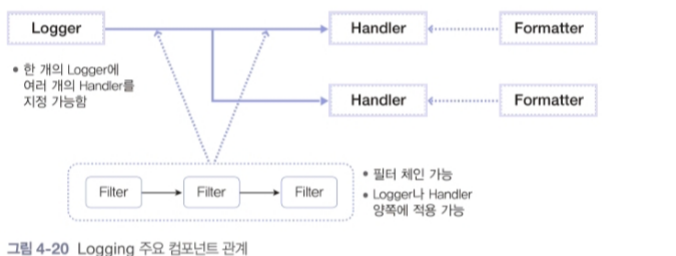
4.6 로그 남기기

장고의 로깅은 기본적으로 파이썬의 로깅 체계를 그대로 따르면서 일부만 추가. 파이썬의 로깅 모듈을 보면 로거, 핸들러, 필터, 포맷터 4가지 주요 컴포넌트를 정의하고 있음. 각각의 주요 역할과 관계를 아래 그림에 표시.

또한 장고의 runserver나 웹 서버에 의해 장고가 실행될 때 장고는 settings.py 파일에 정의된 LOGGING\_CONFIG, LOGGING 항목을 참고하여 로깅에 관련된 설정을 처리함.

settings.py 파일에 관련된 항목이 없더라도 디폴트 로깅 설정으로 처리됨.

따라서 장고의 로깅은 실행되는 시점부터 작동하여 로그가 출력됨. 또는 적절한 로거만 획득하면 우리가 원하는 로그를 기록하는 것이 가능한 상태임.



1) 로거

로거는 로깅 시스템의 시작점으로, 로그 메시지를 처리하기 위해 메시지를 담아두는 저장소.

모든 로거는 이름을 가지고 있음. 또한 로거는 로그 레벨을 갖게 되는데, 이는 로그 메시지의 중요도에 따라 자신이 어느 레벨 이상의 메시지를 처리할지에 대한 기준이 됨

파이썬의 로그 레벨

|  |  |  |
| --- | --- | --- |
| 로그 레벨 | 정수 값 | 설명 |
| NOTSET | 0 | 로그 레벨의 최하위 수준  로거 또는 핸들러가 생성될 때 별도 설정이 없으면 갖는 디폴트 로그 레벨 |
| DEBUG | 10 | 디버그 용도로 사용되는 정보 |
| INFO | 20 | 일반적이고 보편적인 정보 |
| WARNING | 30 | 문제점 중에서 덜 중요한 문제점이 발생 시 이에 대한 정보 |
| ERROR | 40 | 문제점 중에서 주요 문제점이 발생 시 이에 대한 정보 |
| CRITICAL | 50 | 치명적인 문제점이 발생 시 이에 대한 정보, 로그 레벨의 최상위 수준 |

로거에 저장되는 메시지를 로그 레코드라고 하며, 로그 레코드 역시 그 메시지의 심각성을 나타내는 로그 레벨을 가짐. 로그 레코드는 로그 이벤트에 대한 메타 정보도 가질 수 있는데, 일례로 스택 트레이스 정보나 에러 코드 등을 담을 수 있음.

메시지가 로거에 도착하면, 로그 레코드의 로그 레벨과 로거의 로그 레벨을 비교. 로그 레코드의 로그 레벨이 로거 자체의 로그 레벨과 같거나 그보다 높으면 메시지 처리를 계속 진행하고, 더 낮으면 그 메시지는 무시됨. 이렇게 로그 레코드와 로거의 로그 레벨을 비교하여 메시지 처리를 진행하는 것으로 결정되면 로거는 메시지를 핸들러에게 넘겨줌.

2. 핸들러

핸들러는 로거에 있는 메시지에 무슨 작업을 할지 결정하는 엔진.

즉, 메시지를 화면이나 파일 또는 네트워크 소켓 등 어디에 기록할 것인지와 같은 로그 동작을 정의함. 핸들러도 로거와 마찬가지로 로그 레벨을 가지고 있음. 로그 레코드의 로그 레벨이 핸들러의 로그 레벨보다 더 낮으면 핸들러는 메시지를 무시함.

