그래서 그 사이에서 **커뮤니케이션을 이루어줄 인터페이스**가 필요
    - 그 인터페이스가 되어주는 것이 **WSGI**와 **ASGI**

**WSGI**

* **웹 서버와 애플리케이션 사이의 통신**을 위한 **주요 파이썬 표준**
  + 동기(Synchronous)식 코드만 지원
* **WSGI**는 HTTP 스타일로 **request/response 형식**에 고정되어 있음
  + 즉 , 단일 처리만 가능한 동기 호출 방식
  + 그래서 긴 대기시간을 가지는 HTTP 연결에는 적합하지 않음
* HTTP는 Connection이 짧게 유지되는 특성을 지니고 있었기 때문에, Long-Polling HTTP와 Web Socket 같이 상대적으로 connection이 긴 특성을 지닌 Protocol과는 맞지 않았음
* **HTTP Request**는 application 내부에서 **오직 하나의 path**를 가질 수 있기 때문에, **여러 개의 path를 통해 이벤트를 수신하는 Web Socket의 이벤트를 처리할 수 없었음**

**ASGI**

* Django 사이트에서 **비동기 Python 기능과 비동기 Django 기능을 개발하면서 사용할 수 있게 해주는 새로운 비동기 친화적 표준**
* **request/response로 이루어진 WSGI**와 달리 **ASGI는 send/receive**로 되어 있어 비동기적으로 이벤트 처리가 가능한데, 그로인해 여러 송수신 이벤트가 가능
* **WSGI의 상위 집합으로 설계**되어있으며 asgiref 라이브러리를 통해 **ASGI 서버 내에서 WSGI를 실행 할 수도 있다**고 함
* django 3.0 에서부터 Channels 기능을 제공하고 있으며, 이것은 **ASGI 기반**으로 만들어 진 것
  + 기존의 HTTP 통신을 넘어, 웹 소켓, 채팅 프로토콜, ioT 프로토콜 등을 처리할 수 있음

**Tip!** **추가 내용**

**Nginx & Gunicorn**

* **WSGI**는 Python에서 어플리케이션이 웹 서버와 통신하기 위한 인터페이스
* WSGI Server는 웹 서버와 WSGI를 지원하는 웹어플리케이션 사이에서 동작하고 gunicorn, uwsgi 등이 있음
* gunicorn만 있어도 http request를 처리할 순 있지만, gunicorn에는 없고 nginx에는 있는 기능 때문에 둘을 연동해서 씀
  + ① djangodml media, css 등 static한 요청은 직접 처리하고 다이나믹한 요청을 gunicorn에 넘김
    - gunicorn으로 넘어가는 순간 자원 사용이 크게 늘어 static한 요청을 따로 처리해주는게 중요
  + ② nginx는 C로 구현되어 속도와 메모리 사용 측면에서 뛰어남
* 둘을 연동하면 동시에 많은 요청을 처리 할 수 있고, 훨씬 안정화된 서버를 구축 할 수 있게 됨
* Django의 runserver는 단일 프로세스, 단일 쓰레드라 nginx에서 아무리 많은 요청을 처리해도 결국 Django 내장 서버에서 병목이 생김

**Synchronous vs Asynchronous**

* **동기(Synchronous)식 코드**란 **위에서 아래로 부터 내려오며 코드 실행이 진행**되는 방식을 말함
* 그와 반대로 **비동기(Asynchronous)적 코드**는 쉽게 말해 **동기식 코드가 실행되고 난 뒤 실행되는 코드**
* 둘은 런타임시 발생하는 지연시간에 가장 큰 차이가 있음
* **동기식 코드**는 앞의 작업이 아직 끝나지 않으면 뒤는 가만히 기다리고 있어야 함
  + 순서대로 진행되기 때문에 유지와 보수가 쉽고, 코드의 파악이 쉬움
  + 그래서 평소 python으로 코드를 짤때 print() 등으로 breakpoint 를 둬서 한 단계씩 디버깅을 하며 에러를 잡아낼 수 있음
* 반면 **비동기식 코드**는 단계별로 독립적인 작업이라면 순차적으로 진행되지 않아도 되며, 동시에 종료될 필요도 없음
  + 하나의 쓰레드로 동시 처리를 하는 것