bluevine

Django Settings with Pydantic

Ruth Dubin
05 Jan 2025

Who am I?

Ruth Dubin

A Senior backend developer

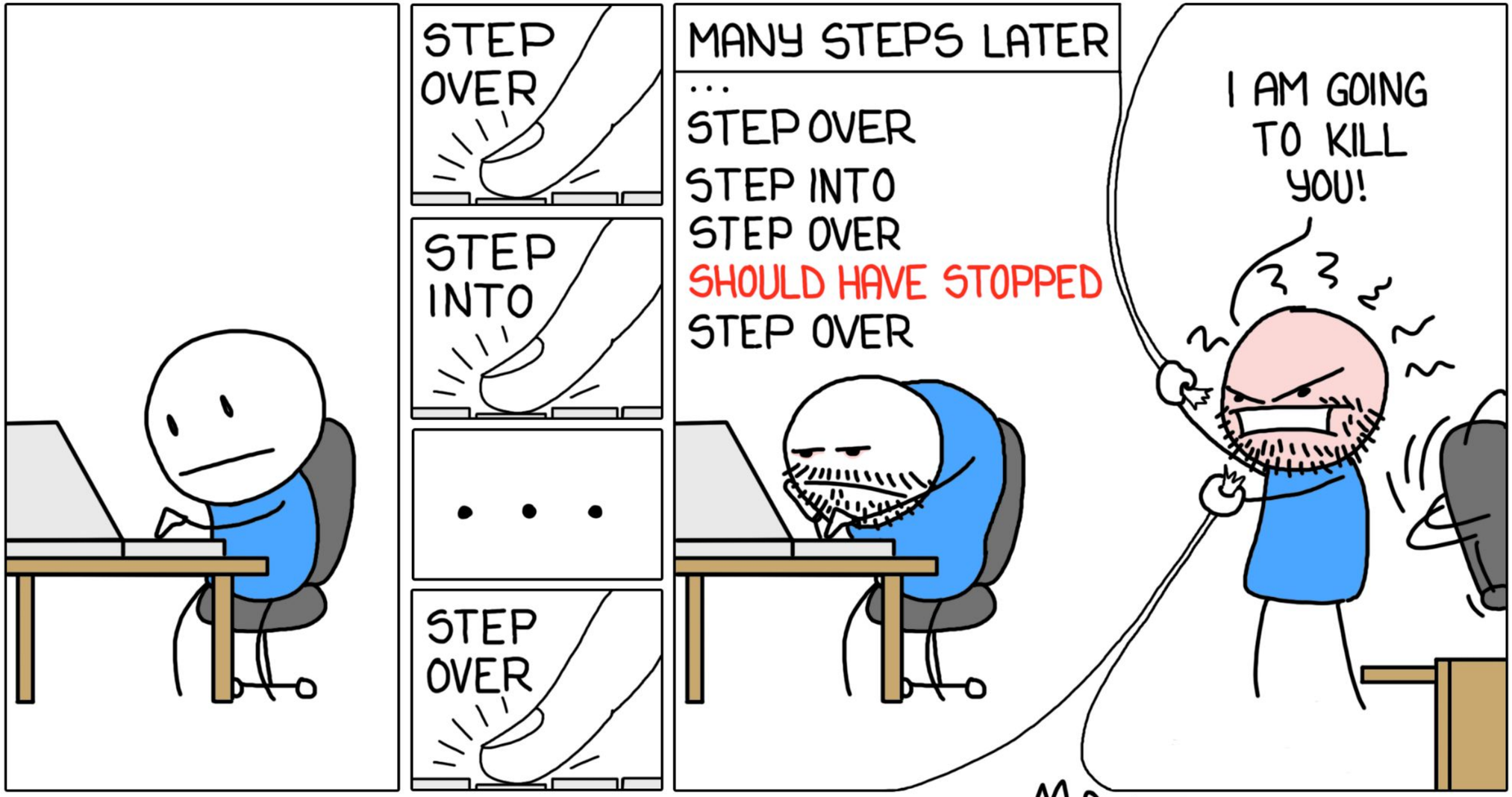
Bluevine Core team

```python
#app.py

from django.core.mail import send_mail

send_mail(
    'Subject',
    'Message body',
    'from@example.com',
    ['to@example.com'],
    fail_silently=False,
)
```

```python
#settings.py

EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'wrong-server.com'  # Points to a non-SMTP server
EMAIL_PORT = 587
EMAIL_USE_TLS = True
EMAIL_HOST_USER = os.getenv('EMAIL_HOST_USER')  # Might be None
EMAIL_HOST_PASSWORD = os.getenv('EMAIL_HOST_PASSWORD')  # Might be None
```

```
Traceback (most recent call last):
  File "/path/to/your/project/env/lib/python3.9/site-packages/django/core/mail/bac
    self.connection.login(self.username, self.password)
  File "/usr/lib/python3.9/smtplib.py", line 734, in login
    (code, resp) = self.auth(
  File "/usr/lib/python3.9/smtplib.py", line 744, in auth
    raise SMTPException("SMTP AUTH extension not supported by server.")
smtplib.SMTPException: SMTP AUTH extension not supported by server.
```