로거는 핸들러를 여러 개 가질 수 있고, 각 핸들러는 서로 다른 로그 레벨을 가질 수 있음. 이렇게 해서 메시지의 중요도에 따라 다른 방식의 로그 처리가 가능함. 예를 들어 ERROR 또는 CRITICAL 메시지는 표준 출력으로 보내는 핸들러를 하나 만들고, 차후 분석을 위해 ERROR 및 CRITICAL 메시지를 포함한 모든 메시지를 파일에 기록하는 또 다른 핸들러를 만들 수도 있음

3. 필터

로그 레코드가 로거에서 핸들러로 넘겨질 때, 필터를 사용해서 로그 레코드에 추가적인 제어할 수 있음. 기본 제어 방식은 로그 레벨을 지정하여 그 로그 레벨에 해당되면 관련 로그 메시지를 처리하는 것임. 그런데 필터를 적용하면 로그 처리 기준을 추가할 수 있음. 예를 들어 필터를 추가하여 ERROR 메시지 중에서 특정 소스로부터 오는 메시지만 핸들러로 넘길 수 있음.

필터를 사용하면 로그 레코드를 보내기 전에 수정하는 것도 가능함. 예를 들어 어떤 조건에 맞으면 ERROR 로그 레코드를 WARNING 로그 레벨로 낮춰주는 필터를 만들 수도 있음.

필터는 로거 또는 핸들러 어디에나 적용이 가능하고, 여러 개의 필터를 체인 방식으로 동작시킬 수도 있음.

4. 포맷터

로그 레코드는 최종적으로 텍스트로 표현되는데, 포맷터는 텍스트로 표현 시 사용할 포맷을 지정해줌. 포맷터는 보통 파이썬의 포맷 스트링을 사용하지만, 사용자 정의 포맷터도 만들 수 있음.

5. 로거 사용 및 로거 이름 계층화

로그를 기록하기 위해서는 앞서 설명한 로거, 핸들러, 필터, 포맷터 등을 설정한 후에 코드 내에서 로깅 메소드를 호출하면 됨.

logging.getLogger() 메소드를 호출하면 로거 객체를 얻을 수 있음. 로거 객체는 이름을 가지며, 로거를 계층화할 때 이름으로 각 로거를 구분함. 앞의 예시에서는 mylogger라는 이름의 로거를 사용하였음.

관행적으로 로거 이름에 \_\_name\_\_ 구문을 사용하는데, 이는 이 구문이 있는 파일의 파이썬 모듈 경로를 말함. 만일 ch3/polls/views.py 파일에서 이 구문을 사용하고 있다면 \_\_name\_\_ 변수값은 polls.views가 됨. 로깅 호출을 모듈 단위로 처리할 수 있어서 많이 사용하는 구문임. 이렇게 모듈 단위로 로그를 기록하고 싶으면 로거를 구분하는 이름을 아래처럼 도트(.) 방식으로 명명해주면 됨.

#로거 이름으로 계층화

logger = logging.getLogger(‘project.interesting.stuff’)

도트 방식의 로거 이름은 계층화를 이룸. 즉, project.interesting.stuff 로거의 부모는 projecting.interesting 로거이고, project.interesting 로거의 부모는 project 로거가 됨. 참고로 로거의 이름을 빈 문자열(‘’)로 지정하면 파이썬의 최상위 로거(루트 로거)가 됨

로거의 계층화가 중요한 이유: 로깅 호출은 부모 로거에게 전파되기 때문

이렇게 하면 로거 트리의 최상단 루트 로거에서 핸들러 하나만을 만들어도 하위 로거의 모든 로깅 호출을 잡을 수 있음. 위 예에서 project 이름 공간에 정의된 로그 핸들러는 project.interesting 로거 및 project.intersting.stuff 로거가 보내주는 모든 로그 메시지를 잡을 수 있음. 이런 로깅 호출의 전파는 로거 단위로 제어할 수 있는데, 특정 로거에서 상위 로거로 전파되는 것을 원하지 않으면 전파 기능을 비활성화시킬 수도 있음.

로거 객체는 로그 레벨별로 로깅 호출 메소드를 갖고 있음.

-logging.debug(): DEBUG 레벨의 로그 레코드를 생성함

-logger.info(): INFO 레벨의 로그 레코드를 생성함.