STEP BY STEP DEBUGGING



We are missing an environment variable!

We forgot to cast the environment variable’s type!

The port that was provided
was wrong!

Overview

1. What is Django settings?
2. Pydantic-settings Introduction
3. Integrating Django settings with Pydantic-settings
4. 3rd party Resources Health Checks
5. Pydantic Schema

Django settings are used
to configure the Django framework
and its applications

Why Traditional Methods Fall Short?

Directory structure

```
my_project/
    ├── my_project/
    │   ├── __init__.py
    │   ├── settings/
    │   │   ├── __init__.py
    │   │   ├── base.py
    │   │   ├── development.py
    │   │   ├── staging.py
    │   │   └── production.py
    │   ├── urls.py
    │   ├── wsgi.py
    │   └── asgi.py
    ├── manage.py
```

```python
# my_project/settings/base.py

import os

# Common settings for all environments
SECRET_KEY = os.getenv('SECRET_KEY',
'your-default-secret-key')

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Other common apps...
]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'mydb',
        'USER': 'myuser',
        'PASSWORD': 'mypassword',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}

# Other common settings...
```

```python
# my_project/settings/development.py

from .base import *

DEBUG = True
ALLOWED_HOSTS = ['localhost', '127.0.0.1']

# Use SQLite or other development database configuration
DATABASES['default'] = {
    'ENGINE': 'django.db.backends.sqlite3',
    'NAME': BASE_DIR / 'db.sqlite3',
}

# Other development-specific settings (e.g., logging, email, etc.)
```

```python
# my_project/settings/staging.py

from .base import *

DEBUG = False
ALLOWED_HOSTS = ['staging.example.com']

DATABASES['default'] = {
    'ENGINE': 'django.db.backends.postgresql',
    'NAME': 'staging_db',
    'USER': 'staging_user',
    'PASSWORD': 'staging_password',
    'HOST': 'staging-db.example.com',
    'PORT': '5432',
}

# Additional staging-specific settings
```

```python
# my_project/settings/production.py

from .base import *

DEBUG = False
ALLOWED_HOSTS = ['example.com']

DATABASES['default'] = {
    'ENGINE': 'django.db.backends.postgresql',
    'NAME': 'prod_db',
    'USER': 'prod_user',
    'PASSWORD': 'prod_password',
    'HOST': 'prod-db.example.com',
    'PORT': '5432',
}

# Secure settings for production
SECURE_SSL_REDIRECT = True
CSRF_COOKIE_SECURE = True
SESSION_COOKIE_SECURE = True

# Other production-specific settings (e.g., caching, security)
```

Why Traditional Methods Fall Short?