-logger.warning(): WARNING 레벨의 로그 레코드 생성

-logger.error(): ERROR 레벨의 로그 레코드 생성

-logger.critical(): CRITICAL 레벨의 로그 레코드 생성

이외에 두 가지 로깅 메소드 추가

-logger.log(): 원하는 로그 레벨을 정해서 로그 메시지를 생성

-logger.exception(): 익셉션 스택 트레이스 정보를 포함하는 ERROR 레벨의 로그 메시지를 생성

6. 장고의 디폴트 로깅 설정

로깅 메소드를 호출했을 때 로그 메시지를 원하는 대로 기록하기 위해서는 당연히 로거, 핸들러, 필터, 포맷터 등을 설정해야 함. 파이썬의 로깅 라이브러리는 다양한 설정 방식을 제공하고 있는데, 장고는 그중에서 사전형 설정 방식을 디폴트로 사용함. 이 방식은 settings.py 파일의 LOGGING 항목에, 로깅 속성을 사전 형식으로 정의하게 됨. 이러한 설정에는 로거, 핸들러, 필터, 포맷터에 대한 정의뿐만 아니라, 각 컴포넌트의 로그 레벨과 같은 속성들도 정의함.

settings.py 파일에 LOGGING 항목을 지정하지 않으면 장고는 디폴트 로깅 설정을 사용함. 우리가 로깅 설정을 할 때, 이러한 디폴트 로깅 설정을 유지할 수도 있고, 디폴트 설정을 무시하고 새로 로깅 설정을 할 수도 있음.

이번 절에서는 우선 장고의 디폴트 로깅 설정 내용을 알아본다. 장고의 로깅 방식이 기본적으로 어떻게 동작하는지 알 수 있으며, 무엇보다 디폴트 설정 내용을 참고해서 로깅을 설정하는 경우가 많기 때문임. 다음은 디폴트 로깅 설정 파일의 위치와 설정 내용임.

디폴트 로깅 설정

#site-packages/django/utils/log.py

DEFAULT\_LOGGING = {

‘version’: 1, #설정이 dictConfig version 1 형식인데, 현재는 버전이 하나

‘disable\_existing\_loggers’: False, #기존의 로거들 비활성화하지 않음. 이전 버전과의 호환성을 위한 항목으로 디폴트는 True이며 기존 로거들을 비활성화. 비활성화의 의미는 로거들을 삭제하는 것이 아니고 로깅 동작만을 중지시키는 것이므로 혼동의 여지가 있음. 따라서 장고에서는 이 항목을 False로 사용하도록 권장함.

‘filters’: { #필터 2개를 정의함

‘require\_debug\_false’: { #require\_debug\_false 필터는 DEBUG=False인 경우만 핸들러가 동작하도록 함. 특별키 ()의 의미는 필터 객체를 생성하기 위한 클래스를, 파이썬의 기본 클래스와는 다르게 장고에서 별도로 정의했다는 것을 알려줌.

‘()’: ‘django.utils.log.RequireDebugFalse’,

},

‘require\_debug\_true’: { #require\_debug\_true 필터는 DEBUG=False인 경우만 핸들러가 동작하도록 함. 특별키()의 의미는 위와 동일.

‘()’: ‘django.utils.log.RequireDebugTrue’,

},

},

‘formatters’: { #포맷터 1개를 정의

‘django.server’: { #django.server 포맷터는 로그 생성 시각과 로그 메시지만을 출력함. 특별키()의 의미는 포맷터 객체를 생성하기 위한 클래스를 별도로 정의했다는 것을 알려줌.

‘()’: ‘django.utils.log.ServerFormatter’,

‘format’: ‘[%(server\_time)s] %(message)s’,

}

},

‘handlers’: { #3개의 핸들러를 정의함

‘console’: { #console 핸들러는 INFO 레벨 및 그 이상의 메시지를 표준 에러로 출력해주는 StreamHandler 클래스를 사용. 이 핸들러는 require\_debug\_true 필터를 사용

‘level’: ‘INFO’,

‘filters’: [‘require\_debug\_true’],

‘class’: ‘logging.StreamHandler’,

},

‘django.server’: { #django.server 핸들러는 INFO 레벨 및 그 이상의 메시지를 표준 에러로 출력해주는 StreamHandler 클래스를 사용. 이 핸들러는 django.server 포맷터를 사용.

‘level’: ‘INFO’,

‘class’: ‘logging.StreamHandler’,

‘formatter’: ‘django.server’,

},

‘mail\_admins’: { #mail\_admins 핸들러는 ERROR 및 그 이상의 로그 메시지를 사이트 관리자에게 이메일로 보내주는 AdminEmailHandler 클래스 사용. 이 핸들러는 require\_debug\_false 필터 사용

‘level’: ‘ERROR’,

‘filters’: [‘require\_debug\_false’],

‘class’: ‘django.utils.log.AdminEmailHandler’

}

},

‘loggers’: { #2개의 로거 정의

‘django’: { #django 로거는 INFO 및 그 이상의 로그 메시지를 console 및 mail\_admins 핸들러에게 보냄. django.\* 계층 즉, django 패키지의 최상위 로거임.