Database Configuration Example

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.getenv('DB_NAME'),
        'USER': os.getenv('DB_USER'),
        'PASSWORD': os.getenv('DB_PASSWORD'),
        'HOST': os.getenv('DB_HOST'),
        'PORT': os.getenv('DB_PORT'),
    }
}
```

Pydantic to the Rescue

- Validation
- Error Feedback
- Centralized and structured configuration management
- Json Schema

```python
from pydantic import BaseModel, Field

class DatabaseSettings(BaseModel):
    engine: str = Field(default="django.db.backends.postgresql")
    name: str
    user: str
    password: str
    host: str
    port: int

db_settings = DatabaseSettings(
    name=os.getenv('DB_NAME'),
    user=os.getenv('DB_USER'),
    password=os.getenv('DB_PASSWORD'),
    host=os.getenv('DB_HOST'),
    port=os.getenv('DB_PORT')
)
```

Pydantic-Settings

- Environment Variable Parsing Automatically
- Hierarchical Configuration Sources
- Validation on Startup
- Ease of Testing (.env)

Environment values Parsing and Dot env files integration

```python
from pydantic_settings import BaseSettings, SettingsConfigDict

class AppSettings(BaseSettings):
  app_name: str = "MyApp"  # Default value
  debug: bool = False  # Default value
  database_url: str  # No default, required to be set
  log_level: str = "INFO"

      @field_validator("log_level")
      def validate_log_level(cls, value):
        allowed_levels = {"DEBUG", "INFO", "WARNING", "ERROR"}
        if value not in allowed_levels:
          raise ValueError(f"log_level must be one of {allowed_levels}")
        return value


  model_config = SettingsConfigDict(
            env_file=".env",
            env_prefix="APP_",
            extra="forbid"
      )

# Initialize and load settings
try:
  settings = AppSettings()
except Exception as e:
  print(f"Configuration Error: {e}")
```

Hierarchical Configuration Sources

```python
@classmethod
def settings_customise_sources(
    cls,
    settings_cls: Type[BaseSettings],
    init_settings: PydanticBaseSettingsSource,
    env_settings: PydanticBaseSettingsSource,
    dotenv_settings: PydanticBaseSettingsSource,
    file_secret_settings: PydanticBaseSettingsSource,
) -> Tuple[PydanticBaseSettingsSource, ...]:
    """
    Customizes the order of input sources for the configuration.

    Args:
        settings_cls (Type[BaseSettings]): The class of the settings.
        init_settings (PydanticBaseSettingsSource): The initial settings source.
        env_settings (PydanticBaseSettingsSource): The environment settings source.
        dotenv_settings (PydanticBaseSettingsSource): The dotenv settings source.
        file_secret_settings (PydanticBaseSettingsSource): The file secret settings source.

    Returns:
        Tuple[PydanticBaseSettingsSource, ...]: A tuple of the customized settings sources.
    """
    return (
        init_settings,
        env_settings,
        dotenv_settings,
        YourSpecificSettingsSource(settings_cls),
        file_secret_settings,
    )
```

Validation error during startup

```
Traceback (most recent call last):
    File "/path/to/your/project/settings.py", line 10, in <module>
        db_settings = DatabaseSettings(
    File "pydantic/main.py", line 341, in __init__
        raise ValidationError(errors)
pydantic.error_wrappers.ValidationError: 1 validation error for DatabaseSettings

port
    value is not a valid integer (type=type_error.integer)
```

traditional vs. Pydantic-based

Aspect	Traditional Django	Pydantic
Error Timing	Runtime	Startup
Type Checking	Manual (or none)	Automatic
Error Messages	Vague	Precise
Environments Support	A mess	Easy to maintain

How to integrate pydantic-settings with Django settings?

One approach ...

```
#configuration

from pydantic_settings import BaseSettings, SettingsConfigDict

class DjangoSettings(BaseSettings):
  secret_key: str
  debug: bool = False
  allowed_hosts: list[str] = ["localhost"]
  database_url: str
  email_host: str
  email_port: int = 587
  email_user: str
  email_password: str

  model_config = SettingsConfigDict(env_file=".env", env_prefix="DJANGO_")
```

```
# settings.py

from .config import DjangoSettings

# Load validated settings
config = DjangoSettings()

# Assign Django settings
SECRET_KEY = config.secret_key
DEBUG = config.debug
ALLOWED_HOSTS = config.allowed_hosts

# Configure the database
DATABASES = {
  "default": {
      "ENGINE": "django.db.backends.postgresql",
      "NAME": config.database_url.split('/')[-1],
      "USER": config.database_url.split(':')[1][2:],
      "PASSWORD": config.database_url.split(':')[2].split('@')[0],
      "HOST": config.database_url.split('@')[-1].split(':')[0],
      "PORT": config.database_url.split(':')[-1],
  }
}

# Configure email
EMAIL_BACKEND = "django.core.mail.backends.smtp.EmailBackend"
EMAIL_HOST = config.email_host
EMAIL_PORT = config.email_port
EMAIL_HOST_USER = config.email_user
EMAIL_HOST_PASSWORD = config.email_password
EMAIL_USE_TLS = True
```

Second approach - Pydjantic

pydjantic.py

```python
def to_django(settings: BaseSettings):
    stack = inspect.stack()
    parent_frame = stack[1][0]

    def _get_actual_value(val: Any):
        if isinstance(val, BaseSettings):
            # for DATABASES and other complicated objects
            return _get_actual_value(val.model_dump())
        elif isinstance(val, dict):
            return {k: _get_actual_value(v) for k, v in val.items()}
        elif isinstance(val, list):
            return [_get_actual_value(item) for item in val]
        elif isinstance(val, SecretStr) or isinstance(val, SecretBytes):
            return val.get_secret_value()
        else:
            return val

    for key, value in settings.model_dump().items():
        parent_frame.f_locals[key] = _get_actual_value(value)
```

your_app/settings.py

```python
from typing import List

from pydantic_settings import BaseSettings, Field
from pydantic.fields import Undefined
from pydjantic import to_django

class GeneralSettings(BaseSettings):
    SECRET_KEY: str = Field(default=Undefined, env='DJANGO_SECRET_KEY')
    DEBUG: bool = Field(default=False, env='DEBUG')
    INSTALLED_APPS: List[str] = [
        'django.contrib.admin',
        'django.contrib.auth',
    ]
    LANGUAGE_CODE: str = 'en-us'
    USE_TZ: bool = True


class StaticSettings(BaseSettings):
    STATIC_URL: str = '/static/'
    STATIC_ROOT: str = 'staticfiles'


class SentrySettings(BaseSettings):
    SENTRY_DSN: str = Field(default=Undefined, env='SENTRY_DSN')


class ProjectSettings(GeneralSettings, StaticSettings, SentrySettings):
    pass


to_django(ProjectSettings())
```

Enhanced second approach

your_app/configuration.py

```
from typing import List

from pydantic import BaseSettings, Field
from pydantic.fields import Undefined
from pydjantic import to_django

class GeneralSettings(BaseSettings):
   SECRET_KEY: str = Field(default=Undefined, env='DJANGO_SECRET_KEY')
   DEBUG: bool = Field(default=False, env='DEBUG')

class SentrySettings(BaseSettings):
   SENTRY_DSN: str = Field(default=Undefined, env='SENTRY_DSN')


class ProjectSettings(GeneralSettings, SentrySettings):
   Pass
```

your_app/settings.py

```
from configuration  import ProjectSetting

to_django(ProjectSettings())


STATIC_URL: str = '/static/'
STATIC_ROOT: str = 'staticfiles'
INSTALLED_APPS: List[str] = [
      'django.contrib.admin',
      'django.contrib.auth',
]
LANGUAGE_CODE: str = 'en-us'
USE_TZ: bool = True
```

Resources connection Health Checks

```python
class OurBaseConfig(BaseSettings):
    model_config = SettingsConfigDict(
        env_nested_delimiter="__",
        env_file_encoding="utf-8",
        extra="forbid",
    )

    def run_all_health_checks(self, fields: Union[dict[str, Any], None] = None) -> None:
        """
        Run all health checks on the config parameters (including inherited configs) recursively
        """
        if fields is None:
            fields = copy.deepcopy(self.__dict__)

        for att in fields:
            self_att = getattr(self, att)

            if issubclass(type(self_att), HealthCheckMixin):
                ...

                self_att.health_check()

                if issubclass(type(self_att), OurBaseConfig):
                    self_att.run_all_health_checks()


class HealthCheckMixin:
    def health_check(self):
        raise NotImplementedError
```

```python
class RedisModel(BaseModel, HealthCheckMixin):
    ssl: bool
    port: int
    host: str
    database: int
    password: str
    timout: int = 3


    def health_check(self):
        """Check the availability of a Redis instance."""
        import redis
        logger = logging.getLogger(f"{self.__class__.__name__}_HealthCheck")

        logger.info(
            f"Running Redis health_check ..."
        )

        with redis.StrictRedis(
            host=self.host,
            port=self.port,
            db=self.database,
            password=self.password,
            ssl=self.ssl,
        ) as redis_client:
            redis_client.ping()

        logger.info("<<<<<<< Redis health_check execution Completed >>>>>>>>")
```

```python
def run_health_checks(config: BaseSettings):
    """
    This function performs health checks on configuration parameters.
    It logs the start and end of the health checks process
    and calls the `run_all_health_checks` method on the `config` object.