‘handlers’: [‘django.server’], #django.server 로거는 INFO 레벨 및 그 이상의 메시지를 django.server 핸들러에게 보냄. 상위 로거로 로그 메시지를 전파하지 않음. 이 로거는 장고의 개발용 웹 서버인 runserver에서 사용하는 로거. 5XX 응답은 ERROR 메시지, 4XX 응답은 WARNING 메시지, 그 외는 INFO 메시지로 출력됨

‘level’: ‘INFO’,

‘propagate’: False,

},

}

-만일 DEBUG=True이면, django.\* 계층에서 발생하는 로그 레코드는 INFO 레벨 이상일 때 콘솔로 보내짐.

-만일 DEBUG=False이면, django.\* 계층에서 발생하는 로그 레코드는 ERROR 레벨 이상일 때 관리자에게 이메일 전송됨.

-django.server 로거는, DEBUG 값에 무관하게, 로그 레코드가 INFO 레벨 이상이면 콘솔로 보냄. django.\* 계층의 다른 로거들과는 다르게 django 로거로 전파하지 않음.

7. 장고의 로깅 추가 사항 정리

앞 절에서 설명한 내용 이외에도 장고 패키지에는 몇 가지가 더 추가됨.

-django 로거: 앞 절에서 설명

-django.request 로거: 요청 처리와 관련된 메시지 기록. 5XX 응답은 ERROR 메시지, 4XX 응답은 WARNING 메시지로 발생. 이 로거에 담기는 메시지는 2개의 추가적인 메타 항목을 가짐.

-status\_code: HTTP 응답 코드

-request: 로그 메시지를 생성하는 요청 객체

-django.server 로거: 앞 절에서 설명.

-django.template 로거: 템플릿을 렌더링하는 과정에서 발생하는 로그 메시지를 기록

-django.db.backends 로거: 데이터베이스와 관련된 메시지 기록. 예를 들어, 애플리케이션에서 사용하는 모든 SQL 문장들이 이 로거에 DEBUG 레벨로 기록됨. 이 로거에 담기는 메시지는 아래처럼 추가적인 메타 항목을 가짐. 성능상의 이유로, SQL 로깅은 settings.DEBUG 항목이 True인 경우만 활성화됨

-duration: SQL 문장을 실행하는 데 걸린 시간

-sql: 실행된 SQL 문장

-params: SQL 호출에 사용된 파라미터

-django.security.\* 로거: 사용자가 보안 측면에서 해를 끼칠 수 있는 동작을 실행한 경우, 이에 대한 메시지를 기록. 예를 들어 HTTP Host 헤더가 ALLOWED\_HOSTS에 없다면 장고는 400 응답을 리턴하고, 에러 메시지가 django.security.DisallowedHost 로거에 기록됨.

-django.db.backends.schema 로거: 데이터베이스의 스키마 변경 시 사용된 SQL 쿼리를 기록

장고에서 추가한 핸들러는 1개임

-AdminEmailHandler: 앞 절에서 설명함.

장고는 또한 3개의 필터 추가

-CallBackFilter: 이 필터는 콜백 함수를 지정해서 필터를 통과하는 모든 메시지에 대해 콜백 함수를 호출해줌. 콜백 함수의 리턴값이 False이면 메시지 로깅은 더 이상 처리하지 않음

-RequireDebugFalse: 앞 절에서 설명

-RequireDebugTrue: 앞 절에서 설명

8. 로깅 설정-디폴트 설정 유지

장고는 항상 로깅이 가능한 상태. 로그가 필요한 시점에 로거를 취득하고 로깅 메소드를 호출하면 됨. 앞 절에서는 장고에서 제공하는 로거를 살펴보았는데 이들 로거를 그대로 사용하는 경우는 많지 않음. 왜냐하면 이들 로거는 장고 패키지의 구성에 맞춰 사용되고 있는 것이고 우리가 개발하고 있는 애플리케이션에 사용할 로거가 필요하기 때문임.