    Args:
        config (BaseSettings): An instance of the `BaseSettings` class or any class that inherits from it.
                               This object represents the configuration settings of an application.

    """

    logger.info("Health checks run on configuration's params - started")

    config().run_all_health_checks()
```

Works seamlessly across environments

Docker compose file - development environment

```
services:
my_service:
  ...
  <<: *dependencies
  healthcheck:
    test: ./run-health-checks || exit 1
    timeout: 3s
    retries: 1
```

ECS Health check in the task definition

```
{
 "containerDefinitions": [
  {
   "name": "my_service",
   ...
   "healthCheck": {
    "command": [
     "CMD-SHELL",
          "./run-health-checks >> /proc/1/fd/1 2>&1 || exit 1"
    ],
    "interval": 300,
    "timeout": 30,
    "retries": 3,
    "startPeriod": 40
   }
  }
 ]
}
```

Pydantic schema

Early validation and documentation

A schema is a JSON representation
of the structure and validation rules
of a Pydantic model

used for Validation and Documentation

Application’s settings

```python
from pydantic_settings import BaseSettings, SettingsConfigDict
from pydantic import Field, EmailStr

class AppSettings(BaseSettings):
    secret_key: str = Field(..., description="The secret key for the application.")
    debug: bool = Field(False, description="Enable or disable debug mode.")
    allowed_hosts: list[str] = Field(default_factory=lambda: ["localhost"], description="List of allowed hosts.")
    database_url: str = Field(..., description="Database connection string.")
    email_host: str = Field(..., description="SMTP server host.")
    email_port: int = Field(587, description="SMTP server port.")
    email_user: EmailStr = Field(..., description="SMTP server username.")
    email_password: str = Field(..., description="SMTP server password.")

    model_config = SettingsConfigDict(env_prefix="APP_")




schema = AppSettings.model_json_schema()
print(schema)
```

Output


```
{
 "title": "AppSettings",
 "type": "object",
 "properties": {
  "secret_key": {
   "type": "string",
   "title": "Secret Key",
   "description": "The secret key for the application."
  },
  "debug": {
   "type": "boolean",
   "title": "Debug",
   "default": false,
   "description": "Enable or disable debug mode."
  },
  "allowed_hosts": {
   "type": "array",
   "title": "Allowed Hosts",
   "description": "List of allowed hosts.",
   "items": {"type": "string"},
   "default": ["localhost"]
  },
  "database_url": {
   "type": "string",
   "title": "Database Url",
   "description": "Database connection string."
  },
  "email_host": {
   "type": "string",
   "title": "Email Host",
   "description": "SMTP server host."
  },
  "email_port": {
   "type": "integer",
   "title": "Email Port",
   "default": 587,
   "description": "SMTP server port."
  },
  "email_user": {
   "type": "string",
   "format": "email",
   "title": "Email User",
   "description": "SMTP server username."
  },
  "email_password": {
   "type": "string",
   "title": "Email Password",
   "description": "SMTP server password."
  }
 },
 "required": [
  "secret_key",
  "database_url",
  "email_host",
  "email_user",
  "email_password"
 ]
}
```

Pretty documentation Example

https://github.com/adobe/jsonschema2md

AppSettings Configuration Schema

Field Name	Type	Description	Default	Required
secret_key	string	The secret key for the application.	N/A	Yes
debug	boolean	Enable or disable debug mode.	false	No
allowed_hosts	array	List of allowed hosts.	["localhost"]	No

Integrating Pydantic with Django Settings

Summary

📈 Advanced Validation

✅ prevent runtime errors

👩‍💻 Boosted Productivity

🔗 Seamless Integration

📜 Standardization

🛡️ health checks

📄 JSON Schema Support

Thank you

bluevine