따라서 우리가 필요한 로거를 추가로 설정해주고, 이 로거를 취득하여 사용하면 됨. 로거를 추가하는 것은 settings.py 파일에 LOGGING 항목으로 가능함. LOGGING 항목에는 새로운 로거를 추가하는 것도 가능하지만, 기존의 디폴트로 설정된 로거들을 오버라이딩하여 핸들러, 필터, 포맷터 등을 동작을 변경하는 방법을 많이 사용함.

로깅 설정 예시-디폴트 설정 유지

#mysite/settings.py

LOGGING\_CONFIG = ‘logging.config.dictConfig’ #생략 가능 LOGGING\_CONFIG는 로깅 설정에 사용하는 함수를 지정하는 항목임. 이 항목의 디폴트 값이 dictConfig이므로 이 라인은 생략 가능

LOGGING = { #개발자가 로깅을 설정할 때는 LOGGING 항목을 사용

‘version’: 1, #앞 절에서 설명한 내용과 동일

‘disable\_existing\_loggers’: False,

‘formatters’: {

‘verbose’: { #verbose 포맷터 정의. 이는 [로그 메시지를 기록한 시간], 로그 레벨 이름, [로거이름: 라인번호], 로그 메시지 순서로 출력함. 로그 메시지를 기록한 시간에 대한 포맷은 날짜/월축약형/연도 시(24시 기준):분:초 형식으로 출력함

‘format’ : “[%(asctime)s] %(levelname)s [%(name)s:%(lineno)s]

%(message)s”, ‘datefmt’ : “%d/%b/%Y %H:%M:%S”

},

},

‘handlers’: {

‘file’: { #file 핸들러를 정의함. file 핸들러는 DEBUG 및 그 이상의 메시지를 파일로 출력해주는 FileHandler 클래스를 사용함. FileHandler 클래스에 의해서 로그가 기록되는 파일 이름은 C:\RedBook\ch5\logs\logfile임. 또한 위에서 정의한 verbose 포맷터 사용

‘level’: ‘DEBUG’

‘class’: ‘logging.FileHandler’,

‘filename’: os.path.join(BASE\_DIR, ‘logs’, ‘mysite.log’),

‘formatter’: ‘verbose’

},

‘loggers’: { #2개의 로거 설정

‘django’: { #django 로거는 디폴트로 설정되어 있는 로거인데, 핸들러와 로거 레벨을 오버라이딩하여 동작 방식을 변경

‘handlers’: [‘file’],

‘level’: ‘DEBUG’,

},

‘mysite’: { #mysite 로거를 새롭게 정의. 이 로거는 DEBUG 및 그 이상의 메시지를 file 핸들러에게 보냄. 즉 로그 메시지를 파일에 기록하게 함. 로거에서 level을 정의하면 이는 핸들러에서 정의한 level을 오버라이딩함.

‘handlers’: [‘file’],

‘level’: ‘DEBUG’,

},

},

}

이번 절에서 설명한 로깅 설정은 장고의 디폴트 설정을 유지하면서 개발자의 로깅을 설정한 것임. 즉 장고에서 제공하는 로거들이 모두 동작하도록 설정하였음. 다만 동작하는 방식은 오버라이딩으로 변경할 수 있음.

반면 장고의 디폴트 설정을 모두 무시하고 새롭게 로깅 방식을 설정할 수도 있음.

9. 로깅 설정-디폴트 설정 무시

만일 파이썬의 로깅 시스템에 익숙한 독자라면 장고의 디폴트 설정을 무시하고 자신만의 로깅 방식으로 설정할 수도 있음. 또는 장고의 로깅 설정이 장황하다고 생각해서 간단하게 설정하고자 할 때도 기존의 디폴트 설정 무시할 필요 있음.

즉 장고에서 이미 정의한 로거들을 사용하지 않고, 새롭게 로거들을 설정할 때는 다음과 같이 작성

LOGGING\_CONFIG = None #LOGGING\_CONFIG=None이면 장고가 기본적으로 수행하는 DEFAULT\_LOGGING 설정 과정을 건너뜀. 그래서 디폴트 로깅 설정이 이뤄지지 않음.

LOGGING = { #개발자가 원하는 내용으로 로거, 핸들러, 필터, 포맷터 등 정의

(원하는 내용으로 로깅 컴포넌트들을 설정함)

}

import logging.config #dictConfig() 함수를 사용하기 위해 해당 모듈 임포트

logging.config.dictConfig(LOGGING) #직접 dictConfig() 함수를 호출하여 사전형 방식으로 LOGGING 항목에 정의된 내용 설정